UNIVERSITY OF NEWCASTLE UPON TYNE
DEPARTMENT OF COMPUTING SCIENCE

PhD Thesis

Run-time Support for Parallel Object-Oriented Computing

The NIP Lazy Task Creation Technique and
the NIP Object-based Software Distributed Shared Memory

by
Savas Parastatidis

Supervisor
Dr. Paul Watson

SEPTEMBER 2000

ABSTRACT

Advances in hardware technologies combined with decreased costs have started a trend towards massively parallel architectures that utilise commodity components. It is thought unreasonable to expect software developers to manage the high degree of parallelism that is made available by these architectures. This thesis argues that a new programming model is essential for the development of parallel applications and presents a model which embraces the notions of object-orientation and implicit identification of parallelism. The new model allows software engineers to concentrate on development issues, using the object-oriented paradigm, whilst being freed from the burden of explicitly managing parallel activity.

To support the programming model, the semantics of an execution model are defined and implemented as part of a run-time support system for object-oriented parallel applications. Details of the novel techniques from the run-time system, in the areas of lazy task creation and object-based, distributed shared memory, are presented.

The tasklet construct for representing potentially parallel computation is introduced and further developed by this thesis. Three caching techniques that take advantage of memory access patterns exhibited in object-oriented applications are explored. Finally, the performance characteristics of the introduced run-time techniques are analysed through a number of benchmark applications.

ACKNOWLEDGEMENTS

I would like to start by expressing my gratitude to my supervisor, Dr. Paul Watson. I consider myself extremely fortuned to have had Dr. Watson as my guide in this research work. He has endured numerous long sessions of discussions without ever showing signs of tiredness and his insights were always accurate. He has been a great source of inspiration and support throughout. I do not think I would have ever been able to bring this thesis to completion if it was not for his paradigmatic supervision. Thank you Paul.

Next, I would like to thank the other two members of my thesis committee Prof. Pete Lee and Dr. Chris Phillips for their valuable input to my research work. Special thanks to Prof. Lee because, in his capacity as leader for the HiPPO research project, he has been extremely patient with me whilst trying to complete this thesis.

My friend, fellow PhD student, and officemate Jim Webber deserves a special mention. His company and support through the years have made this work possible. He has contributed greatly to my integration into British society, culture, and to the more colloquial aspects of the language. I consider myself tremendously lucky to have met him and privileged to be able to call him 'mate.'

To other friends: Kostas Amoiridis, Fefi Axiotidou, Grigoris Lampakis, Panagiotis Karambatzakis, Dr. Lindsay Marshall, Baggelis Skarlatos, Bassilis Theodoridis, Stamatis Xouxos, Colette Wabnitz, I offer my sincerest thanks.

Finally, I would like to dedicate this thesis to my family: my mother Fotini, my father Giorgos, and my brother Filippos. If it were not for

their continuing encouragement and support throughout the years, I would not have managed to complete my academic studies and this research work. Τους ευχαριστώ πολύ.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

TABLE OF CODES

CHAPTER 1

INTRODUCTION

The research work presented in this thesis was inspired by the emergence of high-performance computing architectures built around affordable, commodity-based hardware. The assessment of existing software-based, run-time support tools for the execution of parallel applications on such architectures and the proposition of solutions to possible drawbacks were originally set as the primary objectives for this thesis.

This chapter presents in detail the motivation for the research work undertaken and lists the contributions to knowledge that the rest of the thesis claims to make. The field of high-performance computing is explored and the application area and hardware trends are studied.

The discipline of parallelism is seen as being very closely interrelated to high-performance computing. The basic requirements for development models and software tools to support parallelism on the emerging high-performance architectures are set and the scope for the chapters that follow is established.

1.1. High-Performance Computing

There have always been applications with requirements that exceeded the available computational power at any particular period in time. The effort to meet the needs of these performance-hungry applications has been the driving force in designing and building faster computers.

In the early years of high-performance computing (HPC), applications with excessive demands in processing power were limited to the area of scientific computation (weather prediction, cosmology, particle simulation, etc.). As the processing power made available to scientists increased, the urge to run larger and/or more difficult problems continued and it is still the same today. In addition to scientists, computer practitioners in other application areas started to realise the benefits of harnessing more processing power. Applications in areas such computer vision and graphics, computer-aided design, databases, have all benefited from high-performance computing architectures since the '80s—and still do. Today, performance-critical application areas include computer animation (in 1995, 'Toy Story' was the first full-featured, computer-generated film with many others following it), aerospace, geophysics, World Wide Web (WWW), gaming, finance, bioinformatics, education, healthcare, to name only a few. One can observe a shift from scientific to primarily industrially oriented HPC applications.

The last statement is further supported by the recent growth of the WWW. Performance-critical web applications have emphasised the demand for industry-based HPC platforms. As an example, one only needs to examine the strategies for their future products that the two largest software companies in the world, namely Microsoft and Oracle, have drawn. Both are suggesting a return to the old server-based style of computing (Microsoft 2000; Oracle 1999). The requirements of the multimedia-rich, computationally intensive, network-centric applications that are to be supported by the re-proposed server-based models of computing point to HPC architectures.

With the majority of the application areas being industry-oriented during the last decade, the interest in the HPC field and the high rate of investment were unsurprising. As it will soon be shown, systems with an improved cost/performance ratio attracted most of the attention and architectures based on commodity hardware are becoming more popular due to their ability to offer good performance for a relative low cost.

A study of the HPC field cannot and should not neglect the required application development models and support tools as well as their enabling execution environments.

Additionally, the discipline of parallelism is seen by this thesis as being closely coupled with that of HPC, as the foremost purpose of parallelism is the improvement of performance. To that extent, this thesis considers different models and support tools for parallelism as being the enabling factors for high-performance computing.

Before the discussion moves to parallelism related issues, the application area and hardware trends in the field of high-performance computing are considered. It is hoped that the current and future requirements for development and execution paradigms will be better understood.

1.1.1. Application Area Trend

Analysis of the list of the 500 fastest supercomputers in the world (TOP500 List Authors 2000) reveals a number of trends in the HPC field. The 'TOP500 list' (TOP500 List 2000) is published twice a year, every June and November, and it records the fastest computers in the world based on the results obtained from the execution of the LINPACK benchmark (Dongarra 1994).

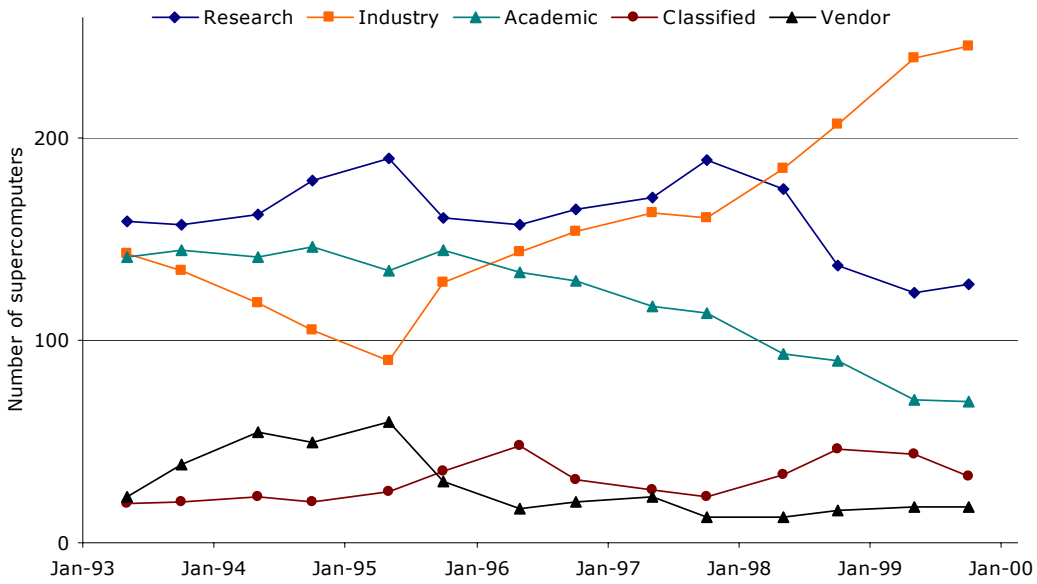


Figure 1-1: Application area evolution of the top 500 high-performance computers based on data from the 'TOP500 list' (TOP500 List 2000; TOP500 List Authors 2000)

The graph of Figure 1-1, which is based on all the published TOP500 lists until November 1999, is illustrative of the transition towards industry-driven high-performance computing. In 1996, the number of computers in the TOP500 list used in the industry exceeded those installed in the academia and two years later those used by research institutes. However, while the graph of Figure 1-1 strengthens the argument about the growing interest of industry in high-performance computing, it does not reveal any particular characteristics about the TOP500 systems.

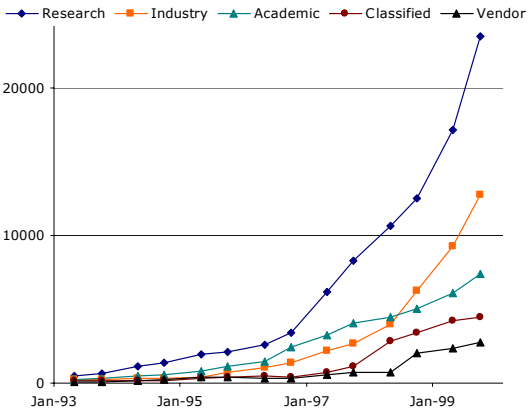


Figure 1-2: Evolution of the total computational power (in Gflop/s) per application area of the top 500 high-performance computers based on data from the 'TOP500 list' (TOP500 List 2000; TOP500 List Authors 2000)

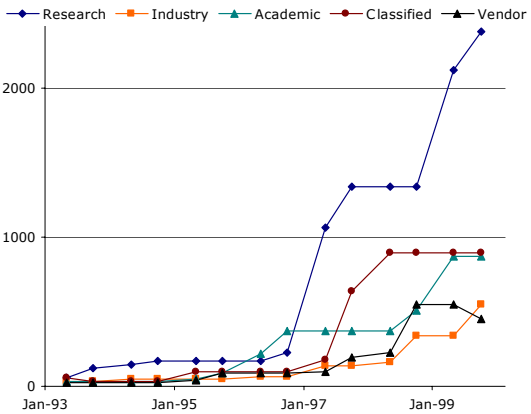


Figure 1-3: Evolution of the computational power (in Gflop/s) of the fastest computer for each application area of the top 500 high-performance computers based on data from the 'TOP500 list' (TOP500 List 2000; TOP500 List Authors 2000)

When the achieved computational power, expressed in Gflop/s[1], of all the supercomputers in the TOP500 list is added together on per application area basis, it is revealed that the industry-based systems are lagging behind in overall performance when compared to the systems employed by the other application areas (Figure 1-2). Despite their great number, as shown in Figure 1-1, the total computational power of the industry-installed TOP500 high-performance computers at the end of 1999 is almost half of the equivalent computational power of research-based supercomputers (Figure 1-2). As a result, the high-performance computers in the TOP500 list that are used in industry are the slowest on average when compared to any other application area.

The investment in exceptionally fast, number-crunching platforms seems to be exclusively research-driven, as suggested by Figure 1-3 which shows the evolution of the fastest of the supercomputers in the TOP500 list for each application area. Again, it is clear that industry does not invest in the highest performing solutions possible. Instead, there is an indication that industry is mostly concerned with the cost/performance ratio.

Although costing information about the supercomputers in the TOP500 list is not made available, it is safe to assume that in the general case the cost/performance ratio becomes significantly high at the top of the TOP500 list. The excessive computational requirements of the scientific applications result in investments on fewer but at the same time faster high-performance architectures. It is for this reason the number of supercomputers in research has been decreasing (Figure 1-1) while their performance has been continuously increasing (Figure 1-2 and Figure 1-3). In contrast and despite the manifested interest of industry in high-performance computing, the lack of

[1] Flop (Floating Operations): Unit used in the measurement of computational power.

industry-installed supercomputers from the top of the TOP500 list suggests that the investment on new platforms is not purely performance-driven but, instead, cost/performance-driven.

1.1.2. Architectural Trends

The computer industry has been experiencing a tremendous rate of advances in VLSI technology, especially during the last decade. Microprocessors and memory have been becoming faster and cheaper, interconnection networks have been being built with more available bandwidth and shorter latency, and local and wide area networking technology has been advancing, mostly due to the exponential growth of the Internet.

Massively Parallel Processing (MPP) and Symmetric Multi-Processing (SMP) computers are now much cheaper to manufacture because they can be based partially or even completely on general-purpose, commodity hardware. An MPP consists of a great number of processing elements with their own private memory. The processing nodes may be interconnected through a variety of existing topologies (*e.g.*, rings, buses, cubes, hyper-cubes, etc.). An SMP consists of a usually small number—when compared to MPP architectures—of processing elements that all share the same physical memory.

If the reasoning of the previous section about the shift towards inexpensive high-performance computing was accurate, then MPP and SMP architectures should lead the TOP500 list in terms of numbers. Indeed, the graph of Figure 1-4 confirms the latter hypothesis by presenting the evolution of the TOP500 hardware architectures since November 1993.

Due to the scalability limitations of the SMP architectures, further improvements in performance are usually difficult and/or extremely costly to achieve. It can be suggested that the combination of the costing and scalability considerations is the reason the number of SMP architectures is in decline. Furthermore, the eclipse of architectures based on Single Instruction Multiple Data (SIMD) processors or just one very expensive but specialised single processor (Figure 1-4) can also be attributed to their unfavourable cost/performance ratio and their scalability limitations.

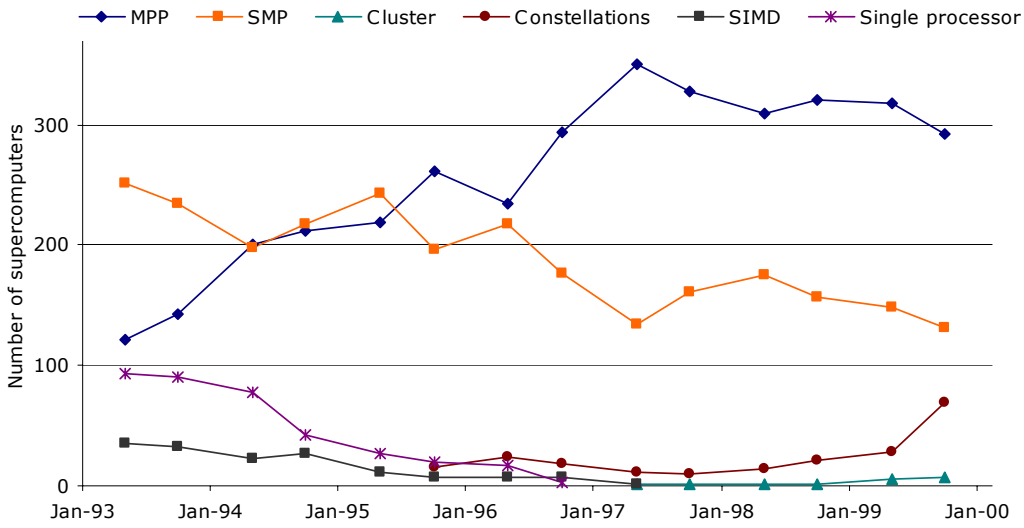


Figure 1-4: Evolution of hardware architecture of the top 500 high-performance computers based on data from the 'TOP500 list' (TOP500 List 2000; TOP500 List Authors 2000)

In contrast, the architectural shift towards MPP architectures can be justified by their excellent scalability and the achieved performance. Since the first publication of the TOP500 list, an MPP supercomputer has always been the fastest. Figure 1-5 shows the evolution of the computational power, in Gflop/s, of the best performing supercomputer per architecture. The fastest computer in the world in November 1999, according to the TOP500 list, was the ASCI Red, located at the Sandia National Labs, US, which consisted of an impressive number of 9,472 Intel PentiumII™ processors. The ASCI Red is probably the best demonstration of the way commodity hardware—in this case the PentiumII™ microprocessors—can be incorporated in a supercomputer that is capable of achieving record-breaking performances.

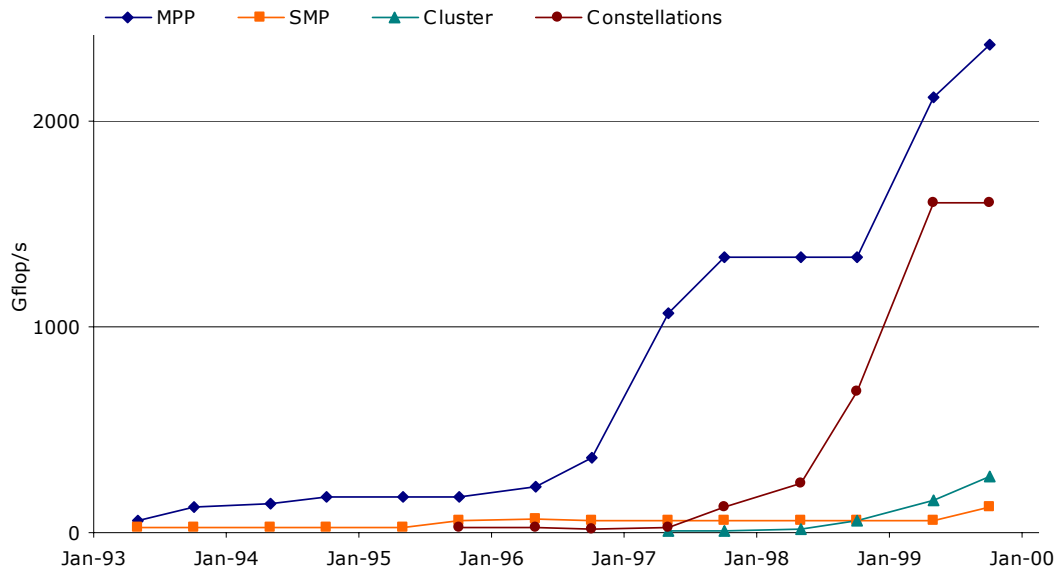


Figure 1-5: Evolution of the computational power (in Gflop/s) of the fastest computer for each hardware architecture of the top 500 high-performance computers based on data from the 'TOP500 list' (TOP500 List 2000; TOP500 List Authors 2000)

The shift towards commodity-based HPC is also indicated by the relatively recent appearance in the TOP500 list of a new type of computer architecture that is based on numbers of workstations interconnected by fast interconnection networks. The cluster architecture, as it is known, has been gaining momentum mostly due to its extremely advantageous cost/performance ratio.

Finally, the general interest for inexpensive HPC is also indicated by the increasing popularity of constellation architectures. Such architectures leverage collections of existing, most often older, supercomputers for the execution of high-performance applications. It should not be of any surprise that in November 1999 industry was the main user of constellations due to the cost savings that can be achieved from the reuse of old platforms.

1.1.3. Trend Synopsis

It is evident that in the last few years, industry has emerged as the main user of HPC. Computational power requirements are no longer exclusive to scientific-oriented applications. The field of general-purpose, high-performance computing is gaining momentum, as the list of HPC applications suggests (page 1). The notable increase in the number of TOP500 supercomputers used in industry is yet another indication of the great interest and investment in HPC. It seems, though, that the focus is mainly on the cost/performance ratio of the platforms employed rather than just on performance. That would explain the inferior computational power that is achieved by HPC platforms used

in industry when compared to those used in the other application areas and especially in research.

In contrast, due to the excessive computational requirements of scientific applications, the maximum achievable performance is pursued for the HPC systems that are deployed in research. Often, the focus on performance dramatically increases the implementation and support costs for a platform and, as a result, there has to be a concentration on fewer but faster installations.

There also seems to be a drive towards HPC systems that are based, partially or completely, on commodity hardware, especially off-the-self microprocessors. The incorporation of commodity hardware into HPC results in more affordable systems without necessarily sacrificing their performance.

Finally, it is clear that parallelism is the main enabling factor for HPC. The prime goal of parallelism is to improve the execution performance of applications. Since supercomputer architectures based on single, specialised, expensive processors have been eclipsed (Figure 1-4), it is reasonable to consider parallelism as the facilitating force for high-performance computing.

All the current HPC architectures are built upon aggregations of processing elements that collaborate. The definition of a processing element may vary between HPC architectures. For example, it may be a microprocessor, a whole workstation, an SMP computer, or even a supercomputer in the case of constellations. Nevertheless, in any case the concept is the same: a number of processing-capable components that work simultaneously having as a purpose the faster execution of an application.

Having recognised the importance of parallelism in the HPC field, this thesis embarks in the investigation of issues related to the field of parallel computing.

1.2. Parallelism

For almost six decades, the computer model based on a single processor, memory, bus, and peripherals—the von Neumann architecture (Burks et al. 1962)—has been the dominant architectural model upon which computers have been built. Nevertheless, even during the '60s, the HPC community began to realise that traditional computers based on the von Neumann model were unable to produce the computational power required. With parallelism, the traditional model of computing could be extended to achieve better performance.

1.2.1. Definition

According to Pfister (Pfister 1998), there are three ways to perform an activity faster in life: work harder, work smarter, or get help. Pfister observes an analogy with the computer world (Table 1-1). The execution performance of applications may be improved when the components of the computer architecture (*e.g.*, processor, memory, bus, etc.) are enhanced. Additionally, an implementation of the same application based on a better algorithm may also improve the execution performance. However, if more than one processing elements work on the same problem, execution performance may be dramatically increased when compared to the two previous approaches.

Work harder	-	Processor speed
Work smarter	-	Algorithms
Get help	-	Parallel processing

Table 1-1: How to perform an activity faster in real life and in computing (Pfister 1998)

Almasi and Gottlieb define a parallel computer as *"a large collection of processing elements that can communicate and cooperate to solve large problems fast"* (Almasi and Gottlieb 1994). However, as they also observe, the definition raises many questions: How large should the collection of processing elements be? How do they communicate? What means do they use to cooperate? How large a problem should be? In answering these questions, computer architects and software developers have come up with a variety of solutions since the '60s, descriptions of which can be found in (Almasi and Gottlieb 1994; Culler and Singh 1999; Lewis and El-Rewini 1992; Pfister 1998; Tanenbaum 1999).

Finally, based on the definition by Almasi and Gottlieb, Culler and Singh believe that a parallel architecture is just *"the extension of conventional computer architecture to address issues of communication and cooperation among processing elements"* (Culler et al. 1993b). However, this definition does not capture the essence of parallel computation, which according to this thesis is speedup. As it was suggested earlier in this chapter, the foremost purpose of parallelism is the faster execution of applications and for that reason the definition by Almasi and Gottlied is perceived as more accurate for the purposes of this thesis.

1.2.2. Hardware Support

Since the 60's, there has been a plethora of research works and considerable progress in the field of parallel computer architecture. The rate of computational power increase, as shown by graphs presented in the previous section, is an indication of the strong hardware developments in the field. The four decades of advances have resulted in a great

diversity of hardware platforms, as the discussion in Chapter 2 demonstrates where a taxonomy of existing architectures is also presented.

One may suggest, though, that despite the advances in hardware, application development and run-time support tools for the various parallel platforms have not received the same attention.

1.2.3. Programming and Execution Models

Tanenbaum describes a computer system in terms of a series of layers (Tanenbaum 1999). At the bottom of the organisational structure, there is the hardware layer and at the top the application layer. In between, layers like the programming language and the operating system exist. A layer in the organisational structure of a computer system should provide a simple, well-defined abstraction of the underlying ones (Chapter 2 presents a more detailed discussion on the layered approach to parallel computer system organisation). Perhaps more than other disciplines, in the case of parallel computing the performance is an extremely important characteristic of a layer. Consequently, it should be noted that efficiency should not be sacrificed in favour of abstraction.

Unfortunately, after observing the evolution of parallelism since the '60s, one notices that in practise there have not been significant advancements in the layered approach to parallel computer organisation. Parallel computing practitioners have been reluctant to explore new approaches to development and run-time support, as the hesitation in adopting innovative methods of programming and execution of applications on parallel computers suggests. The same troublesome—according to this thesis—methods have been used for decades now.

It is suggested that the absence of a clear distinction between a programming model and an execution model of parallel computing is to blame for the apparent lack of advancement in parallel computing practices. The former model represents the view of the parallel system as it is seen by the developer while the latter is the abstraction of a parallel system as it is perceived by the application. The two models are going to be defined and discussed in detail in Chapter 2.

The lack of a clear distinction between the programming and execution models has allowed hardware characteristics to be filtered up to the application developer. Programmers are required to write applications with the hardware configuration always in mind. The management of parallelism, synchronisation, communication, and other related issues, burden the application developers. As a result, the parallel software development and maintenance processes have become troublesome, time-consuming, and costly.

The most popular models of parallel computing are message-passing and shared-memory. The two are considered both as programming and execution models. Developers have to reason about the implementation of algorithms with the architectural characteristics of the parallel platforms in mind. There is no attempt by either of the two models to hide architectural details from the programmers. There are, however, programming models that have been designed to abstract from the underlying architectures, such as the functional model. A synopsis of the main characteristics of all the above models are given below (a more detailed discussion is presented in Chapter 2):

- The message-passing programming and execution model do not provide an abstraction of the underlying parallel architecture. Application developers have to manage communication and synchronisation between processing elements. In most cases and when the programming primitives of the model are used appropriately, efficiency is not an issue. Modern operating systems, often in combination with optimised user-level libraries that vendors of parallel systems supply, provide good run-time support. However, using the message-passing programming primitives correctly in order to better utilise the parallel architecture is a huge task for the application developer.
- The abstraction that the shared-memory model provides is that of a computer with a collection of processing elements that have access to a common memory. The architecture of the underlying parallel system is not hidden from application developers, as they have to manage parallelism. As in the case of the message-passing model, efficiency can be achieved when the programming primitives of the model are used suitably for a specific architecture and with the appropriate operating system and/or user-level libraries support.
- Unlike the two previous models, the functional programming model manages to hide the details of the underlying parallel architecture. It requires an execution model that is not made visible to application developers, who are not burdened with the task of managing parallelism but, instead, they only need to concentrate on algorithmic issues. However, the model greatly depends on software tools (i.e., compilers, run-time systems) that have not been able to match the performance of the tools available for the message-passing and shared-memory models. The functional programming model appeared to be good alternative to the traditional models but it has not managed to gain a sufficient following in order to become a commercial success (Almasi and Gottlieb 1994).

It can also be suggested that the existing programming models have poor support for good software engineering practices (some to a lesser degree than others) like abstraction for managing complexity, structure reuse, maintenance, profiling/debugging, which are very important for the application development lifecycle.

1.3. Possible Future Directions for Parallel Computing

An attempt to determine the future directions of parallel computing may provide an indication of those initiatives that must be undertaken today. It is not the purpose of this thesis to discuss and attempt to predict the future of parallel computing. Instead, using as a starting point the observations that have been made up to now in this chapter and based on the findings of a working group on 'Enabling Technologies for Petaflops Computing' (Sterling et al. 1995), this section attempts to identify those areas that are likely to influence the field of parallel computing in the future.

1.3.1. Microprocessor Technology

There are indications that the remarkable rate of advances in microprocessor technology will be reduced as physical limits in their production process are reached. The vast manufacturing costs will probably yield the current processor manufacturing technologies commercially unviable (Moore 1965; Moore 1997; Moore 1998).

Exotic technologies are under investigation and they might provide alternatives to HPC architectures. For example, research on the use of technologies based on optics and superconductivity in processor architecture promise enormous availability of computational power. However, such technologies are unlikely to make an impact during the next two decades. Other promising areas of research such molecular and quantum computing are even more unlikely to make an impression in the near future. The findings of the working group on Petaflops computing confirm these observations (Sterling et al. 1995).

The use of huge numbers of microprocessors in massively parallel computer architectures is likely to become standard practise in the HPC field. Cost concerns appear to favour the use of commodity rather than special purpose microprocessors in such architectures. The hardware trends observed earlier in this chapter seem to confirm the last statement. Furthermore, in the analysis of the findings of the working group on 'Enabling Technologies for Petaflops Computing,' Sterling, Messina, and Smith make a very important remark that strengthens the arguments toward the use of commodity

processors. The analysis also hints at the use of other commodity components in addition to microprocessors (*e.g.*, network interconnections, memory, etc.):

> *"The level of investment being applied to technology development by the commercial semiconductor marketplace is substantial and greatly exceeds any augmentation likely from government research programs. Thus, the opportunity to influence expensive development cycles is limited. This situation is exacerbated by the tight coupling between mass production and component cost. Specialty parts become significantly more costly than mass-produced commodity parts of equivalent complexity. Consequently, any initiative to develop a Petaflops computer will have to rely heavily (although not exclusively) on commercially-available components. By leveraging advances that occur as commercial by-products, development costs can be acceptable."* (Sterling et al. 1995)

1.3.2. Architectures

The findings of the working group on 'Enabling Technologies for Petaflops Computing' (Sterling et al. 1995) reinforce the argument that was presented earlier in this chapter about the shift towards parallel computer architectures that are focused on cost/performance ratio. The use of commodity hardware on such architectures is essential if the manufacturing costs are to be kept down. In (Sterling et al. 1995), it is also predicted that there will be a shift towards massively parallel architectures with tens of thousands processors. The hardware trend of Figure 1-4 (page 5) confirms the prediction.

However, the move towards architectures with massive numbers of processors will probably require new techniques to emerge, especially in the field of memory technology (Sterling et al. 1995). Such techniques will attempt to eliminate the latency in memory access, improve memory bandwidth, allow for faster communication between processing elements, etc.

1.3.3. Software

The authors of 'Enabling Technologies for Petaflops Computing' reveal the lack of good programming models and software tools for parallel computing (Sterling et al. 1995). They predicted that for the Petaflops mark to be achieved, a degree of parallelism of up to one million would be required. Programmers cannot be expected to manage that degree of parallelism without adequate support from software.

In (Sterling et al. 1995), a series of areas where developments are required is proposed. The subset that is relevant to the topic of this thesis is presented bellow.

- **Global address space:** The available memory on a massively parallel architecture should be accessible in a uniform way.
- **Latency hiding:** Either via software or hardware means, the latency in memory operations should be hidden.
- **Implicit and explicit parallelism:** New models for developing parallel applications are required that allow programmers to concentrate on algorithm issues rather than the management of parallelism.
- **Software support tools:** New support tools for transparent resource management and automatic migration of data and tasks are required.

This thesis attempts to reflect on the requirements of software tools for well-organised, cost-effective, parallel application development and for efficient execution on the parallel architectures of the future.

1.4. Research Goals

The discussion to this point attempted to sketch the current state of parallel computing and hint on potential future developments. A need for new parallel programming and execution models and their required software support tools emerged. Furthermore, as the complexity of (parallel) applications grows, the implementation, debugging, and profiling stages of the application development process become increasingly troublesome. It is apparent that good software programming practices need be introduced in the development lifecycle of parallel applications.

In this section, the motivation for the research work undertaken is discussed. The objectives that were set and their augmentation are also presented. Finally, the contributions to knowledge that this thesis claims to make are outlined.

1.4.1. Motivation

This research work was originally inspired by the emergence of cluster architectures based on commodity hardware as parallel platforms. The considered clusters consisted of single-processor workstations that were interconnected via commodity networking equipment. The investigation of run-time techniques for the execution of parallel object-oriented applications on such affordable platforms was originally set as the primary objective. The main characteristics of the targeted platforms were their distributed, non-shared memory and their slow interconnections.

However, it was clear even during the early stages of the research work that the original objectives were too narrow. There was no reason why the run-time techniques under investigation could not target other high-performance architectures as well. As a result, the goals were extended to target shared-memory multiprocessors and clusters of single-processor and/or multiprocessor architectures. The investigation into programming and execution models for parallel computing was also included in the set of objectives. The exploration of run-time techniques for parallel computing on shared- and distributed-memory architectures was maintained as part of the research goals.

1.4.2. Contributions to Knowledge

The exploration of the research goals resulted in the original work that the rest of this thesis presents. The list of the contributions to knowledge together with a synopsis of the main findings is presented bellow.

NIP Programming Model

The NIP programming model attempts to satisfy the requirements of the parallel application development process as those were identified in this chapter and as Chapter 2 discusses in more detail. The main characteristics of the model are the abstraction from any underlying architectural details. The focus of the NIP programming model is on implicit parallelism and object-orientation. The programmer is not burdened with the tasks of identifying and managing parallelism in an application. Instead, developers are presented with a programming model that allows them to concentrate on the exploration of algorithmic issues only.

The NIP programming model combines the benefits of two programming paradigms, namely functional and object-orientation. The model is based on a previously introduced *functions plus objects* paradigm (Sargeant 1993).

NIP Execution Model

The NIP execution model was originally designed to provide the semantics around which run-time environments for the NIP programming model could be implemented. Nevertheless, the semantics of the model are such that even non-implicitly parallel programming models could target it through appropriate software tool support.

The main characteristic of the model is the abstraction from the underlying hardware details. The NIP execution model provides a set of features for the management of parallelism, like automatic exploitation of computational resources, implicit memory

access synchronisation, distributed shared memory abstraction. Furthermore, the model defines that the memory is structured as objects.

NIP Lazy Task Creation and Tasklets

The NIP run-time system is an implementation of the NIP execution semantics. It employs the novel NIP lazy task creation technique to manage the excessive degree of parallelism that many applications exhibit. NIP lazy task creation is based around the *tasklet*, which is a newly introduced construct for the representation of potentially parallel tasks. The concept of a tasklet was first proposed by (Watson 1996) and it is further developed in this thesis.

NIP parallel applications are expected to identify, rather than create, parallel tasks using the tasklet construct. The NIP run-time system does not create tasks from tasklets unless there are available computational resources, a technique also known as lazy task creation. The main advantage of NIP lazy task creation over previous approaches is the tasklet. Unlike previous approaches, just a single tasklet can represent entire iterative or recursive computations from which parallelism can be extracted.

NIP Distributed Shared Memory

The NIP run-time system incorporates an all-in-software, object-based, distributed shared memory system. The NIPDSM implements the memory semantics of the NIP execution model. In addition, NIPDSM provides an enhanced approach to maintaining the consistency of objects in distributed memory architectures and features three novel caching techniques. For the first time, an object-based distributed shared memory system attempts to exploit both temporal and spatial locality in memory access without suffering from the problems like false sharing that appear in previous approaches. Additional information about object-based structures may be exploited to further enhance the performance of the memory system. Memory access patterns may be defined using associations between objects and recurring operations on objects are automatically captured and used to improve caching.

1.5. Remaining Thesis Structure

The remaining of this thesis explores each of the research goals and analytically describes the findings of this research work.

Chapter 2 deals with some background material on parallel computer architectures, programming and execution models and their support tools. Then, the discussion focuses on the NIP programming and execution models.

Chapter 3 thoroughly considers existing lazy task creation techniques in an attempt to clearly reveal their drawbacks but also identify their strengths. Then, the NIP lazy task creation technique and the tasklet construct are introduced and described in detail.

Chapter 4 introduces the field of software-based, distributed shared memory. The main characteristics and issues of such systems are discussed before the NIPDSM system is comprehensively explored.

Chapter 5 presents a description of the NIP run-time system, an implementation of the NIP execution model semantics that incorporates the NIP lazy task creation technique and the NIPDSM. The major components of the NIP run-time are examined.

Chapter 6 includes the results and analysis of the performance evaluation process of the NIP run-time system. A number of micro-benchmarks, small programs, and a scientific application are used to show the behaviour of the various components of the run-time system.

Finally, Chapter 7 concludes the discussion with a summary of the research work findings.

CHAPTER 2

PARALLELISM AND THE NIP PROGRAMMING AND NIP EXECUTION MODELS

The aim of this chapter is to introduce the NIP programming and execution models. In doing so, background material on parallel computer architectures, programming languages for parallel computing, and parallel program execution techniques is presented.

A layered approach to the parallel computing paradigm is considered after examining the idiosyncrasies, advantages, and disadvantages of some popular programming and execution models. The distinct elements that make a programming and an execution model are identified. Issues related to programming languages, run-time systems, operating systems, and hardware architectures for parallel computing are considered.

The discussion leads to the description of the NIP programming and execution models for parallel computing. The proposed models form a new approach to combining good software engineering practices, implicit parallelism, and efficient execution in parallel computing.

2.1. Models and Abstraction

There has been a misuse of the terms 'programming model,' 'execution model,' and 'computational model' in the computing literature. The terms have been used interchangeably in many different works. Earlier works leading to this thesis also misused the term 'computational model' (Watson and Parastatidis 1999a; Watson and Parastatidis 1999b). This thesis argues that the terms describe different levels of abstraction of a computer system and for that reason they cannot be used interchangeably. It is imperative that the differences between the levels of abstraction implied by the three models are established before the tools and techniques available for parallel programming and run-time support can be effectively explored.

2.1.1. Programming Model

A programming model is a methodology for the implementation of algorithms. It outlines the features and limitations of a computer system as they are presented to application developers, with all, some, or none of the hardware details hidden. The collection of the characteristics and the implied behaviour of a model (i.e., rules, limits, features, etc.) constitute its semantics. The semantics of a programming model are realised through appropriate software support tools that (*e.g.*, a compiler for a particular programming language and perhaps run-time libraries).

Pfister describes a programming model as *"the all-pervasive atmosphere, the internalised set of assumptions about how a computer works that imbue every program written for that computer"* (Pfister 1998). Culler and Singh understand the programming model as *"the conceptualisation of the machine that the programmer uses in coding applications"* (Culler and Singh 1999). Others have similar approaches to giving a definition to the term 'programming model' (Skillicorn and Talia 1998).

2.1.2. Execution Model

An execution model is the description of the services available for the execution of applications on a computer system. While a programming model is the conceptualisation of a computer system as presented to application developers, an execution model outlines the specifications and behaviour of an abstract architecture for the benefit of the programming model support tools (*e.g.*, programming language compilers). A programming model may hide the available primitives for the execution of applications from developers (*e.g.*, the NIP programming model, which is described in Section 2.8) or

it may incorporate them (*e.g.*, message passing, shared memory). The semantics of an execution model may be implemented by the run-time system, the operating system, or the hardware.

The abstract machine implied by an execution model need not have the same architectural characteristics as the underlying hardware. For example, an execution model offering shared memory semantics does not necessarily indicate the existence of hardware with physically shared memory. Furthermore, application execution techniques that are used to implement the semantics of a model need not be part of the abstract machine specifications. For example, the NIP lazy task creation technique (Chapter 3) and the NIP distributed shared-memory system (Chapter 4) are implemented as run-time tools to support the behaviour of the NIP abstract machine, which is defined by the NIP execution model (Section 2.9).

2.1.3. Computational Model

A computational model thoroughly defines the behaviour of applications when executed on specific computer systems. Unlike the programming and execution models, a computational model is not an abstract representation of a computer system. Instead, it is a mathematical tool for analysing the performance and behavioural characteristics of an application running on a computer system with known specifications. Computational models assist developers in refining the implementation of their algorithms in an attempt to achieve better performance on particular architectures. A computational model can be seen as the combination of the programming model and the execution model.

Almasi and Gottlieb consider two aspects to a computational model: (a) a tool that assists in the design process of an algorithm, and (b) a tool that provides a way to mathematically analyse and improve the efficiency of algorithms on the architectures that the model targets. Using the known costs of the exposed primitive operations (*e.g.*, operating system calls, communication costs, etc.) for a particular computer system, Almasi and Gottlieb propose a methodical approach to optimising the efficiency of an algorithm for the specific programming and execution models used. However, there exist abstract programming and execution models that hide the operations of a computer system. In such cases, information about primitive hardware or software operations is not available and, therefore, efficiency analysis of the execution of an application cannot take place. Indeed, Skillicorn and Talia (Skillicorn and Talia 1998) agree that some programming models are unsuitable, due to their abstract nature, for performance analysis of algorithms.

This thesis does not consider computational models to any further extent. The abstract nature of the NIP programming and execution models that are proposed later in this chapter does not facilitate the use of any computational model. However, as this thesis discusses issues related to parallelism, it should be mentioned that there have been several efforts to create parallel computational models. The Parallel Random Access Machine (PRAM) (Fortune and Wyllie 1978) was amongst the first and most widely used ones but it made unrealistic assumptions about the communication features of the computer architecture. Examples of other models proposed are the Bulk-Synchronous Parallel model (Valiant 1990), Message-Passing Block PRAM (Agarwal et al. 1989), and LogP (Culler et al. 1993b; Culler et al. 1996).

2.1.4. Lack of Abstraction Causes Confusion

In the discipline of parallelism, the terms 'message passing' and 'shared memory' have been used to describe (a) computer architecture models, (b) execution models, and (c) programming models. Due to the lack of abstraction in early parallel systems and parallel application development methodologies, the hardware architecture was reflected at the programming model, making the distinction between programming and execution models unnecessary. Today, as several software layers may abstract the hardware architecture, the distinction between programming and execution models becomes important for the study of parallelism.

A message passing architecture, for example, requires that messages be used for the communication between the processing elements of a computer system. A message passing execution model suggests that parallel computations use messages to communicate with each other without, however, implying that the underlying architecture is a message passing architecture or that messages are used in the programming model. Message-passing programming models assist application developers in designing and implementing an algorithm with the notion of messages as the means for data exchange between components of the application but it is not necessarily the case that a message passing execution model and architecture exist. It is clear that although there are three different levels of abstraction, the same term, 'message passing,' is used for all, causing confusion.

2.1.5. Towards Two New Abstract Models

As the previous chapter suggested, a new approach to parallel programming is required that takes away the burden of managing parallelism from application developers to allow them to concentrate on algorithmic issues. The discussion in this chapter leads to the

introduction of the NIP programming model, an implicitly parallel, functional plus objects model (Section 2.8). To support the NIP programming model, the NIP execution model was devised and it is described towards the end of this chapter (Section 2.9).

2.1.6. Layered Approach to the Parallel Computing Paradigm

To facilitate the discussion on programming and execution models and to emphasise the distinction between the two, this thesis proposes a layered approach to examining the parallel computing paradigm (Figure 2-1 and Figure 2-2). The design model has been included for completeness.

Figure 2-2 encapsulates the definitions that were presented earlier in this chapter for the programming, execution, and computational models. The layers represent the three major steps in the lifecycle of an application. (a) An algorithm is devised via a design model. (b) The developer implements that algorithm via the programming model, and (c) the execution model is used to host the resulting program. The computational model may be used for the mathematics-based evaluation of the behaviour and performance of the application. The result of the evaluation may be used in reviewing the algorithm.

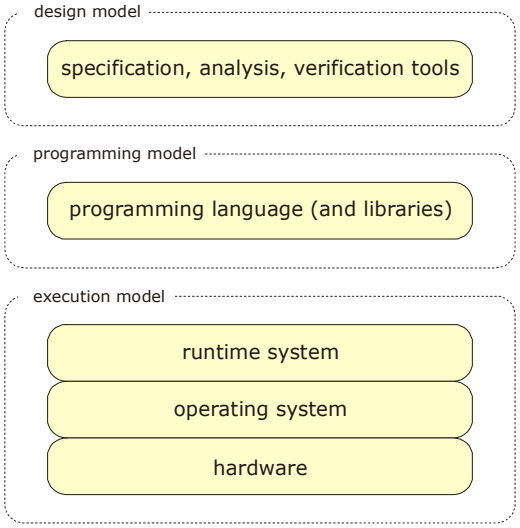


Figure 2-1: A layered approach to the parallel computing paradigm

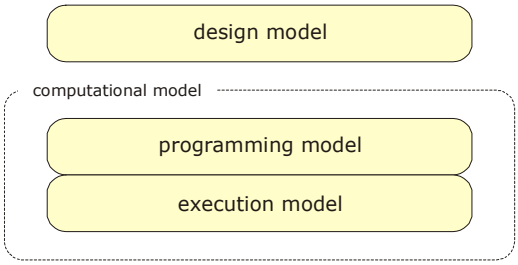


Figure 2-2: A condensed view of Figure 2-1 that also shows the computational model

Since this thesis does not consider any issues related to specification, analysis, verification, etc., the design model and its supporting tools are not examined. Instead, the following sections focus on issues related to the hardware, operating system, run-time system, and programming language from a parallel computing perspective.

2.2. Hardware

A plethora of hardware architectures have been built and described in the literature. The most referenced and used one is the von Neumann architecture, which is the architecture upon which all serial computers are built. There is a wide choice of parallel computer architectures from which system designers may choose. As Chapter 1 suggested, the MPP architecture (Figure 2-3) is emerging as the dominant one.

In an effort to study the available architectures for parallel systems, many researchers have tried to create taxonomies of computer architectures (Flynn 1972; Gajski and Peir 1985; Lewis and El-Rewini 1992; Tanenbaum 1999; Treleaven 1985). The taxonomy proposed by Flynn is the most general one and is presented in Table 2-1. Originally created in 1974, the taxonomy does not illustrate the variety of parallel computer architectures that have emerged since then because it is too coarse.

Instruction stream	Data streams	Name	Examples
1	1	SISD (Single Instruction Single Data)	Von Neumann architecture
1	Many	SIMD (Single Instruction Multiple Data)	Vector computers, array processors
Many	1	MISD (Multiple Instructions Single Data)	
Many	Many	MIMD (Multiple Instructions Multiple Data)	Multiprocessors, Clusters of Workstations

Table 2-1: Taxonomy proposed by Flynn (Flynn 1972)

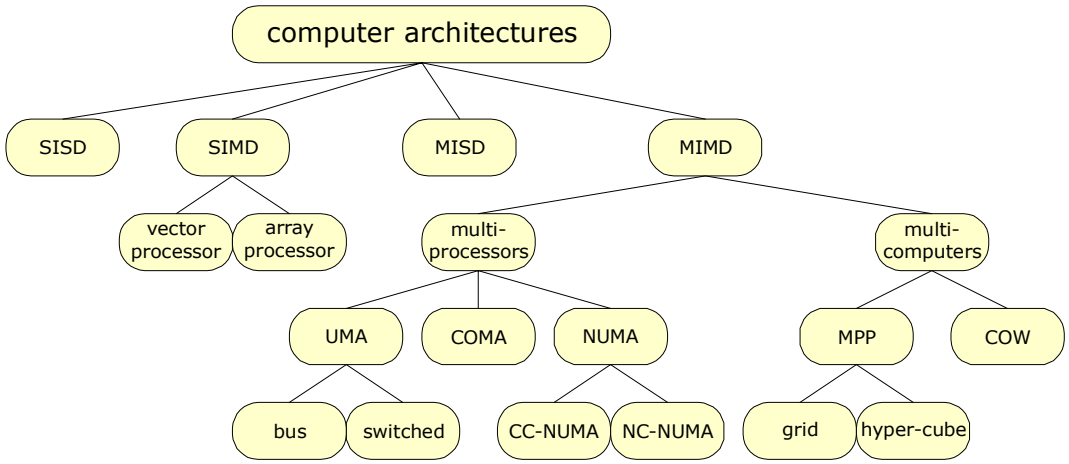


Figure 2-3: Taxonomy proposed by Tanenbaum (Tanenbaum 1999)

UMA	Uniform Memory Access
COMA	Cache-Only Memory Access
NUMA	Non-Uniform Memory Access
CC-NUMA	Coherent Cache NUMA
NC-NUMA	Non-Coherent NUMA
MPP	Massively Parallel Processing
COW	Clusters Of Workstations

Table 2-2: Legend of acronyms presented in Figure 2-3

In (Tanenbaum 1999), Tanenbaum expands the taxonomy proposed by Flynn by incorporating all the diverse parallel computer architectures available today (Figure 2-3). Tanenbaum refers to all the multi-computer architectures as message passing architectures and to all the multi-processor architectures as shared memory architectures. However, it should not be assumed that there is an association between particular architectures and programming or execution models, although there have been architectures designed and built to support a particular programming model, like the Manchester Dataflow Machine (Gurd et al. 1985) for dataflow programs. A parallel architecture may favour a particular programming model due to its architectural characteristics but often others are also supported. For instance, the shared-memory programming model can be used on both multi-processors and multi-computers with the appropriate software support. The NIP programming model (Section 2.8) is applicable to both multi-computer and multi-processor parallel architectures. For a detailed discussion of the parallel architectures the reader is referred to (Culler and Singh 1999).

2.3. Operating Systems

The AOSP, the TOPS-10 on DEC PDP-10, and OS/VS2 on the IBM System/370, were amongst the first operating systems to support parallel architectures. In the '70s and '80s, the commercially successful minicomputers and mainframes utilised proprietary operating systems with better support for parallelism, like VMS on DEC VAX minicomputers, UNICOS on CRAY supercomputers, MUNIX on PDP-11 (Almasi and Gottlieb 1994). Nowadays, different flavours of UNIX, like Solaris by Sun, Digital UNIX by Compaq, IRIX by SGI, dominate the parallel architectures. Solaris, Windows NT, and Linux (a freely available flavour of UNIX) emerge as the leading operating systems for COW architectures.

In the past, there was a case for creating taxonomies for the different operating systems due to the diversity in the ways parallelism was supported (*e.g.*, multiprocessing or

multithreading, synchronisation and communication primitives, etc.). Today, all the modern operating systems offer essentially the same features and, therefore, a classification based on the support provided for parallelism would be of little interest. The Application Programming Interface (API), the implementation details, or even the efficiency of the available features may differ between operating systems but the support for parallel applications is in essence the same. For example, all modern operating systems are internally parallel; they support multiple processors; they offer the means for multithreading and synchronisation; and they provide basic communication primitives. However, this does not imply that the area of operating system support for parallelism is closed to further developments, as the following areas of research suggest.

As multi-computers become more popular, the operating systems face the challenge for providing an environment for resilient, fault-tolerant, and reliable parallel computing, just like the expensive mainframes (Culler and Singh 1999).

Providing *"the illusion […] that a collection of computing elements is a single computing resource"* (Pfister 1998), or a 'Single System Image', at the operating system level is another challenge. Examples of commercial and research operating systems attempting to provide a Single System Image include Locus, QNX, and Amoeba, cited in (Pfister 1998).

Additionally, the new run-time techniques for the execution of parallel applications that emerge from research efforts could greatly benefit from operating system support. Examples of such techniques are lazy task creation (Mohr et al. 1991) which is examined in Chapter 3, software-based distributed shared memory (Li 1986) which is discussed in Chapter 4, and dynamic load balancing (Eager et al. 1986) which is considered in Chapter 5.

2.4. Run-time System

A run-time system is the environment with which an application interacts during its execution. It is the responsibility of the run-time system to manage the resources that the operating system makes available for the execution of an application (*e.g.*, stack and heap management, task creation, task scheduling, secondary storage management). Even though modern, general-purpose operating systems incorporate functionality that could make the use of specialised run-time systems obsolete, there are still reasons for using them.

- **Portability:** A run-time system can hide the API differences between operating systems. API calls provided by the run-time system are translated to the

appropriate operating system calls. Cross-platform development frameworks like ACE (Schmidt 1995) provide such functionality. Virtual machines, like the one used for the Java programming language (Newman 1996), and interpreters, like the one used for tcl/tk (Ousterhout 1994), also allow portability of applications by hiding the underlying operating system from applications.

- **Efficiency:** Due to their general-purpose nature, many operating systems are not able to provide the most efficient implementation of a particular service. In such cases, run-time systems can be of great use. For example, database management systems very often take over the management of the secondary storage from the operating system because their fine-tuned implementation can achieve better performance.
- **Flexibility:** With run-time systems it is possible to alter the execution environment of applications by incorporating new or removing obsolete features, or by changing the implementation of existing features without requiring changes to the underlying operating system.
- **Functionality:** There are cases where the functionality required by an application is not provided by the operating system. Advanced techniques for the execution of applications may include lazy task creation (Chapter 3), software-based distributed shared memory (Chapter 4), or dynamic load balancing (Chapter 5). A run-time system can incorporate such advanced techniques and make them available to applications.

Run-time systems exist in the form of software libraries that developers use during the implementation process, as part of the code that a compiler produces, or as virtual machines.

2.5. Programming Language

According to Hudak, programming languages are classified into imperative and declarative (Hudak 1989). Programs written in imperative programming languages consist of a series of commands that provide a description of a problem and how it is to be solved. In declarative programming languages, the application programmer is only concerned with the implementation of an algorithm, while the compiler and run-time system take care of other issues, like the management of parallelism, the way memory is accessed, hardware architecture independence, etc. The main difference between imperative and declarative

languages is state. *"Imperative languages are characterised as having an implicit state that is modified (i.e., side effected) by constructs (i.e., commands) in the source language"* (Hudak 1989). To deterministically control the state in imperative languages a sequencing of commands is required.

Hudak suggests that one of the advantages of declarative languages is parallelism. The lack of state reduces the need for command sequencing and makes extracting and exploiting parallelism a simpler task for the compiler and run-time system. However, software practises to date have shown that imperative languages are preferred for parallelism. To investigate the reasons behind the popularity of imperative languages, it is necessary that the properties and features of parallel programming languages be closely examined. Another taxonomy, more comprehensive than the one proposed by Hudak, is needed.

Almasi and Gottlieb point out in (Almasi and Gottlieb 1994) that the execution of a parallel application requires the provision of operations to define, start and stop, and coordinate parallel subtasks as well as operations to partition and distribute data. The method with which these operations are supported can be used to differentiate between programming languages. Indeed, Skillicorn and Talia propose an extensive classification of parallel programming languages based on whether the parallelism primitives are provided explicitly or implicitly (Skillicorn and Talia 1998).

Everything Explicit
Communication Explicit, Synchronisation Implicit
Mapping Implicit, Communication Implicit
Decomposition Explicit, Mapping Implicit
Parallelism Explicit, Decomposition Implicit
Nothing Explicit, Parallelism Implicit

Table 2-3: Summary of the examined parallel programming model properties (Skillicorn and Talia 1998)

Skillicorn and Talia start by identifying the important criteria a parallel programming model should satisfy: ease of programming, software methodology, architecture independence, ease of understanding, guaranteed performance, and cost measures. The identified criteria are used in outlining the parallelism related operations and the level of abstraction (i.e., implicit or explicit), upon which the classification of existing programming languages is based (Table 2-3 summarises the resulting six main categories of the taxonomy). Skillicorn and Talia conclude their work by observing a trend towards programming languages that support abstract models of parallel programming (Skillicorn

and Talia 1998), a view that is consistent with the discussion presented in the previous chapter.

In addition to the comprehensive classification by Skillicorn and Talia, it should also be interesting to examine the means by which programming languages support parallelism. Since it is not in the scope of this thesis to present a new taxonomy, only the characteristics of four different approaches to supporting parallelism in programming languages are considered.

2.5.1. Auto-parallelisation Compilers

The simplest and most cost-effective approach of introducing parallelism to existing applications is to feed serial, legacy code, to an auto-parallelisation compiler. The tasks of identifying (*e.g.*, task decomposition) and managing (*e.g.*, synchronisation) parallelism are taken over by the compiler.

The auto-parallelisation technique was popular in the early days of parallel computing and especially when large commercial and scientific applications written in FORTRAN had to be run in parallel. There is no need for programmers to receive extra training or learn another programming language. Programmers just continue to use their favourite serial programming language to build parallel applications.

However, it is difficult for auto-parallelisation compilers to achieve good performance results. The implementation of algorithms is based on a serial programming model and the von Neumann architecture. As Bacon et al. suggest (Bacon et al. 1994), application code can be successfully transformed and optimised by a compiler for sequential architectures but the involvement of the programmer is required for parallel architectures. A survey of issues related to compiler transformations for high-performance computing can be found in (Bacon et al. 1994).

2.5.2. Software Libraries

Software libraries can provide support for parallelism in established, widely used, serial programming languages. In essence, software libraries complement the run-time environment that is provided by the serial programming language compiler. A software library introduces the required primitives for parallelism related operations, like task management, communication, synchronisation, etc.

There are several ways a software library may provide support for parallelism. The Parallel Virtual Machine (PVM) (Sunderam 1990) and implementations of the Message Passing Interface (MPI) standard (Forum 1994) provide support for message passing programming (the model is discussed in Section 2.6.2). Implementations of the POSIX

standard (IEEE 1996) or operating system specific libraries provide threading and synchronisation support for shared memory programming (the model is discussed in Section 2.6.3). Software libraries like Linda (Ahuja et al. 1986) allow shared memory programming on distributed memory system by hiding communication.

There is a short learning curve for application programmers using such languages because they are only required to learn the Application Programming Interface (API) of a particular software library. However, application programmers are required to deal with issues related to parallelism, like task decomposition, data distribution, synchronisation, communication, etc. Programming languages like FORTRAN, C, and C++ that are complemented by software libraries, like PVM and MPI, belong to this class. The large academic, scientific, research, and commercial user-base of these programming languages have made the approach of software libraries for parallelism the most popular today.

In addition to the problem of managing parallelism, developers face the problem of portability. An application is only portable across parallel architectures if the software libraries it uses are available for the targeted architectures. Standards like MPI attempt to overcome the portability issue. Even then, though, there are cases where an application has to be refined when moved to a different parallel architecture due to efficiency variations in the performance of hardware components.

2.5.3. Language Extensions/Integration

As discussed in Chapter 1, parallel computing is gaining wider acceptance amongst academics, scientists, researchers, and industrialists. As a result, there have been significant research efforts in integrating parallelism support into programming languages. Extensions to the popular programming languages have been proposed, like //C++, Concurrent C, Concurrent SmallTalk, etc. Also, new programming languages appear providing constructs for parallel computing, with Java (Newman 1996) being the most popular example (Java provides threading and synchronisation support through appropriate classes that are considered to be part of the language rather than a library addition). Programmers are still required to manage parallelism themselves while they have to learn additional primitives in their favourite programming language, or they are obliged to switch to a new programming language.

The portability issue across parallel architectures is resolved because there is no dependence on extra software libraries, provided, of course, the programming language compiler and the standard libraries are available. Furthermore, features that ease program development are finding their way into the new parallel programming languages. For

example, Java has been gaining support because it combines good programming practices, namely object-orientation and component reuse, with a collection of primitives for parallelism, communications, graphical user interface, etc.

The transition from the serial programming languages to the languages with integrated support for parallelism requires resources and time. The language extensions/integration approach is only slowly gaining support by the scientific and research communities and especially by the industry, where FORTRAN, C, and C++, dominate.

2.5.4. Implicit Parallelism

Implicitly parallel programming languages allow application programmers to concentrate on the algorithmic issues of the development process rather than having to worry about the management of parallelism. There are no parallel programming primitives or constructs available to the developer, as the compiler and the run-time system control all the aspects of parallelism (the last class of programming models in the taxonomy by Skillicorn and Talia, Table 2-3, page 21). Parallel computations are automatically identified and exploited; synchronisation and communication are handled; and, data is partitioned and distributed where it is necessary. Additionally, source-code portability is not a major issue, once the programming language compiler has been ported to an architecture.

Implicitly parallel programming languages should not be associated with declarative languages (described in page 20) despite the fact that in declarative languages parallelism may be implicit. Although both categories of programming languages favour algorithmic focus over implementation details, implicitly parallel programming languages do not have to be stateless, a property of declarative languages. The functional plus objects United Functions and Objects (UFO) (Sargeant 1993; Sargeant and Kirkham 1994) and the visual, parallel object-flow VORLON (Webber 1998; Webber 1999) programming languages are examples of implicitly parallel languages which support state (i.e., objects in both cases).

The growing complexity of parallel applications and architectures and the management of the ever-increasing degree of parallelism place a great burden on parallel application developers. Tools for implicit parallelism could simplify the development process. Indeed, many parallel computing practitioners expect implicitly parallel programming languages to gain support in years to come and become the languages of choice (Culler and Singh 1999; Sterling et al. 1995). However, research on compiler and run-time technology has yet to demonstrate that performance is not sacrificed in the transition from explicitly parallel to implicitly parallel programming.

2.5.5. Transition Towards Implicit Programming Languages

Auto-parallelisation compilers and software libraries for serial programming languages have dominated parallel application development. FORTRAN has been the programming language of choice for decades despite its poor programming practices and the lack of support for parallelism at the language level. The required resources and time to switch to appropriate tools for parallel computing has been the main cause for the reluctance by practitioners to move away from serial programming languages. The position expressed by Perrott and Zarea-Aliabadi in (Perrott and Zarea-Aliabadi 1986) is representative of the dominating views during the last decade. They believe that a conventional, sequential language should be the basis for programming a parallel supercomputer.

> *"[…] it has been recognised that if computation on supercomputers is to have a major impact, languages that are generally similar to conventional languages in their approach to computing must be provided. The strength of this influence and the implicit dependence on conventional sequential language principles explain why most existing supercomputer languages are based on the ubiquitous language FORTRAN"* (Perrott and Zarea-Aliabadi 1986).

Still, Perrott and Zarea-Aliabadi also recognise the importance of relinquishing FORTRAN and moving to a better programming language. They propose a language for parallel programming on supercomputers, Actus (Perrott and Zarea-Aliabadi 1986), based on the sequential—but with better programming practices than FORTRAN—Pascal (Wirth 1971).

It is apparent from the works of experts in the field of parallel computing, that new approaches to programming parallel architectures are required. The widely used sequential programming language plus software library approach is not sufficient for developing the parallel applications of the future. Parallel computing experts seem to favour implicitly parallel programming languages that incorporate good software engineering practices (Almasi and Gottlieb 1994; Culler and Singh 1999; Skillicorn and Talia 1998; Sterling et al. 1995).

2.6. Common Programming Models

As mentioned at the beginning of this chapter, programming models abstract the architectural characteristics of computer systems for the benefit of the application developer. In an effort to study the available programming models, to better understand the differences between them, and to examine the way they are used in practice, Skillicron

and Talia identify in their work (Skillicorn and Talia 1998) the important properties they think a model should have (Table 2-4). They go on to create a classification of programming languages based on the level of abstraction in which parallelism primitives are supported (Table 2-3, page 21). However, Skillicorn and Talia consider the different programming models only through the characteristics of the examined programming languages. This thesis takes a more general approach and uses some of the properties identified by Skillicorn and Talia in considering four common approaches to programming.

A programming model should	be easy to program
	have a software development methodology
	be architecture independent
	be easy to understand
	have guaranteed performance
	provide accurate information about the cost of programs

Table 2-4: Properties a programming model should have (Skillicorn and Talia 1998)

There has been a plethora of models proposed in the literature and used in practise for parallel programming, including Active Objects (Lavender and Schmidt 1996), Active Messages (von Eicken et al. 1992), Actors (Agha and Hewitt 1987). It can be suggested, however, that the models this section considers are the most commonly used in practice: message passing, shared memory, functional.

Useful Terms

Before the discussion moves to the programming models, it is necessary to establish a common terminology:

- The unit of execution for applications is the task, which consists of a series of commands executing in a serial fashion. Parallelism is achieved through the concurrent execution of a number of tasks. According to the programming model or the run-time system used, a task may be known as a thread, process, lightweight process, etc.
- Processors execute tasks. A processor can only execute one task at a time. Parallelism is possible when more than one processor is available for the execution of a single application that is divided into multiple tasks.
- The memory stores the data required during the execution of tasks. Memory may not be directly accessible by a task running on a processor.
- The unit of information exchange between tasks is the message. Tasks communicate with each other by exchanging messages. Depending on the

architecture of the underlying parallel system, sending a message from one task to another may result in transmission of data over an interconnection network or it may just cause memory accesses.

- Tasks coordinate their execution using synchronisation primitives. There are two cases in which tasks may need to coordinate their execution: (a) when the ordering of execution is important (*e.g.*, barrier constructs), and (b) when the access to a shared resource is critical (*e.g.*, mutual exclusion constructs).
- Load balancing relates to the distribution of work across a computer system with the aim that all processors are utilised in the best way possible.
- Tasks may have different granularities. A task with a short execution time is considered to be of fine granularity, while a larger execution time suggests a task of coarse granularity.

2.6.1. Serial

It may be of surprise that the first programming model to be examined amongst parallel models is the serial. However and as discussed in Section 2.5.1 (page 21), the serial programming model was the basis for parallel application development during the early years of parallel computing, when auto-parallelisation compilers on sequential code were extensively used. The knowledge and experience acquired on the use of the model for developing applications was the important factor for the reluctance to move to models that are more suitable for parallel computing. Additionally, modern programming models, like the functional programming model (Section 2.6.4) and the NIP programming model (Section 2.8), bear some similarities in the way the targeted computer architecture is conceptualised.

The model is very simple in that it only assumes one processor and directly accessible memory (Figure 2-4). The key characteristics of the model are presented in Table 2-5.

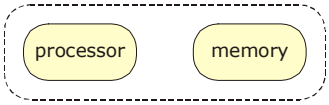


Figure 2-4: The conceptualisation of a computer system by the serial programming model

- The processor has direct access to the memory
- There is only one task executing on the processor
- All the memory is available to the task
- When the execution of the task starts, it is never interrupted

Table 2-5: Properties of the serial programming model

The serial model is easier to program and understand when compared to the message passing (Section 2.6.2) and shared memory (Section 2.6.3) models, as there is only one sequence of instructions. There are no communication or synchronisation considerations

and the software development methodology is well studied, understood, and applied throughout the years.

2.6.2. Message Passing

Unlike the serial model, message passing introduces the notion of parallelism as part of the model. The model assumes a number of processors with their own private memory (Figure 2-5). There can be many tasks executing at a time on the available processors but only one at a time on a particular processor. A task running on a processor cannot directly access the memory of another processor. Instead, tasks communicate and exchange data by conveying messages to each other. The interconnection network of Figure 2-5 is the transportation medium for the messages.

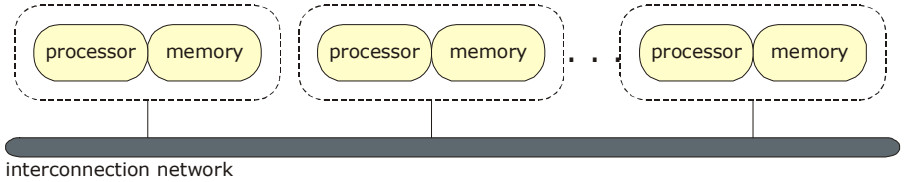


Figure 2-5: The conceptualisation of a computer system by the message-passing programming model

The model does not make any attempt to introduce any level of abstraction. Instead, application programmers are burdened with all the parallelism related issues. They have to contemplate the following problems and provide solutions:

- Decomposition of the application to parallel tasks. The granularity of the identified parallel tasks may also be important as it may directly affect the performance of the application.
- Partitioning and distribution of the data across the memories so that the parallel tasks can access it.
- Management of the communication and synchronisation between the parallel tasks (data exchange and execution ordering, respectively).
- Consideration for the possible communication latency and bandwidth variations at different parts of the interconnection network.

Properties Supported by the Model

After regarding all the above, it is safe to suggest that the message-passing programming model is lacking the first of the properties suggested by Skillicorn and Talia (Table 2-4, page 24): it is not easy to program. As the programs increase in size and complexity, the developers are faced with a great number of parallelism related issues that they need to resolve. Still, even though the model is difficult to program, it has been popular because it

is easy to understand and it can be easily supported by serial, conventional, programming languages.

Portability and performance are major concerns for programmers. In most cases, the message-passing programming model reflects the architectural characteristics of the underlying hardware. The exposure of the architectural details of the parallel system results in non-portable applications because system-specific features are used. Even if the issue of portability is resolved, though, there is no performance guarantee when an application is moved across parallel platforms. For example, an application developed to heavily depend on the low latency performance of a certain interconnection network will not perform well when moved to an architecture that favours bandwidth or processor speed. The consequence is that the application has to be rewritten or retuned for particular architectures, forcing human resources and time to be consumed.

In the last decade, as the complexity of applications increased and the portability issue became more of a concern, efforts to provide software tools with a standard interface have emerged.

Supporting Tools

The most widely used tools, adding support to conventional languages for the message-passing programming model, are PVM (Sunderam 1990) and the implementations of the MPI standard (Forum 1994), such as LAM, MPICH, and others. PVM (Parallel Virtual Machine) was the first environment to provide a portable solution to message-passing programming. However, the implementations of the MPI (Message Passing Interface) standard have been gaining momentum since the introduction of the standard back in 1994. Both PVM and MPI come as software libraries for conventional programming languages providing facilities to spawn new tasks on remote processors, to send and receive messages, to synchronise tasks, etc.

Due to the importance of the interconnection network in message-passing programming, vendors of parallel systems have been providing custom implementations of PVM or MPI libraries that are optimised for their architectures. As discussed earlier, applications developed to take advantage of the customised libraries are not portable, especially when performance is concerned, and they must be reprogrammed when moved to other architectures.

2.6.3. Shared Memory

The shared-memory programming model is more abstract to some extent than message-passing. The interconnection network is abandoned in favour of directly

accessible memory. All the processors have direct access to the memory of the computer system (Figure 2-6). There can be many tasks executing at a time on the available processors but only one at a time on a particular processor. Tasks communicate and exchange data or messages via memory operations.

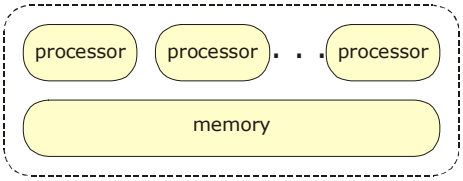


Figure 2-6: The conceptualisation of a computer system by the shared-memory programming model

Although some of the development burdens that were associated with the message-passing programming model have disappeared, application developers are still responsible with the explicit management of parallelism.

- Programmers have to identify and define the parallel tasks of the application (explicit decomposition).
- They have to deal with synchronisation between tasks (order of execution and mutual exclusion on memory access).

Unlike the message-passing model, the programmers are not required to partition and distribute data because there is only a single memory that all tasks can access. Communication latency and bandwidth are not a concern as they are in the message-passing programming model.

Properties Supported by the Model

As the programmer is not required to deal with network communication, data partition, and distribution issues, the shared-memory model can be regarded as being easier to program. However, the single memory of the model introduces synchronisation issues that need to be addressed. Understanding the synchronisation issues and presenting an efficient solution is a big challenge for shared memory application programmers.

The shared-memory programming model requires that the application developers deal with primitive, mainly architecture specific, operations like task creation, task synchronisation, load balancing, etc. As a result, the portability of applications is problematic. Still, even if the portability problem is resolved, the efficient execution of applications cannot be guaranteed across architectures. During the development process, the programmers may have to make decisions about the granularity and the number of the parallel tasks as well as the synchronisation and load balancing policies. The decisions may not be valid for all the targeted systems. For example, parallel systems may support

unequal number of processors, or the efficiency of the synchronisation primitives may differ. The application will have to be rewritten or retuned resulting in additional resources and time.

During the last decade, standards emerged and tools were developed providing better support for shared-memory programming.

Supporting Tools

The growing popularity of the model has resulted in a variety of tools being available today for shared-memory programming. There have also been efforts to create standards for the development of portable applications across different platforms, like the subset of POSIX (IEEE 1996) dealing with the interface for shared memory programming. Additionally, techniques have been proposed to get around the performance issue when moving to different architectures, as discussed earlier. Lazy task creation (Mohr et al. 1991), which is extensively studied in Chapter 3, is an example of such a technique.

The shared memory abstraction can be implemented in different levels of the computer architecture (Chapter 4 offers a more detailed discussion on shared memory architectures and tools):

- **At the hardware level**. (a) The memory is physically shared across the processors of the system. The architecture is widely known as Symmetric Multi-Processing (SMP). (b) The memory may be distributed across many processors in the parallel system (*e.g.*, MPP systems). Additional hardware provides the shared memory abstraction (*e.g.*, SGI Origin series). Finally, (c) the hardware may support the shared memory abstraction over a Cluster of Workstations. The Scalable Coherent Interface (SCI) (James 1994) and SHRIMP[2] (Billas et al. 1998; Iftode et al. 1999) are examples of the last approach.
- **At the operating system level**. The operating system provides the illusion to applications that there exists physically shared memory, although the memory may be distributed across many processors (*e.g.*, MPP systems) or even across many parallel systems (*e.g.*, Clusters of Workstations). Amoeba (Tanenbaum 1995; Tanenbaum et al. 1990) is an example of such an operating system.
- **At the programming library level**. Software libraries may provide the abstraction of shared memory on MPP or Clusters of Workstations architectures. Examples include Treadmarks (Keleher et al. 1994) and NIPDSM (Chapter 4).

[2] SHRIMP is actually a hardware/software hybrid approach to the shared memory abstraction over clusters of workstations.

Parallel computing practitioners have been disagreeing on which of message passing and shared memory is the best model for parallel programming. The architectures best suited for message passing-programming model (MPP architectures, Clusters of Workstations) provide scalability but may have limited performance due to the high communication costs introduced by the interconnection networks. In contrast, the architectures that favour the shared-memory programming model are not scalable due to the complexities related with the design and implementation of bus architectures.

A Hybrid Model

There have been research efforts to combine the message-passing and shared-memory programming models together (Figure 2-7). The architecture, which is gaining momentum in COW architectures, like the Avalon (Warren et al. 1997; Warren et al. 1998), combines the scalability of message passing architectures and the performance of shared memory architectures. However, the resulting, hybrid, programming model is even more complicated to program as it mixes the difficulties of both message-passing and shared-memory models.

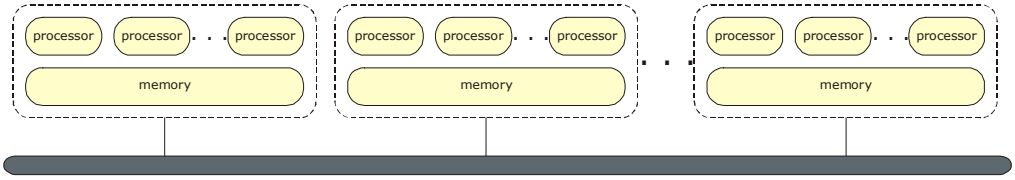


Figure 2-7: The conceptualisation of a shared-memory/message-passing computer system

2.6.4. Functional

The functional programming model completely abstracts away from any computer system architectural details. Application developers are not burdened with the management of parallelism. The computer system suggested by the functional programming model does not offer memory and only incorporates one processor. Of course, it is not implied that resulting applications will be executed on systems without memory and only with one processor. It is merely suggested that the programmer does not have to deal with memory state, task decomposition, synchronisation, etc. Nevertheless, the programming methodology is such that it allows the supporting tools to extract maximum parallelism from an application.

The functional programming model regards applications as expressions rather than a series of commands. There is no notion of state. The functional model of programming supports features like higher order functions, pattern matching, and abstract data types, which make it a very easy and effective model for developing applications. The in-depth examination of the issues related to the functional programming model is out of the scope

of this thesis. Instead, the reader is referred to (Bird and Wadler 1988; Field and Harrison 1988; Hudak 1989).

Properties Supported by the Model

Due to its abstract nature, the functional programming model is very easy to program and very easy to understand. It provides a straightforward way of implementing algorithms. Also, the resulting applications are highly portable as no architecture specific primitives are used. The implementation of an algorithm cannot be based on the performance characteristics of particular hardware components (*e.g.*, latency or bandwidth of an interconnection network).

Many have argued that the lack of state is a disadvantage for the functional programming model. However, Hudak suggests that functional programming languages do support state but in an explicit fashion rather than implicitly as it is the case with imperative languages (Hudak 1989). There have been efforts to incorporate state in functional programming languages (*e.g.*, ML). Such languages, although they support state, they are still considered to support the functional programming model.

Implicit parallelism is another important characteristic the functional programming model has to offer. Parallel computations can be easily identified, extracted, and exploited from applications developed with a functional programming language (Skillicorn and Talia 1998).

Supporting Tools

The functional programming model greatly depends on supporting tools (i.e., programming language compiler, run-time system). As programmers have no input on issues related to the management of parallelism (*e.g.*, task decomposition, data distribution, load balancing, synchronisation) the responsibility falls to the supporting tools. Examples of functional programming languages include Haskell (Hudak and Fasel 1992) and Miranda (Turner 1985).

2.7. Common Execution Models

As discussed in Section 2.1.2 (page 15), an execution model defines the run-time environment that will host the execution of applications. The characteristics of the model may reflect the underlying hardware details or the model may be abstract and not associated with any particular architecture. An execution model may also be seen as an

abstract machine that programming models target. It is not unusual for a particular execution model to abstract the semantics of another execution model.

2.7.1. Message Passing and Shared Memory

The message-passing and shared-memory execution models are the realisations of the abstract machine architectures assumed by the programming models with the same names that are presented in Sections 2.6.2 (page 26) and 2.6.3 (page 27). The non-abstract nature of the message-passing and shared-memory programming models allow for the execution primitives offered by the two execution models to become visible to application developers.

It has to be noted, however, that a message-passing or shared-memory execution model does not implicitly imply the existence of a message-passing or shared-memory parallel system respectively (as categorised by Tanenbaum in Figure 2-3 on page 18). For example, software based distributed shared memory (Li 1986) offers a shared memory execution model for shared memory programming over message-passing hardware architectures (*e.g.*, Clusters of Workstations).

These two execution models have been the most widely used in parallel computing, mostly due to the popularity of the message-passing and shared-memory programming models.

2.7.2. Dataflow

The dataflow execution model is based on the evaluation of a data-driven graph. The graph consists of nodes representing computation that operates on data and arrows representing the movement of data. Data is delivered from one computational node to another. The computation associated with a node in the graph can only be evaluated when the data it is operating upon is available.

The structure of a dataflow graph eases the process of implicitly exploiting parallelism in an application. Usually, a dataflow execution node is considered to be just one instruction and, therefore, the resulting parallel tasks are of very fine granularity. Depending on the underlying architecture the run-time system may determine different strategies for the evaluation of a dataflow graph.

The underlying architecture may be another execution model (*e.g.*, message passing, shared memory) or a dataflow machine. There have been efforts to design and build machines based on the dataflow execution model, including the Manchester Dataflow Machine (Gurd et al. 1985). A survey of dataflow machines can be found in (Veen 1986).

2.7.3. Functional

Unlike the dataflow execution model, the functional execution models are control-driven: a computation is executed when its result is required. Examples of functional execution models include graph reduction and string reduction.

In graph reduction, a graph is constructed from the functional program. A node of the graph represents a function (computation) while the arrows represent the flow of control. The transformation of the graph in consecutive steps represents the execution of a functional program. The graph is reduced to a simpler graph at each step. The structure of the graph simplifies the extraction and exploitation of parallelism from a functional program (Skillicorn and Talia 1998).

In string reduction, the functional program is represented as one expression. A transformation process takes place during the execution of the application. Each step of the process, transforms the functional program to a simpler expression through the evaluation of sub-expressions.

There have been efforts to design and build parallel systems for the execution of functional programs, like ALICE (Darlington and Reeve 1981) and Flagship (Watson et al. 1988). However, most of the research work has concentrated on the efficient implementation of functional execution models on top of the traditional message passing and shared-memory execution models, such as the GUM implementation of the Haskell programming language (Trinder et al. 1996).

A detailed discussion of the issues related to execution models and their implementations can be found in (Jones 1987).

2.8. NIP Programming Model

The two most widely used programming models for parallel computing today, message passing and shared memory, require the involvement of application developers in the management of parallelism. As the complexity of applications increase, the management of parallelism becomes problematic and consumes much of the software engineering effort. Increased complexity also results in applications that are difficult to debug and profile.

As discussed in (Watson and Parastatidis 1999a), in the opinion of many software developers, the functional programming model has a number of advantages over conventional, imperative languages including their expressiveness, and their amenity to reasoning about semantics. Moreover, the functional programming model also assists in

the development of parallel applications. In particular, functional programs contain far fewer constraints on execution order than do their imperative counterparts. This is because all expressions in a functional program are without side effects, or they are referentially transparent (Henson 1987). Therefore, the order of their execution cannot affect the result of the application, and this increases the scope for parallel execution.

Unfortunately, the very property that makes functional programs so well matched to parallel systems—referential transparency—makes the programming of certain important classes of computations unnatural, contorted and complex. In particular, many computations (or parts thereof) are naturally expressed through an object-oriented programming style in which objects encapsulate state, which may be updated through method calls. Method calls to objects may not be referentially transparent, as identical calls can return different values, and therefore these types of computations cannot be directly expressed in a functional program (Watson and Parastatidis 1999a).

Taking into consideration the above observations, a new approach to programming parallel applications is proposed. The NIP programming model was devised to act as the inspiration for the introduction of a new breed of parallel software development and run-time tools. The new model combines some of the characteristics of the previous models and introduces object-orientation as part of the programming model.

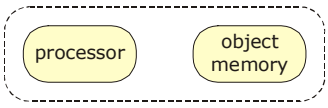


Figure 2-8: The conceptualisation of a computer system by the NIP programming model

The NIP programming model is abstract, as it hides all the architecture specific primitives from the developers. As Figure 2-8 suggests, the view of a single processor abstract machine with memory structured as objects is presented to the developers. The single processor view of the computer system does not imply that programs are executed serially. Like the functional programming model, parallelism is implicit. The model embraces the functional way of programming for the implementation of algorithms but it also allows state to be encapsulated in objects when developers feel it is natural to do so. As it was the case with the functional programming model, it would be the job of a compiler and run-time system to identify and exploit parallelism respectively from an application developed with the NIP programming model.

The adoption of object-orientation as an integral part of the NIP programming model brings to parallel computing an analysis, design, and development methodology that has

benefited serial programming but has yet to be widely accepted by the parallelism community.

Properties Supported by the Model

The model is easy to understand and use and provides a natural way to program applications. The applications developed with the NIP programming model are portable as there are no architecture specific primitives available. Furthermore, the implementation of an algorithm cannot depend on the performance characteristics of particular hardware components (*e.g.*, latency or bandwidth of an interconnection network, processor speed, etc.). Of course, the efficiency of the underlying hardware and/or the run-time will influence the performance of the application. It is merely suggested that application developers cannot take into consideration the efficiency related attributes of the hardware when implementing and refining algorithms.

Supporting Tools

As it is the case with abstract programming models, the NIP programming model greatly depends on supporting tools. However, there are no new programming languages yet designed and implemented to support the NIP programming model as proposed by this thesis. However, there do exist programming languages that satisfy the requirements of the model. UFO (Sargeant 1993) and VORLON (Webber 1998; Webber 1999; Webber 2000) are such programming languages.

The United Functions and Objects (UFO) programming language is an implicitly parallel programming language that supports objects. In UFO, as much as possible of the computation is expressed in a purely functional programming style. However, where it is natural to do so, objects that encapsulate state can be constructed and manipulated. The UFO programming language was a great inspiration in defining the properties of the NIP programming model.

VORLON is a visual, object-flow programming language that has been created to support the design and implementation phases of a parallel application and it features support for implicit parallelism and object-orientation.

The NIP run-time system (Watson and Parastatidis 1999a) is responsible for exploiting the parallel computations that are identified by the programming language compilers. The NIP run-time (Chapter 5) is an implementation of the NIP execution model semantics, which is the subject of the next section (Section 2.9).

2.9. NIP Execution Model

The NIP execution model defines an abstract machine for the parallel execution of applications. The abstract machine consists of a number of processors, shared memory structured as objects, and a special component that is responsible for the management of parallelism (Figure 2-9) and it is not associated with any particular underlying hardware architecture.

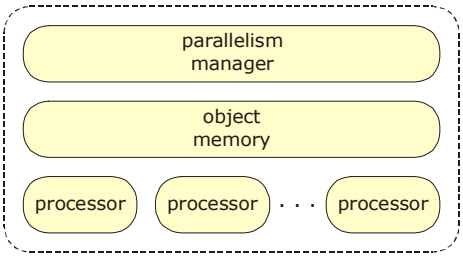


Figure 2-9: The major components of the abstract machine as suggested by the NIP execution model

Unlike other execution models, the NIP execution model takes over most of the operations that relate to the management of parallelism: load balancing, task creation, and task and memory access synchronisation. The identification of parallel tasks is left to the programming tools or the application programmer, depending on the programming methodology used. The NIP execution model was originally created to act as the target abstract machine for tools supporting the NIP programming model. However, the features introduced in the NIP execution model can also support other programming methodologies, even explicitly parallel in nature.

The NIP execution model attempts to combine the good features of the shared memory and the functional execution models while introducing implicit management of parallelism. The 'single, shared memory' view of the model makes the abstract machine an easier target for programming model support tools (*e.g.*, compilers). In addition, the NIP execution model defines that the memory is structured as objects in order to better support the object-oriented nature of the NIP programming model.

2.9.1. Model Requirements

As already mentioned, it is not the responsibility of the NIP abstract machine to identify parallelism. The abstract machine only exploits the parallelism already identified in applications. It is the responsibility of the parallel programming language support tools (*e.g.*, the compiler) or the programmer to identify the parallel tasks in an application. The NIP execution model defines the *tasklet* as the means for applications to represent potentially parallel tasks. The tasklet is semantically rich in that only one is enough to represent whole groups of potentially parallel tasks. Currently, tasklets support three

different patterns of parallel computation: function parallelism, loop parallelism, and recursion parallelism. The tasklet construct was originally introduced by Watson (Watson 1996) and it is further developed in this thesis (Chapter 3).

The NIP execution model also requires that the points in the application be identified where a tasklet is created and the result of the associated parallel computation(s) is used. The NIP execution model semantics do not define implicit task synchronisation and/or object availability. The execution of a task will wait for the computation(s) associated with a tasklet to complete via an explicitly introduced operation on that tasklet.

In addition to the identification of parallel tasks, the NIP execution model requires applications to indicate the type of the method calls on the objects. Applications should specify whether the state of objects is altered due to methods performed on them (read or write object access). The NIP abstract machine is not a virtual machine with a predetermined instruction set like the Java Virtual Machine (Newman 1996). Therefore, it would have been difficult and inefficient to expect implementations of the NIP execution model to identify the nature of the methods calls without assistance from applications. However, a programming language compiler could easily identify and provide the required information from the application source code.

The semantics of the NIP execution model specify that objects in the memory be implicitly destroyed when they are not required, or garbage collected. However, the current implementation of the memory system in the NIP run-time, the implementation of the NIP execution model, does not support garbage collection.

Finally, the NIP execution model does not make any assumptions about the correctness of applications when concurrency issues are concerned. The application should be created as if every potentially parallel task was actually going to be evaluated on a separate processor. The NIP execution model does not specify any means of avoiding problems with concurrent execution of applications (*e.g.*, races, deadlocks, etc.).

2.9.2. Run-time Environment

In addition to the requirements imposed by the NIP execution model semantics, the behaviour of the run-time environment during the execution of an application is also specified.

Not all the potentially parallel computations represented by the tasklets are converted to tasks. Parallel tasks are created depending on the availability of computational resources on the underlying computer system. The creation and distribution of parallel tasks throughout the computer system is the responsibility of the run-time environment.

The run-time environment must ensure that a parallel task can always call methods on an object, irrespective of the processor on which that task is executing.

Method calls on objects have sequential consistency semantics. Chapter 4 presents a detailed discussion on consistency semantics and memory access and gives the formal definition of sequential consistency.

Every task is allowed to have objects in a private memory. Objects placed in the private memory are not managed by the run-time environment and they do not adhere to the sequential consistency semantics nor they are accessible by other tasks.

2.9.3. Run-time Environment on Diverse Architectures

The NIP execution model is not associated to any specific architecture. Implementations of the run-time environment that adhere to the semantics of the NIP execution model must provide a 'single computer' view of the underlying hardware. The semantics of the NIP programming model specify that algorithms should not have to be re-implemented when moved between architectures. Hence, the run-time environment needs to address issues related to both distributed memory and shared memory architectures. As efficiency is of a concern, latency tolerance and overlapping of communication and computation are welcomed features. Furthermore, optimisations on shared memory architectures are also required.

The rest of this thesis presents the original research work undertaken on the advanced run-time techniques that are used for the implementation of the NIP execution model semantics. The techniques (Chapters 3, 4, 5) were implemented and evaluated (Chapter 6) as part of the NIP run-time system (Watson and Parastatidis 1999a; Watson and Parastatidis 1999b; Watson and Parastatidis 1999c).

The NIP lazy task creation technique together with analogous previous approaches to task creation are presented in Chapter 3. NIP lazy task creation features the tasklet construct, which was briefly mentioned above. In combination with the NIP load balancing service (Chapter 5), it is possible to achieve overlapping of computation and communication.

The NIP distributed shared memory system (Parastatidis and Watson 1999a; Parastatidis and Watson 1999b) is presented in Chapter 4. It provides an object-based view of the memory and introduces three new caching techniques. Object caching allows the NIP run-time system to tolerate latency in method calls on a distributed memory system.

CHAPTER 3

NIP LAZY TASK CREATION

The NIP execution model that was described in the previous chapter encouraged programming language support tools or application programmers to identify and expose the maximum logical parallelism possible in applications. In this chapter, the run-time techniques used to efficiently manage the resulting degree of parallelism are examined and the drawbacks of the existing systems are identified.

The NIP lazy task creation technique is proposed as a run-time technique to assist parallel applications better utilise the available computational resources. The NIP lazy task creation technique is built around a new construct, the tasklet, which is used by applications to expose the logical parallelism in applications.

Like similar constructs found in earlier lazy task creation techniques, a tasklet is a representation of a potentially parallel call. In contrast to previous approaches, however, the NIP tasklet construct is semantically rich. Just one NIP tasklet instance is enough to efficiently represent the parallelism available in entire iterative and recursive computations.

3.1. The Granularity Problem

As discussed in the previous chapter, many programming languages provide the means of expressing—in the form of language constructs or library calls—the logical parallelism in an application. Implicitly parallel programming languages, like UFO (Sargeant 1993), take away the burden of identifying parallelism from the application developers and move it to the programming language compiler and run-time.

In both cases, the degree of logical parallelism in the resulting application may be higher than is required to efficiently exploit the parallelism made available by the underlying parallel system. A higher degree of logical parallelism than is required often results in reduced performance as the overheads of managing parallel tasks (*e.g.*, task creation, task destruction, task switching, load balancing) may overwhelm the execution time.

The degree of parallelism in an application is inversely proportional to the granularity of the parallel tasks. Consequently, a lower degree of logical parallelism in an application results in tasks of coarser granularity and reduced run-time costs due to the management of parallel tasks. The underlying parallel system may then be more effectively exploited, provided, of course, that there is still enough parallelism to keep the system busy. However, lowering the degree of logical parallelism is can be difficult and not always a panacea, as will be explained subsequently.

3.1.1. The 'Expert' Programmer Solution

With the most popular parallel programming models (i.e., message passing and shared memory), it is the programmer who has to make the decisions about the granularity of the parallel tasks in an application. Quintessentially, a programmer decides on the degree of logical and usually the actual parallelism in an application.

Achieving the exact balance between the degree of logical parallelism and the granularity of parallel tasks can be a troublesome and time-consuming process. The process requires knowledge of the particular performance characteristics and primitive operation costs of the underlying hardware architecture and run-time system. The resulting application may only perform well on that particular combination of architecture and run-time system. When the application is moved to another platform, it will have to be rewritten or retuned, causing more resources and time to be consumed.

Additionally, a parallel algorithm may be more naturally expressed with a higher degree of parallelism. If the high degree of parallelism does not match the parallelism offered by the underlying architecture, the application developer would have to reconsider and redesign the algorithm before even having to deal with all the performance related issues mentioned above.

The granularity issue and its relation to the degree of logical parallelism in an application, as the above discussion reveals, are yet another incentive to utilise implicitly parallel compilers in the development process of parallel applications.

3.1.2. The 'Clever' Compiler Approach

Implicitly parallel compilers can relieve programmers from the difficult job of deciding the granularity of parallel tasks or the appropriate degree of logical parallelism in an application. Such compilers can use information about the underlying architecture when exploiting the parallelism in an application. Compile-time static analysis of the application source code can determine the appropriate granularity of the parallel tasks for a particular combination of hardware architecture and run-time.

However, there exist applications where even after analysis of the source code an efficient solution cannot be generated. Such applications are usually data dependant or irregular. Moreover, static analysis of the source code cannot utilise run-time information about the load of the underlying system in the case of a multiprocessing environment, where it is very likely that the ideal degree of parallelism that can be exploited is influenced by the availability of computational resources.

3.1.3. A Run-time Solution

In 1991, Mohr et al. proposed a run-time solution to the granularity problem (Mohr et al. 1991). They described a way in which applications with excessive parallelism could be efficiently executed on parallel architectures. Their approach did not require programmer assistance but instead it was a combination of compiler and automatic run-time techniques. The granularity of the parallel tasks could change dynamically at run-time depending on the availability of computational resources. The name of the run-time technique proposed by Mohr et al. was lazy task creation and is the subject of Section 3.2.

Other approaches similar to the lazy task creation technique have also been described in the literature. The lazy threads (Goldstein et al. 1996) technique is one of them and is described in Section 3.3. Section 3.4 briefly presents some other systems implementing similar approaches. Finally, Section 3.6 comprehensively examines a new technique originally devised by Watson (Watson 1996) and further developed in (Watson and

Parastatidis 1999a; Watson and Parastatidis 1999b; Watson and Parastatidis 1999c) and in this thesis, the NIP lazy task creation technique which provides support for iterative and recursive computations.

3.2. Lazy Task Creation

In 1991, Mohr et al. proposed a run-time technique for increasing the granularity of the identified parallel tasks in an application (Mohr et al. 1991). Their method utilised the Scheme functional programming language with the addition of a construct for explicit parallelism borrowed from Multilisp (Halstead 1985), called `future`. Their intention was to allow programmers to implement algorithms in a natural way, even if the resulting implementation had a very high degree of parallelism. The way the identified parallelism is exploited is left to the run-time system. As Mohr et al. notice, *"the programmer's task is to expose parallelism while the system's task is to limit parallelism"* (Mohr et al. 1991).

In their work (Mohr et al. 1991), Mohr et al. describe and compare three techniques for exploiting the parallelism in an application: eager task creation, load-based inlining, and lazy task creation.

- With eager task creation, every `future` call is converted to a parallel task at run-time, resulting in a great number of fine-grained tasks, when the programmer identifies the logical parallelism at a fine-grained level. Of course, as described in Section 3.1, the run-time costs due to management of parallelism overwhelm the execution time.
- In load-based inlining (Kranz et al. 1989), and during the execution of an application, a decision has to be made every time a `future` call is encountered. Depending on the load of the parallel system, a new parallel task is created to execute the computation associated with the `future` call or the computation is executed inline with the task that issued the `future` call. Drawbacks of the load-based inlining technique include the following (Kranz et al. 1989):
  - The involvement of the programmer is still required. The programmer has to code the load-based inlining behaviour and to define the load threshold upon which the decision to convert `future` calls to parallel tasks is based.
  - Deadlocks may arise in some applications due to use of load-based inlining.
  - The load-based inlining technique is not applicable in parallel iterative computations.

  - Most importantly, though, when computational resources become available, there may not be any parallel tasks to execute until another `future` call is encountered.

- The third approach, lazy task creation, resolves the issues associated with eager and load-based inlining techniques and it is the subject of the discussion bellow.

3.2.1. Concept

With lazy task creation as proposed by Mohr et al. in (Mohr et al. 1991), the source code looks exactly the same as in the case of eager task creation (for examples the reader is referred to (Mohr et al. 1991)). Programmers are not expected to write extra code or make decisions about the granularity of the parallel tasks. Using `future` calls, programmers are only required to identify all the computations that can be executed in parallel, resulting in applications with a high degree of logical parallelism. Lazy task creation attempts to reduce any run-time costs that may be incurred on a particular architecture due to the high degree of logical parallelism identified by the programmer.

During the execution of a Scheme program, when a `future` call of the form `(K (future X))` is encountered, where `K` is the *parent task* or the *continuation* of `X` and `X` is the computation identified as parallel, the run-time system starts executing `X` inline with the current task. Enough information is saved, however, so that the continuation `K` can be moved to a separate task if computational resources become available. Each processor maintains a list of available continuations that could be executed in parallel. When a processor becomes idle, it steals continuations from other processors, a method also known as task stealing. If continuation `K` of computation `X` has not been stolen when the execution of `X` completes, `K` will be removed from the list of available continuations and executed inline.

The lazy task creation approach differs from eager task creation because with the latter technique a new task would have been created immediately to evaluate computation `X` in parallel with `K` even if there were not any computational resources available.

In proposing the lazy task creation technique, Mohr et al. observed that the run-time cost of saving the information for the continuation `K` could be significantly lower than the cost of creating a new parallel task. In addition, they expected that in applications with a high degree of logical parallelism, the number of parallel tasks created to execute continuations would be significantly lower than the total number of tasks executed inline. Consequently, Mohr et al. anticipated that the run-time cost of saving information for all the continuations plus the cost of converting some of them to parallel tasks would be

significantly less than the run-time cost incurred if a new parallel task was created for every `future` call encountered. Indeed, the simulation-based performance results presented in (Mohr et al. 1991) confirmed the suitability of lazy task creation as a technique for reducing the run-time costs incurred due to excessive parallelism.

3.2.2. Implementation

Mohr et al. described two implementations of their lazy task creation technique in (Mohr et al. 1991) based on the Mul-T programming language (Kranz et al. 1989): one for the Encore Multimax shared memory multiprocessor and another one for the ALEWIFE distributed, globally shared memory multiprocessor (Agarwal et al. 1995).

Both implementations are based on the same principles. Every processor maintains a globally accessible lazy task queue with pointers to the available continuations in the stack frame of the executing task on that processor. When a `future` call is encountered during the execution of a task on a processor, a pointer to the `future`'s continuation is pushed onto the lazy task queue of that processor. If the execution of the `future` call completes and the continuation is still in the queue, the processor removes that continuation from its queue and executes it inline. Another processor becoming idle may also remove, or steal, that continuation from the lazy task queue.

Two levels of locking must take place to ensure the correctness of the operations on lazy task queues, as they are accessible by all the processors. First, a lock guarding a lazy task queue data structure must be acquired to prohibit two processors from stealing the same continuation. Then, another lock must be acquired to ensure that an idle processor does not steal a continuation while the processor that queued that continuation is trying to inline it. The ALEWIFE implementation of Mul-T takes advantage of specialised hardware to improve the performance of these locking operations.

When a processor steals a continuation from a lazy task queue, it removes a frame from the stack of the task executing on the processor owning the queue. The frame is used by the new task that is created to execute the stolen continuation. The traditional stack implementation that was used on the Encore Multimax resulted in run-time overheads due to the costly frame copying of continuation stealing operations. A more complicated implementation, which was based on a double-linked list of stack frames, was implemented for the ALEWIFE machine. The implementation utilised specialised hardware available on the ALEWIFE to improve the performance of continuation stealing operations.

3.2.3. Weaknesses

Logical Parallelism

Although recursive parallel applications may benefit from lazy task creation, as shown by the simulation-based performance evaluation presented in (Mohr et al. 1991), Mohr et al. believe that it is not possible to automatically increase the granularity of parallel, iterative, fine-grained computations. They stated that *"the sequentiality of iteration inherently limits parallelism"* (Mohr et al. 1991). Mohr et al. suggest that parallel, iterative computations are a case of algorithms where the convenient way in which they are expressed inherently limits parallel performance. This thesis attempts to demonstrate that the lazy task creation technique, as described by Mohr et al., fails to improve the execution of iterative, fine-grained, parallel computations mostly due to the way in which parallelism is identified for such computations using the `future` construct in the Scheme and Multilisp programming languages.

The pseudo-code of two versions of an iterative parallel map function, similar to the two Scheme versions presented in (Mohr et al. 1991), are given in Code 3-1 and Code 3-2. The iteration over the elements of a list is implemented as a recursive computation while the function `func` is considered to be fine-grained.

```
parallel-map func list
   if list not empty
      head-result is (future (func list.head))
      rest-result is (parallel-map func list.rest)
      return (concatenate head-result rest-result)
```

Code 3-1: First version of parallel map

In the first version of parallel map (Code 3-1), where the `future` call is the application of the function `func` to the head of the list, the task that starts the computation, the parent task, inlines that `future` call and makes its continuation available for parallel execution by adding it to the lazy task queue. If the function `func` is fine-grained, the continuation is removed almost immediately from the lazy task queue giving little opportunity to other processors in the system to steal it. As the lazy task queue is not built up, the load of the parallel system remains initially low. When another processor manages to steal a continuation, the parent task is not able to inline any more `future` calls and—depending on the implementation—will have to block or steal a continuation from another processor. The former approach means that whenever tasks are blocked computational resources are wasted. The latter approach results in many parallel tasks of fine granularity to be created.

```
Parallel-map func list
   if list not empty
      rest-result is (future (parallel-map func list.rest))
      head-result is (func list.head)
      return (concatenate head-result rest-result)
```

Code 3-2: Second version of parallel map

In the second version of parallel map (Code 3-2), the application of the function `func` to the rest of the list is identified as the `future` call. The parent task iterates through the whole list inlining all the `future` calls and making available a great number of continuations to other processors. The parallel tasks that may be created from those continuations to execute the application of the function `func` to the head of the list are of very fine granularity, as they do not include any `future` calls. The lack of `future` calls means that there are no possibilities for inlining. Only the granularity of the parent task may be increased.

These issues discussed above were recognised by Mohr et al. and they dismissed the suitability of lazy task creation for iterative computations. This thesis discusses an alternative approach, based on a scheme originally perceived by Watson (Watson 1996), which overcomes the limitations of the technique. The NIP lazy task creation technique, discussed in Section 3.6, provides an alternative way of identifying parallelism in parallel, fine-grained, iterative and recursive computations.

Memory and Stack

In the simulation-based performance evaluation of the Encore Multimax and ALEWIFE presented in (Mohr et al. 1991) the cost of the memory operations was not taken into consideration. With lazy task creation, whenever a `future` call is encountered, memory has to be allocated on the global heap to store the pointer to the stack frame of the continuation. As heap memory operations are significantly more expensive than stack memory operations, the allocation of the lazy task queue on the heap introduces extra run-time costs.

Additionally, the lazy task creation technique, as proposed by Mohr et al., relies on the manipulation of the parallel application stack. As such, the technique depends on specialised compilers. In the case of the ALEWIFE machine, the technique also utilises specialised hardware for the implementation of the stack structure in applications to improve performance. This dependence on specialised compilers and hardware makes the technique difficult to port on different platforms.

Probably the most important memory related drawback of the Mohr et al. lazy task creation technique is the copying of stack frames during a continuation stealing operation.

Apart from the run-time cost incurred, pointers to data structures on the heap become invalid on distributed memory architectures. Additionally, data structures that are allocated on the stack frame of the continuation are no longer available to the `future` call that has been inlined, resulting in poor memory access locality.

The NIP lazy task creation technique, discussed in Section 3.6, does not use the heap when identifying potentially parallel computations and it does not require specialised stack structures or copying of stack frames.

3.3. Lazy Threads

Goldstein and Culler proposed lazy threads as a run-time technique to increase the execution efficiency of parallel applications. The technique was described in (Goldstein et al. 1996) and later in more detail in Goldstein's doctorate thesis (Goldstein 1997). The main objective of lazy threads is to provide a way of executing potentially parallel calls as efficient as sequential calls when parallel execution is not required (i.e., computational resources are not available).

3.3.1. Concept

With lazy task creation, a `future` call causes information to be saved in order to make its continuation available for parallel execution. As discussed in Section 3.2, it is computationally less expensive to save the necessary information for a continuation than actually creating a new parallel task. Still, the run-time cost was significantly higher when compared to the cost incurred for a sequential call because of the queuing and synchronisation operations required. Goldstein and Culler attempted to design and implement an execution mechanism for parallel applications that further reduced the run-time costs due to potentially parallel calls (or, `future` calls in lazy task creation). Their run-time technique greatly depended on compiler support and a tree-like storage model for stacks, called a cactus stack.

With the lazy threads execution mechanism, a parallel call is executed as a parallel-ready sequential call. The function associated with the parallel call is executed just like a sequential call. Space on the stack of the calling thread, the parent, is allocated for the execution of the parallel-ready call, the child. If the execution of the child completes before computational resources become available, the parent will be resumed exactly in the same manner as when a sequential call finishes. Registers are used for the transfer of control and data between the parent and the child, making the execution of the

parallel-ready call as efficient as the execution of a sequential call. If the child suspends or yields the processor, it will have to be disconnected from its parent. A new thread will be created to execute the continuation of the parallel-ready call, just like continuation stealing in the lazy task creation. In this case, the stack frames of the parent and child are disconnected. The compiler produces code for every parallel-ready call to allow the child to be joined with its parent in case they were disconnected.

Goldstein and Culler observed that in many cases the parent was resumed as a separate thread only to make another parallel-ready call. They introduced thread seeds as an optimisation, to allow new threads to be created without the parent having to be resumed. A thread seed points to the next available parallel-ready call in the parent, following the one that is using the parent's stack. When an idle processor attempts to steal work from the suspended parent, it uses the thread seed to locate an available parallel-ready call and create a new thread to execute it. The thread seed used for the new thread is changed to point to the next available parallel-ready call. It must be pointed out that thread seeds can only be used with consecutive parallel-ready calls.

It is not always possible to represent a potentially parallel call as a parallel-ready sequential call. Sequential calls use the same stack frame as the parent and there may be situations where the data structures passed as arguments to a parallel-ready call are modified by the parent before the result of the parallel call is required. The disconnection of the child from its parent and the parallel execution of the two may lead to erroneous execution. In order to deal with such situations Goldstein and Culler introduced another representation of a potentially parallel call, the closure. A closure is a data structure that contains the arguments for the potentially parallel call. When a parallel-ready call is encountered, a closure is queued and control is returned directly to the parent. When the result of the potentially parallel call is required, the closure is dequeued and executed (or, inlined as in lazy task creation). Another processor, though, may steal the closure and execute the associated call in a new thread. The parent will have to wait until the new thread completes.

The lazy threads execution mechanism greatly depends on the generation strategy followed by the programming language compiler. The compiler decides on the appropriate representation (i.e., closure, thread seed, parallel-ready sequential) of the potentially parallel calls in an application. The compiler also generates extra code to allow for work stealing operations and thread synchronisation for every potentially parallel call.

A requirement for the lazy threads execution mechanism is that *"threads are scheduled independently and are not pre-emptive"* (Goldstein et al. 1996). The programming language compiler provides the code necessary for the scheduling of the created threads.

3.3.2. Implementation

In his thesis (Goldstein 1997), Goldstein describes the implementation of two programming compilers that provide support for lazy threads, the Split-C+threads and the Id90. Goldstein developed the Split-C+threads programming language based on the imperative Split-C (Culler et al. 1993a). Id90 is an implicitly parallel functional programming language.

The Split-C+threads compiler was a modified GCC compiler to support the lazy threads execution model. During the performance evaluation of the lazy task creation technique in a distributed memory parallel environment, the compiler did not generate any parallel-ready sequential calls or thread seeds for the potentially parallel calls. Such representations are only effective in a shared memory parallel environment. Instead, closures were used for the representation of potentially parallel calls, which are not as efficient as the other representations.

Finally, the applications compiled for the distributed memory parallel environment had to poll the network for messages from other processors. The polling operation was very expensive, especially for the Berkeley NOW parallel system (Culler et al. 1997) that was used. The slow network of the Berkeley NOW parallel system had a great impact on performance. Goldstein had to introduce a polling period (i.e., a certain number of polling operations were skipped) to improve performance. The execution on the Thinking Machine CM-5 multiprocessor (Hillis and Tucker 1993) did not suffer from the same problems.

3.3.3. Weaknesses

Threads

The integration of support for threads into the programming language compiler causes portability considerations, as Goldstein notes in his thesis (Goldstein 1997). The lazy threads execution technique is not easily portable across architectures. Furthermore, despite the resulting performance benefits due to the integration, additional limitations are imposed. As the threads are scheduled independently and there is no pre-emption, problems may arise. Goldstein and Culler note in (Goldstein et al. 1996) that because all the computational threads are required to complete before an application can finish, fairness is not an issue. Indeed, some applications, usually scientific in nature, only require

processing resources in order to perform a number of calculations. However, there exist applications where pre-emption is required, such as real-time applications. Most importantly, though, independent scheduling is not suitable for applications that perform operating system calls (*e.g.*, I/O, synchronisation, etc.). Operating system calls may block a thread but due to the lack of pre-emption and global scheduling, computational resources are not freed and therefore possibilities for parallelism are lost. Finally, due to independent scheduling, it is difficult to take advantage of the symmetric multiprocessing support many modern operating systems offer.

Granularity

The lazy threads execution mechanism succeeds in meeting the same goals as the lazy task creation technique described in Section 3.2 while introducing performance improvements. Applications with parallel, recursive computations and potentially parallel calls can be executed efficiently without incurring the overhead of excessive parallelism. However, parallel, iterative computations are still a great problem, as was the case with lazy task creation (Section 3.2.3). The granularity of parallel, iterative computations cannot be dynamically increased based on the availability of computational resources.

The independent iterations in an iterative computation are represented as parallel-ready sequential calls, threads seeds, or closures by the programming language compiler that supports the lazy threads execution mechanism. However, only the less efficient closure representation is suitable for distributed memory architectures. When a processor becomes available, and regardless of the representation of the potentially parallel call, the granularity of the new thread that is created is determined by the granularity of the stolen iteration. Fine-grained iterations will result in many fine-grained threads. The NIP lazy task creation technique (Section 3.6) provides a solution to the granularity problem for iterative computations.

3.4. Other Run-time Techniques

A number of other run-time techniques similar to lazy task creation and lazy threads have been proposed in the literature (Blumofe et al. 1995; Engler et al. 1993; Freeh et al. 1994; Vandervoorde and Roberts 1988; Wagner and Calder 1993). Two of them are briefly described in this section as they have some common characteristics with the NIP lazy task creation described in Section 3.6.

3.4.1. WorkCrews

In 1988, Vandervoorde and Roberts described in (Vandervoorde and Roberts 1988) WorkCrews as a technique for controlling parallelism in Modula-2+ (Rovner 1986). A number of workers (i.e., threads) are created during the initialisation of a Modula-2+ application. One worker is created for every processor in the parallel system and every worker maintains a queue of help requests. A help request consists of a procedure and the data structure that represents its arguments.

In WorkCrews, an active worker exposes parallelism by placing help requests in the help request queue. When a worker becomes idle, it tries to assist other workers by removing and executing a help request from their queues. When the result from a help request is required and if another worker has not already stolen that help request, the worker that queued the request executes it. The scheme is similar to the closures of the lazy threads execution mechanism (Section 3.3.1).

The main problem with WorkCrews is the potential for deadlock. Blocked workers do not attempt to execute work from other workers. As there is no provision for additional workers (only one per processor is available), an application may enter a deadlock state. Application developers must be aware of this erroneous execution behaviour and code their applications accordingly.

In (Mohr et al. 1991), Mohr et al. favour their lazy task creation technique over WorkCrews due to the required involvement of the application programmers in the latter technique. In WorkCrews, application programmers have to expose parallelism by queuing potentially parallel calls and their arguments and synchronise the execution of the workers with the results from any lazily created tasks. Instead, in lazy task creation the `future` construct is all that is required. The stylistic differences in the two techniques also result in implementation differences. The lazy task creation technique requires specialised compilers and the manipulation of the stack, which decrease portability, while the WorkCrews technique is applicable to conventional stack implementations and does not require compiler support.

3.4.2. LeapFrogging

LeapFrogging was described in (Wagner and Calder 1993) as an extension to the original lazy task creation technique (Mohr et al. 1991) that deals with imbalanced computational trees. Actually, it is a combination of the lazy task creation and WorkCrews techniques. The closure-like representation for potentially parallel calls is used in LeapFrogging, as in WorkCrews and lazy threads. A collection of threads is available to execute work that is

identified as potentially parallel. The number of available threads exceeds the number of processors in order to overcome the deadlock problem of WorkCrews. The LeapFrogging technique is implemented in C++ and it is compatible with a variety of thread libraries.

3.5. On Potentially Parallel Calls and their Representations

All the run-time techniques described above and others that have been proposed in the literature (Blumofe et al. 1995; Engler et al. 1993; Freeh et al. 1994; Vandervoorde and Roberts 1988; Wagner and Calder 1993) attempted to deal with the fine-grain and/or excessive parallelism in some applications, mostly recursive in nature. However, none of the proposed techniques has been able to handle the fine-grain parallelism in iterative computations.

3.5.1. The Problem with Iterative Computations

With the run-time techniques that have been proposed in the literature, parallelism is exposed through representations of potentially parallel calls. At run-time, such representations may be converted to parallel tasks when computational resources become available. Otherwise, the calls with which the representations are associated are executed like sequential calls. The granularity of the task that executes the potentially parallel call as a sequential call is increased and the extra costs of managing parallelism are avoided (i.e., task creation, scheduling, synchronisation, etc.).

For run-time techniques to work, the execution environment must be made aware of the available potentially parallel calls in an application. The approach employed in introducing a potentially parallel call to the execution environment depends on the technique used (*e.g.*, queuing of `future` constructs in lazy task creation by Mohr et al., advanced stack manipulation and parallel-ready sequential calls, thread seeds, and closures in lazy threads).

In all the techniques that have been proposed, each potentially parallel call must be represented separately. Although this approach may work with recursive computations, it introduces extra overheads with iterative, parallel computations. For example, with the lazy task creation technique iterations would be provisionally inlined as they are encountered. If computational resources become available, the remaining iterations would be stolen leaving the current task without work after the currently executing iteration completes. As a result, the active task would have to suspend and attempt to steal work from the task that just stole the rest of the iterations. This would continue until all the

iterations have been completed. It is clear that unnecessary overhead is introduced and tasks of fine granularity are created as a result. The parallel-ready sequential call representation in the lazy threads execution mechanism suffers from exactly the same problem (also refer to the discussion in Section 3.2.3).

In lazy threads, the thread seed representation of potentially parallel calls provides an improved solution for the execution of iterative, parallel computations. The granularity of the task that introduced the thread seeds for the iterations may be dynamically increased as potentially parallel calls are executed sequentially. The thread seed representation, though, is not appropriate for distributed-memory systems. Instead, the closure representation for potentially parallel calls must be used (the `future` structure of the leapfrogging technique and the help request of WorkCrews are semantically the same as the closures of lazy threads). The closure representation allows a task to continue its execution even when a potentially parallel call is encountered. The closure is queued so when computational resources become available, the closure is stolen and converted to a parallel task. Otherwise, the call associated with it is executed sequentially at a synchronisation point. However, thread-seeds and closures still result in fine-grained parallel tasks from iterations as only one iteration may be stolen at a time. Only the granularity of the parent task has the potential to be increased. Finally, all the possible parallel iterations have to be identified one-by-one by either a thread seed or a closure, which introduces additional run-time overhead.

The above discussion suggests that the reason the existing run-time techniques cannot efficiently handle iterative, fine-grained, parallel computations is the way in which individual iterations are made available for parallel execution. Only one at a time is introduced to the execution environment and only one at a time may be stolen and converted to a new parallel task.

3.5.2. A Solution

A new run-time technique, NIP lazy task creation, devised by Watson (Watson 1996) and further developed in (Watson and Parastatidis 1999a; Watson and Parastatidis 1999b; Watson and Parastatidis 1999c) and in this thesis, attempts to provide a solution to the efficiency problem with iterative, fine-grained, parallel computations. NIP lazy task creation is based around Watson's representation of iterative, parallel computations, the tasklet (Watson 1996). A tasklet exposes the parallelism available in iterative computations and allows new parallel tasks to be created lazily as computational resources become available. In contrast to previous approaches of potentially parallel calls representations

(*e.g.*, a `future` construct or a closure), just one tasklet may represent all the iterations of an entire iterative computation. The previous execution environments required that iterations were uniquely identified.

This thesis builds upon the ideas first introduced by Watson (Watson 1996) and extends the tasklet construct to represent whole recursive, tree-like computations and independent potentially parallel calls. Moreover, an object-oriented approach to the design and implementation of the tasklet construct in the NIP run-time system (Chapter 5) was preferred. Unlike the original tasklet, which was introduced by Watson (Watson 1996), tasklet constructs in the NIP run-time system are allocated on the stack frame of their parent tasks rather than the heap of the applications (Section 3.6.2), which improves efficiency.

The tasklet construct is examined in detail together with the NIP lazy task creation technique in the remaining of this Chapter.

3.6. NIP Lazy Task Creation

The NIP lazy task creation technique (Watson 1996; Watson and Parastatidis 1999a; Watson and Parastatidis 1999b; Watson and Parastatidis 1999c) was designed as part of the NIP run-time system, which is an implementation of the NIP execution model (Section 2.9). The main aim of the NIP run-time system is to provide an efficient execution environment for general-purpose, object-oriented parallel applications on distributed memory and/or shared memory multiprocessor architectures. Parallel applications are expected to expose as much logical parallelism as possible. The NIP run-time is responsible for efficiently exploiting the identified parallelism and keeping the parallel system busy. The algorithm for the distribution of work employed by the NIP run-time is based on the work stealing, or task stealing, technique. The implementation details of the load distribution algorithms that were adopted in the NIP run-time system are presented in Chapter 5.

This section discusses the details of the NIP lazy task creation technique. It provides a closer look at the tasklet construct and the way it is used to expose parallelism. The internal data structures required for the implementation of the NIP lazy task creation technique are also described.

3.6.1. The Tasklet

As it has already been mentioned, a tasklet is a new representation for potentially parallel calls (Section 3.6.4) that can efficiently expose the parallelism in entire iterative (Section 3.6.5) and recursive (Section 3.6.6) computations. A tasklet is a run-time construct that it is best described as an instance of a type whose prototype interface is shown in Code 3-3. Specialisations of the basic `Tasklet` type must provide implementations for the first four methods: `createTask()`, `executeTasket()`, `returnTask()`, and `waitOrInline()`.

```
type Tasklet
public:
   virtual bool createTask(Task&)
   virtual void executeTask(Task&)
   virtual void returnTask(Task&)
   virtual void waitOrInline()
   void activate()
   void deactivate()
protected:
   bool waitForStolenTasks()
```

Code 3-3: Public interface of a tasklet in pseudo-code

The tasklet representation was designed to support a task creation mechanism based on work stealing (or, task stealing), similar to the lazy task creation techniques described in the previous sections. When a processor of a parallel system runs out of computational work, it attempts to 'steal' work from other processors. Processors find additional work through the tasklet availability queue, which is maintained by the NIP run-time system. Tasklets expose logical parallelism to the execution environment by adding themselves to the tasklet availability queue (Section 3.6.2).

The Interface

Idle processors create a `Task` object to store the necessary run-time information for the execution of a new parallel task. A `Tasklet` object differs from a `Task` object in that the former is a representation of one or more potentially parallel calls while the latter maintains the required information for the execution of a new parallel task (*e.g.*, function call, arguments). Once the `Task` object has been created, idle processors iterate through the tasklet availability queue until they can find a tasklet from which they can steal computational work. After a tasklet instance has been chosen, the `createTask()` method is called for that instance. The method checks to see if a new parallel task can be created from the selected tasklet in which case the already instantiated `Task` object is updated. Otherwise, another tasklet will have to be chosen.

After the `Task` object has been instantiated, updated, and moved to the idle processor, the `executeTask()` method of the tasklet that was used is called. The method uses information from the `Task` object to evaluate the stolen computation in parallel with the computation that created its tasklet.

Upon completion of the parallel task, the `Task` object is returned to the tasklet from which the computation was stolen. If necessary, the `Task` object is updated to contain the return value of the evaluated computation. The `returnTask()` method of the original tasklet is then called and the tasklet is notified that the stolen computation has finished.

When the task that created the tasklet reaches the point where the result of the associated computation is required, it calls the `waitOrInline()` method. If an idle processor stole the computation, the execution of the task will have to block until the stolen computation completes. Otherwise, the computation is executed inline, like a sequential call.

An instance of the `Tasklet` type is added to the tasklet availability queue when the `activate()` method is called and it is removed from the queue by the `deactivate()` method. The automatic addition (removal) to (from) the queue could have been associated with the construction (destruction) of tasklets but the chosen approach allows for more flexibility in the design of `Tasklet` specialisations. Since there is an associated run-time cost with the initialisation of a tasklet instance, a `Tasklet` specialisation could allow its instances to be reused in exposing different potentially parallel computations without having to incur the construction related run-time costs.

Finally, the `waitForStolenTasks()` method automatically removes the tasklet instance from the tasklet availability queue. Additionally, if there are parallel tasks that were created from the tasklet and are still running, the method will suspend the execution of the task from which it was called. The execution of the suspended task will resume when all of the parallel tasks have returned. The method can only be called from within a specialisation of the `waitOrInline()` method. In some cases, the `waitOrInline()` method may only include a call to `waitForStolenTasks()`.

3.6.2. Tasklet Internals and the Tasklet Availability Queue

The objective of the lazy task creation technique, as discussed in previous sections, was to allow parallel applications to expose the maximum logical parallelism possible to the execution environment, without incurring the run-time costs due to excessive parallelism. The insight was that only a small number of the total identified potentially parallel calls were actually going to be converted to parallel tasks. It was expected that most of the calls

were going to be executed inline, like sequential calls. The previously published works validated the lazy task creation ideas (Goldstein 1997; Mohr et al. 1991).

As the NIP lazy task creation technique is based on the same concept as the original lazy task creation technique, the execution efficiency of the parallel applications greatly depends on the costs of creating tasklets and inlining their associated computations. Therefore, the tasklet creation and inlining costs should be kept as low as possible. Only then can parallel applications expose the maximum logical parallelism possible to the NIP execution environment by creating as many tasklets as possible, without incurring significant run-time costs. It is expected that for a parallel application with a high degree of logical parallelism, the run-time cost of instantiating tasklets, creating a small number of parallel tasks from those tasklets, and inlining the rest, would be significantly smaller than eagerly creating all the possible parallel tasks (the performance evaluation of the NIP lazy task creation technique is presented in Chapter 6).

Tasklet Instantiation

The tasklet representation for potentially parallel calls has been designed with efficiency in mind. Instances of the `Tasklet` type are allocated on the stack of the task that creates them rather than on the heap, which is the case with most of the previous potentially parallel call representations. As a result, the memory allocation costs associated with the heap are avoided.

Every new tasklet is added at the end of the tasklet availability queue, which is a double linked list of tasklets. No extra space is required to be allocated, as the linking pointers are part of the state of each tasklet (Code 3-4). However, a locking operation is necessary to guarantee the integrity of the tasklet availability queue data structure. The cost of the locking operation is considered to be part of the tasklet instantiation costs even though it takes place when the `activate()` method is called. A tasklet exposes potentially parallel calls only when it is part of the tasklet availability queue and it is only added to the queue when the `activate()` method is called.

```
type Tasklet
private:
   Tasklet* next
   Tasklet* previous
   Mutex*   lock
```

Code 3-4: Tasklet private data members

Every tasklet is associated with a lock during its instantiation process. A new lock does not have to be created for every new tasklet. A pool of already instantiated locks may be used. This—private to the tasklet—lock is used during the inlining and stealing operations and to guarantee the integrity of the internal state of the tasklet instance with which it is associated.

Besides the three essential private data members (as shown in Code 3-4), and more often than not, specialised tasklets will have private data members to store additional information. The initialisation cost of such tasklets also depends on the cost of creating the additional data members. However and unlike previous approaches, a tasklet can be constructed in such a way that once instantiated it can be reused within the same scope of a task, reducing the effect of the instantiation costs. Code 3-5, for example, shows how a tasklet with a member function `expose()` might be reused to expose to the NIP run-time two different potentially parallel calls, *f* and *g*.

```
ExposeFunctionTasklet tasklet
tasklet.expose(f)
tasklet.activate()
...
tasklet.waitOrInline()
...
tasklet.expose(g)
tasklet.activate()
...
tasklet.waitOrInline()
```

Code 3-5: A tasklet can be reused within the same scope

Two-level Locking

When a processor runs out of work, it looks into the tasklet availability queue in order to locate a tasklet from which it can steal work. A lock must be acquired to guarantee the integrity of the queue data structure while the search for a tasklet is in progress. Once a tasklet is chosen, the private lock of that tasklet is also acquired and the `createTask()` method is called. The acquisition of the private to the tasklet lock guarantees that there is not going to be an attempt to inline the associated computation while the stealing operation is in progress.

At first, it may seem that this two-level locking is unnecessary and that the same result could be achieved with just one lock operation. That would have been true if there was an one-to-one association between a tasklet and a potentially parallel call. However, as it has already been mentioned and it is discussed in more detail in the following sections (3.6.5 and 3.6.6), one tasklet may represent more than one potentially parallel call. Consequently, an inlining operation does not necessarily result in the removal of the tasklet from the tasklet availability queue. There may still be potentially parallel calls associated with the tasklet that have not been executed and, therefore, available to be converted to parallel tasks.

In inlining computation from a tasklet, only the private to the tasklet lock has to be acquired. As a result, multiple inlining operations on different tasklets may take place

simultaneously. Additionally, an inlining operation does not prohibit the manipulation of the tasklet availability queue (i.e., adding, removing, searching for tasklets).

The two-level locking does not have a significant impact on performance, as the number of inlining operations is expected to be much higher than the number of stealing operations.

3.6.3. Use of Tasklets

Instances of the basic `Tasklet` type, as they were described in the previous section, cannot be used directly to expose parallelism. The basic `Tasklet` type provides the abstract prototype upon which new tasklets are based.

If an implicitly parallel programming language is used for the development of parallel applications, the responsibility for managing the new tasklet representations falls to the compiler. Otherwise, in an explicitly parallel development environment, the application programmer has to design and implement the new tasklet types.

As the process of designing and implementing new tasklet types can become complicated even for a compiler, generic templates (i.e., parameterised types) are available that automatically produce the necessary code for commonly encountered patterns of parallel computations. The three basic patterns and the way tasklets are used to represent them are described in the following sections. The presented pseudo-code for the tasklets is not the actual implementation of the NIP run-time system but a simplified view for presentation purposes.

3.6.4. Function Calls

With eager task creation, a running task—the parent—creates new parallel tasks—the children—for every function call that may be evaluated in parallel. The execution of the parent task continues until a synchronisation point is reached. The parent task will then have to wait until the child task has completed its execution. In the example of Figure 3-1, task *T1* spawns task *T2* and then continues with some sequential code. *T2* executes its computation, function *f*, in parallel. When *T1* reaches the synchronisation point, it has to wait for *T2* to finish. Code 3-6 presents the pseudo-code for tasks *T1* and *T2*.

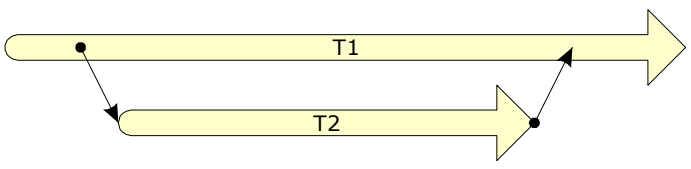


Figure 3-1: The parallel call pattern

```
T1: ...
    fork T2
    ...
    wait T2
    ...

T2: f()
```

Code 3-6: Pseudo-code for *T1* and *T2*

With NIP lazy task creation, the parent task does not spawn a child task. Instead, an instance of a tasklet is created to expose as a potentially parallel call the computation that the child task would have evaluated. Code 3-7 shows the interface and implementation of a parameterised tasklet type whose instances may expose any function as a potentially parallel call (i.e., the function is used as a parameter to the type). Only the `createTask()`, `executeTask()`, and `waitOrInline()` methods are implemented. In this case, the functions to be exposed as potentially parallel calls do not return a value and, therefore, there is no need to provide an implementation for the `returnTask()` method.

Instances of the `ExposeFunctionTasklet<Function>` tasklet type automatically add themselves to the tasklet availability queue by calling the `activate()` method from within their constructor. When an idle processor that is looking for work calls the `createTask()` method, the tasklet instance is immediately removed from the tasklet availability queue by the `deactivate()` method. The tasklet represents just one potentially parallel call and, therefore, once the call is stolen, there is no need for the tasklet to remain in the tasklet availability queue. When the `executeTask()` method is called by the new parallel task, the function passed as a template parameter to the tasklet type is executed.

```
type ExposeFunctionTasklet<Function g> : public Tasklet
public:
   ExposeFunctionTasklet ()
   bool createTask(Task&)
   void executeTask(Task&)
   void waitOrInline()

ExposeFunctionTasklet<Function g>:: ExposeFunctionTasklet()
   activate()

bool ExposeFunctionTasklet<Function g>::createTask(Task&)
   deactivate()
   return true

void ExposeFunctionTasklet<Function g>::executeTask(Task&)
   g()

void ExposeFunctionTasklet<Function g>::waitOrInline()
   if waitForStolenTasks()
      // Do nothing. The function was stolen and
      // executed by a parallel task.
   else
      g()
```

Code 3-7: Pseudo-code of a tasklet that exposes a function call as a potentially parallel call

When the parent task that created the tasklet reaches the synchronisation point, it calls the `waitOrInline()` method. The `waitForStolenTasks()` method is used to

remove the tasklet from the tasklet availability queue if it has not already been removed. Then, three possible scenarios are possible:

- A new parallel task was created to evaluate the potentially parallel call and it is still active: the `waitForStolenTasks()` method blocks the execution of the parent task until the parallel task completes.
- A new parallel task was created to evaluate the potentially parallel call and it has already completed: the `waitForStolenTasks()` returns immediately.
- The potentially parallel call has not been stolen: the `waitOrInline()` method executes the function like a sequential call.

Code 3-8 shows how the program of Code 3-6 (page 52) is transformed to use tasklets and NIP lazy task creation.

```
T1: ...
    ExposeFunctionTasklet<f> tasklet
    ...
    tasklet.waitOrInline()
    ...
```

Code 3-8: The NIP lazy task creation version of the pseudo-code in Code 3-6

3.6.5. Iterative Computations

Iterative computations unveil the full strength of the tasklet construct. When a tasklet is encountered, it is registered with the execution environment and control is returned immediately to the executing task. The task may continue to execute unrelated computation while parallel tasks are created to execute iterations from the tasklet as computational resources become available. There exists a synchronisation point at which all the iterations that have not been converted to parallel tasks are executed sequentially (inlined). However, even during the inlining process, if a processor becomes idle and requests work, remaining iterations may still be converted to parallel tasks. Code 3-9 presents an example in pseudo-code of a typical iterative computation and its equivalent tasklet representation (the iteration-related lines are highlighted).

```
unrelated code1
start loop n iterations
   call function
end loop
unrelated code2
```

```
create tasklet(n, function)
unrelated code1
unrelated code2
wait for tasklet
```

Code 3-9: Relation between an iterative computation and its tasklet representation

Although a tasklet immediately returns control to the executing task, so resembling a closure-like representation, the execution semantics of a tasklet differ from those of a closure. Code 3-10 presents two more versions of the iterative, parallel computation

shown in Code 3-9. Now, closures are used to represent the potentially parallel calls, the iterations. In the first version, logical parallelism may be lost, as the fragments of unrelated code must be executed sequentially, before and after the iterative computation. Additionally, only one potentially parallel call is exposed to the execution environment at any particular time. If the granularity of that call is fine, idle processors will have little opportunity to steal it because it is going to be inlined almost immediately. An alternative approach, which has not been suggested by anyone in the literature, can also be considered (the second version of Code 3-10). A loop exposes all the iterations as potentially parallel calls, giving the opportunity to idle processors to steal them while the unrelated fragments of code are executed. At the end, another loop is required to synchronise with all the closures. In both versions, there is an one-to-one association between iterations and closure representations. The run-time overhead of creating all the closures and then synchronising with them can be significant. As the NIP lazy task creation version shows (Code 3-9), the tasklet representation may solve this problem. The way in which only one tasklet instance can represent the parallelism in iterative computations is described next.

```
Unrelated code1
start loop n iterations
   closure(function)
   closure.wait
end loop
unrelated code2
```

```
start loop n iterations
   closure(i, function)
end loop
unrelated code1
unrelated code2
start loop n iterations
   wait for closure(i)
end loop
```

Code 3-10: Two closure representations of the same iterative computation

A Tasklet Type for Iterative Computations

In NIP lazy task creation, a tasklet instance may contain enough information to represent a pool of potentially parallel calls. One of the most common patterns of parallelism is a parallel loop, where iterations can be evaluated in parallel. A tasklet type can be designed to expose the parallelism in such computations (Figure 3-2).

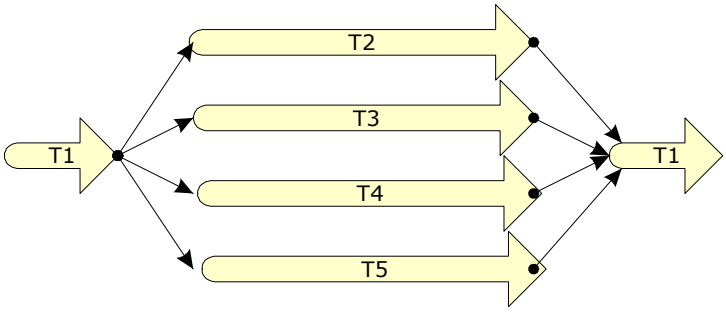


Figure 3-2: The parallel loop pattern

The design, implementation, and application of tasklet types, whose instances expose the available parallelism in loops, are illustrated through an example. In the example, a function *f* is applied on all the elements of a vector of size *N*. The serial version of this iterative computation is shown in Code 3-11. The NIP lazy task creation version is presented in Code 3-12.

```
VectorType vector(N)
for i = 1 to N
   vector(i) = f(vector(i))
```

Code 3-11: Serial version of the application of a function onto the elements of a vector

```
VectorType vector(N)
MapTasklet<VectorType, f> tasklet(vector)
...
tasklet.waitOrInline()
```

Code 3-12: NIP lazy task creation version of the application of a function onto the elements of a vector

The `MapTasklet<VectorType, f>` type is used to expose the available parallelism in the iterative computation where the function `f` is applied to all the elements of the vector `vector` of type `VectorType` and of size `N`. Before looking into the design of the new tasklet type, two additional methods must be added to the interface of the basic `Tasklet` type. The two additional methods (`beginCriticalSection()` and `endCriticalSection()`) provide access to the private lock of the tasklet and allow new tasklet types to define critical sections. As will be seen, there are cases where the `createTask()`, `returnTask()`, and `waitOrInline()` methods attempt to simultaneously access the same data structures of a tasklet instance. The defined critical sections are used to protect the integrity of the internal state of tasklets from such concurrent accesses.

The design and implementation in pseudo-code of the `MapTasklet` type is presented in Code 3-13. The constructor of the tasklet accepts a reference to a vector object and initialises the `_vector` private data member. The constructor also adds the tasklet to the tasklet availability queue by calling `activate()`. In this way, all the iterations of the parallel loop are made available to the execution environment as potentially parallel calls as soon as an instance of `MapTasklet` is created.

```
type MapTasklet<type T, Function g> : public Tasklet
public:
   MapTasklet(T&)
   bool createTask(Task&)
   void executeTask(Task&)
   void returnTask(Task&)
   void waitOrInline()
private:
   int _index
   T&  _vector
   int _stealAnIteration()

MapTasklet<type T, Function g>::MapTasklet(T& v) : _vector(v),
                                                   _index(0)
   activate()

bool MapTasklet<type T, Function g>::createTask(Task& task)
   i = _stealAnIteration()
   if i < _vector.size()
      put i into task
      put _vector[i] into task
      return true
   else
      deactivate()
      return false

void MapTasklet<type T, Function g>::executeTask(Task& task)
   get element from task
   put g(element) to task

void MapTasklet<type T, Function g>::returnTask(Task& task)
   get result from task
   get i from task
   _vector[i] = result

void MapTasklet<type T, Function g>::waitOrInline()
   while(true)
      i = _stealAnIteration()
      if i < _vector.size()
         _vector[i] = g(_vector[i])
      else
         stop while
   waitForStolenTasks()

int MapTasklet<type T, Function g>::_stealAnIteration()
   beginCriticalSection()
   i = _index++
   endCriticalSection()
   return i
```

Code 3-13: Design and implementation of the `MapTasklet` type

There is a private data member, the `_index`, which points to the first element in the vector for which the map function has not been executed, either as a separate parallel task or inline. When an idle processor looking for work calls the `createTask()` method, the `_index` is atomically increased by the `_stealAnIteration()` private method call. The previous value of the `_index` is returned to the calling method. The returned value identifies a unique iteration (an element of the vector). If the returned value is within the limits of the vector, the `Task` object is configured. The value identifying the iteration to be evaluated and the element of the vector upon which the function is going to be applied

are placed into the `Task` object. If the `_stealAnIteration()` method returns a value that exceeds the limits of the vector, the tasklet instance is removed from the tasklet availability queue because there are no more potentially parallel calls available.

The `_executeTask()` method only requires the element of the vector upon which the map function is to be called. The index that is already in the `Task` object is left untouched. The map function is called and the `element` is passed as an argument. The result of the function is stored in the `Task` object.

When the `Task` object is returned to the tasklet it was stolen from and the `returnTask()` method is called, both the result and the index are retrieved. The index is used to indicate the position in the vector where the result should be stored. The `returnTask()` method completes the cycle of a stolen iteration (steal, execute, and return).

If the granularity of the computation in Code 3-12 between the instantiation of the tasklet and the synchronisation point (i.e., the `waitOrInline()` call) is large enough, all the iterations may be stolen and executed as separate parallel tasks. When the `waitOrInline()` method is called, a loop is entered and lasts until all the iterations have been executed, either inline or stolen for execution in parallel. Of course, some or all of the iterations may already have been stolen and executed in parallel. The `waitOrInline()` will start from the last available iteration, if there is one available. A vector index is atomically chosen by the `_stealAnIndex()` method. If the index does not exceed the limits of the vector, the map function is called and the associated position in the vector is updated. Finally, the `waitForStolenTasks()` is called to remove the tasklet from the tasklet availability queue and wait for any possible executing parallel tasks.

The example presented in this section demonstrates the strength of the tasklet representation for potentially parallel calls. Only one tasklet instance is required to expose the parallelism in an entire loop, rather than one per iteration as is the case with earlier schemes.

Granularity Considerations

Although the tasklet manages to overcome the problems associated with other representations in exposing the parallelism in iterative computations, it suffers from the same granularity issue affecting the earlier approaches to lazy task creation. A tasklet may expose the parallelism of a parallel loop that consists of fine-grained iterations. When iterations are stolen from the tasklet to be executed as separate parallel tasks, the

granularity of the resulting tasks is also going to be fine. The run-time costs of lazily creating parallel tasks may overwhelm the total execution time.

Unlike previous approaches, the NIP lazy task creation technique offers a solution to the granularity issue mentioned above. The flexibility of the tasklet representation allows for more advanced tasklets than the one presented in Code 3-13 to be designed and implemented. Tasklets may be created that are able to dynamically increase the granularity of the parallel tasks without reducing the degree of logical parallelism available.

When a processor issues a request for a new task, a group of iterations can be stolen rather than just one. The stealing processor creates a new tasklet instance to make sure that the stolen iterations are still available to other processors, if any runs out of work. Now, the new parallel task will inline iterations from the stolen group and, therefore, its granularity is coarser compared to the granularity of a task that only steals and executes one iteration.

The size of the group of iterations to be stolen at one time can be determined in various ways. For example, heuristic information about the execution of iterations that is collected at run-time (*e.g.*, the execution time of iterations) can be used to dynamically change the size of the group. Information collected from the execution environment, such as the number of available processors and communication delays, in combination with information about the total number of iterations in the loop may be used to determine a fixed or varying group size.

The performance of NIP lazy task creation can be further improved at run-time by allowing the tasklets with grouping functionality to reduce the available logical parallelism, when appropriate. The parallel tasks that are created to execute a group of stolen iterations do not expose them as potentially parallel calls but, instead, execute them all as sequential calls. Of course, this can only be possible when enough information is known at run-time about the execution environment (*e.g.*, processor, memory, communication, etc.), the granularity of the iterations, and the details of the data structures involved (*e.g.*, number of elements).

Several tasklet types that provide grouping functionality are provided by the NIP run-time system.

Consecutive Parallel Calls

Iterative tasklets can be used to improve the representation of a series of independent potentially parallel calls. There is an one-to-one association between such potentially parallel calls and tasklets, as discussed in Section 3.6.4. A series of parallel calls, one

following the other, will result in a number of tasklets being created. The run-time overhead due to tasklet instantiation costs may be reduced with the help of an iterative tasklet.

If all the potentially parallel calls were placed in a data structure such as a vector or a list, an iterative tasklet similar to that of Code 3-13 (page 56) could be used. As a result, only one tasklet is used to expose the logical parallelism associated with a series of independent parallel function calls.

3.6.6. Recursive Computations

The last pattern of parallel calls to be examined is that of recursive computations. The parallel calls form a tree-like computational flow, like the one shown in Figure 3-3. Each of the potentially parallel calls in the recursive computation can be represented as a separate tasklet, like the one described in Section 3.6.4. However, as it was the case with iterative computations, it is more efficient to design and implement a tasklet type whose instances expose the parallelism in whole recursive computations.

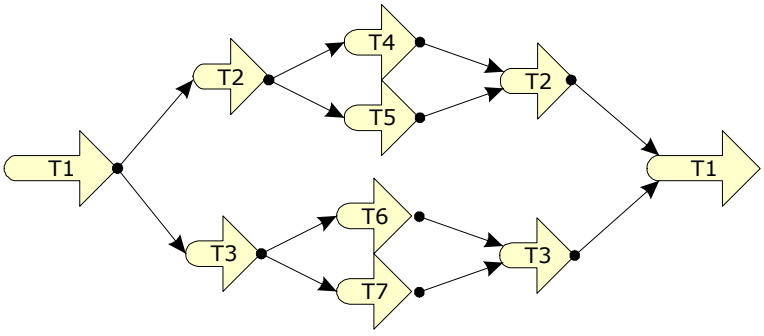


Figure 3-3: The parallel recursion pattern

```
int nfib(int n)
   if n > 1
      return nfib(n - 1) + nfib(n - 2)
   else
      return 1
```

Code 3-14: The serial version of *nfib*

```
int nfib(int n)
   if n > 1
      ExposeFunctionTasklet<nfib> tasklet(n - 2)
      tmp = nfib(n - 1)
      tasklet.waitOrInline()
      return tmp + tasklet.result()
   else
      return 1
```

Code 3-15: The NIP lazy task creation version of *nfib* using simple tasklets

The serial version of a recursive algorithm, the calculation of the fibonacci number, is presented in Code 3-14. A possible implementation of the same algorithm in NIP lazy

task creation using simple tasklets is illustrated in Code 3-15. An appropriate tasklet is designed and implemented whose instances expose a function as a potentially parallel call. Unlike the example of Section 3.6.4 (page 52) the functions exposed by the tasklets accept an argument, the fibonacci number to be calculated, and return a result. Therefore, the tasklet type must provide the appropriate functionality. The detailed description of such a tasklet is unnecessary for the purposes of this discussion, as it greatly resembles the one in Section 3.6.4 (page 52).

The problem with using a tasklet type like `ExposeFunctionTasklet` is the one-to-one association that exists between tasklet instances and exposed potentially parallel calls. A new tasklet has to be created for every recursive call to *nfib*. If the computation at each recursive node is fine-grained, as it is in this example, there might be cases where a tasklet instance is added to the tasklet availability queue and almost immediately removed from it. The processing activity on the tasklet availability queue may become very high. Since there is a lock that must be acquired before any operation on the queue, a congestion point may arise. Other tasks that are trying to add new tasklets to the tasklet availability queue or remove old ones from it will be delayed. Stealing operations issued by idle processors may also be affected. Previous approaches of lazy task creation also suffer from the same problem but do not attempt to provide a solution.

Having identified the problem as the frequent addition and removal operations on the tasklet availability queue, a similar solution to iterative computations was designed. NIP lazy task creation provides tasklets that are able to expose the parallelism in whole recursive computations. Like their iterative counterparts, the recursive tasklets are nothing more than specialised implementations of the basic `Tasklet` type. Therefore, they also adhere to the same interface.

```
int nfib(RecursiveTasklet& tasklet, int n)
   if n > 1
      NfibNode node(tasklet, n - 2)
      tmp = nfib(n - 1)
      tasklet.waitOrInline()
      return tmp + node.result()
   else
      return 1

main
   RecursiveTasklet<NfibNode> nfibTasklet(N)
   ...
   nfibTasklet.waitOrInline()
```

Code 3-16: The NIP lazy task creation version of *nfib* using a recursive tasklet

Code 3-16 shows how the recursive tasklet is used to implement the *nfib* algorithm. Only one instance of the `RecursiveTasklet` type is constructed. Accordingly, only one addition operation is required on the tasklet availability queue. At every recursive call, a small object, an instance of the `NfibNode` type, is allocated on the stack of the running task. The object stores the fibonacci number that may be evaluated in parallel (n-2). The object also stores the result of the calculation, when that completes. Effectively, an `NfibNode` instance represents a branch of the computational tree.

The `NfibNode` instance is automatically added in a queue, which is privately held by the recursive tasklet. No additional memory space is allocated as the required pointers for the construction of the queue are allocated on the stack of the running task as part of each `NfibNode` instance. The recursive tasklet makes use of its private lock (Section 3.6.2, page 50) to maintain the integrity of the queue.

In the example of Code 3-16, when the `createTask()` method is called on the tasklet, the top of the queue of `NfibNode` instances is checked. If there are available instances, one is stolen and is sent to the idle processor to be evaluated as a parallel task. No logical parallelism is lost because the processor receiving the new task will create a recursive tasklet. The new tasklet instance is used to expose any potential parallelism that may result from the execution of the stolen computational branch.

When the `waitOrInline()` method is called on the tasklet, the bottom of the queue is examined. If the last `NfibNode` in the queue was stolen and the result returned, then the execution may proceed. If, however, the result has not been returned yet, the execution of the running task will have to suspend. Finally, if the last entry in the queue was not stolen, it is inlined.

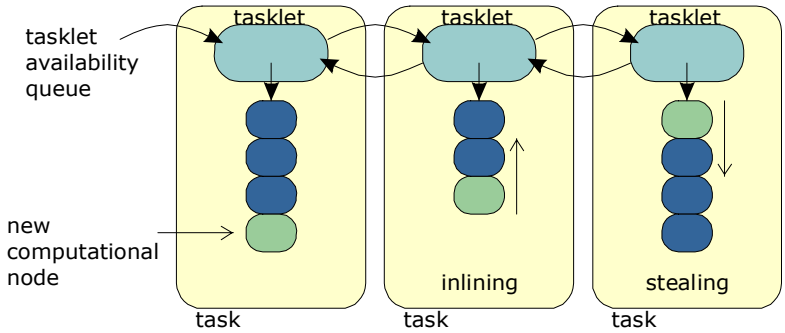


Figure 3-4: Operations on the queue maintained by recursive tasklets

Figure 3-4 summarises the discussion on recursive tasklets and the way they are used. It illustrates three running tasks, each one of which has added a recursive tasklet to the tasklet availability queue. When a new recursive call is made, a new object representing the new computational node in the tree-like computational flow is added at the end of the

queue. Inlining operations remove node representations from the bottom of the queue, while stealing operations use nodes from the top of the queue.

Nodes that are encountered early in the computational flow occupy the top of the queue that is maintained by the recursive tasklets. If the recursive computation is balanced, the nodes at the top of the queue will represent branches of coarser granularity than the ones at the bottom of the queue. In such cases, recursive tasklets favour the creation of coarse-grained parallel tasks.

3.6.7. Implementation

The NIP lazy task creation technique and the tasklet representation for potentially parallel calls have been implemented in C++, as part of the NIP run-time library (Chapter 5). Several tasklet types are available as template C++ classes. Compilers or programmers may use the template classes to easily expose parallelism to the execution environment. In most cases, there will not be any need to design and implement new specialisations of the basic `Tasklet` type. Additionally, the available tasklet types incorporate optimisations that are transparently used on shared-memory multiprocessors.

The NIP lazy task creation technique and the tasklet construct are compatible with conventional stack implementations. There is no need for specialised hardware or compiler support, and pre-emptive threads are supported. All the above contribute to the portability of NIP lazy task creation on any platform. Additionally, the technique was designed to support both shared-memory and distributed memory architectures.

There is a tight integration with the load balancing service provided by the NIP run-time. The integration allows for communication between processors and computation to be overlapped, resulting in better performance (Chapter 5). Also, a task that is blocked on a `waitOrInline()` call will give up the processor for either another task that is ready to run or a new one that is lazily created so that computational resources are not wasted. The NIP run-time, the maintenance of the tasklet availability queue, and the load balancing service are described in detail in Chapter 5.

3.7. Discussion

Lazy task creation techniques attempt to efficiently manage at run-time the identified logical parallelism in applications. The main goal of the techniques is to reduce parallelism so that the execution costs due to the management of excessive parallelism are avoided.

The main concept is that parallel tasks are only created when computational resources become available.

The NIP lazy task creation technique follows the same concept as the earlier approaches but it introduces a new construct for representing potential parallel calls, the tasklet. The key features of the NIP lazy task creation technique and the tasklet construct many of which cannot be found in earlier approaches are:

- The ability to represent the parallelism in whole iterative and recursive computations using only one instance of the tasklet construct.
- The allocation of the tasklet construct on the stack rather than the heap of the executing application.
- A two-level locking scheme that allows concurrent inlining and manipulation of the tasklet availability queue.
- The support for pre-emption and symmetric multiprocessing environments.
- An object-oriented design that eases the process of introducing tasklets with new functionality.
- The option of adjusting at execution time the grain size of the parallel tasks that were lazily created from tasklets representing parallel loops (i.e., grouping).
- The availability of the technique through library support. There is no requirement for compiler support or special stack manipulation.
- The support for distributed memory environments.

CHAPTER 4

NIP SOFTWARE-BASED DISTRIBUTED SHARED MEMORY

The NIPDSM system is the implementation of the memory system as that is defined by the NIP execution model semantics and according to the requirements of the NIP programming model. NIPDSM provides applications with an object view of the memory and strict consistency semantics.

NIPDSM introduces, as part of its design and implementation, the relaxed NIP entry consistency model, which guarantees the required strict memory semantics. NIP entry consistency defines that objects are implicitly associated with locks and facilitates the coupling of synchronisation and cache management.

Cache management techniques are incorporated in the NIPDSM system to reduce object access delays. Applications may benefit from spatial locality, temporal locality, recurring object access, and object associations.

The rest of this chapter explores the issues related to the design and implementation of distributed shared memory systems and describes in details the all-in-software, object-based NIPDSM system.

4.1. The Shared Memory Abstraction

Computer architectures based on multiple processors that physically share memory provided one approach to the demand for greater performance (Chapter 1). The architectures, which are mostly referred to as tightly coupled multiprocessors or just multiprocessors, favour the popular shared-memory programming and execution models (Chapter 2). Read and write memory operations provide a convenient way of accessing the physically shared memory. Synchronisation operations may be used to protect information stored in the memory from concurrency related issues.

Another approach to building parallel computer systems is to have memory distributed amongst a number of processors. Every processor in a parallel system has a private memory and all the processors are interconnected together via one of the various interconnection topologies that are available. Processors can only access data held in remote memories by exchanging messages with each other. Thus, such parallel systems are known as message passing architectures or just multicomputers. The hardware architecture of multicomputers favours the message passing programming and execution models. The main advantage of multicomputer over multiprocessor designs is scalability. Additional processors with private memory can be introduced into a parallel system, contributing to its efficiency but not increasing the complexity of its design.

Despite their scalability advantage, multicomputers and the message passing programming and execution models that they favour have a negative effect on the simplicity of the application development process (Lu et al. 1997). Unlike the shared memory programming model which is favoured by multiprocessors, the message passing programming model expects application developers to manage the movement of data across the distributed memories of a multicomputer system. The complexity of application development based on message passing has lead to the introduction of shared memory abstractions for multicomputer architectures. Applications and/or developers are presented with the illusion of an existing shared memory system, known as a distributed shared memory (DSM) system. The abstraction of shared memory is implemented in hardware, software, or in a combination of the two.

The hardware approach to implementing shared memory abstractions on distributed memory architectures usually increases the cost and the complexity of a parallel system. The hardware implementations are efficient and are considered as extensions to the

techniques used in the design of cache coherent multiprocessors but they are not as flexible as the software approaches.

Software-based DSM systems have been a convenient tool for researchers during the last decade in their endeavour to study ideas and techniques on hiding the hardware message-passing architectures and replacing them with the shared memory abstraction. Often, techniques that were originally developed and tested on software-based DSM platforms found their way on hardware implementations. The flexibility in the design and the low implementation costs of software-based DSM systems allowed researchers to better understand and extensively test all the issues of implementing shared memory abstractions on top of message passing hardware designs.

This chapter does not attempt an in-depth comparison between software and hardware implementations of DSM systems. Instead, the reader is referred to the work by Cox et al. (Cox et al. 1994). Although most of the issues to be discussed are common to hardware and software implementations, this chapter mostly considers the software-based DSM systems. In the rest of the chapter, the DSM design issues and their effect on efficiency is discussed (Section 4.2), existing techniques used to maintain consistency of replicated information across a DSM system are described (Section 4.3), and a representative group of existing systems is considered (Section 4.4). Finally, the design of a new, all-in-software, object-based DSM system and the caching techniques it incorporates are thoroughly examined in Sections 4.5 to 4.8.

4.2. The Design Considerations for DSM Systems

The decisions that are made during the design process of a DSM system may have a significant impact on the performance of parallel applications. Apart from the obvious decision on whether the DSM will be implemented in hardware or software, numerous other issues need be considered. This section explores those issues and examines the ways in which they may influence the implementation of a DSM system.

4.2.1. Structure, Sharing Unit, and Granularity

Perhaps, the most divisive design aspect of DSM systems is the layout of the shared memory that is presented to parallel applications. There exist two main design approaches: applications may perceive memory (a) as a continuous, unstructured shared space, or (b) as a collection of data structures or objects.

Associated with the decision about the layout of the shared memory is the unit of sharing or transfer unit, which can be a byte, a word, a page, a data structure, or object. The size of the sharing unit is also known as the granularity of the shared memory. The layout and the granularity of the shared memory play a key role in the rest of the design considerations for a DSM system and ultimately they greatly affect its performance.

Two design approaches have emerged as dominant in the design of DSM systems: page-based and object-based, where the sharing units are the page and the object respectively. Examples of page-based systems include TreadMarks (Keleher et al. 1994) and IVY (Li 1986), while cases of object-based systems are Linda (Ahuja et al. 1986), Orca (Bal et al. 1992), and Emerald (Jul et al. 1988). Tanenbaum in (Tanenbaum 1995) correctly identifies DSM systems that belong to neither the page-based nor the object-based approaches. Examples of such DSM systems are Munin (Carter et al. 1991) and Midway (Bershad and Zekauskas 1991), which although provide an unstructured, linear memory space, they exploit information made available by applications about the shared variables in the synchronisation operations.

Research efforts in the field of DSM system have yet to produce an irrefutable conclusion on the superiority of one approach over the other in terms of their efficiency and suitability for parallel programming (Buck and Keleher 1998; Levelt et al. 1992). Nevertheless, there is a good understanding of the differences between the approaches and the issues involved. It is clear that the memory access patterns of a particular parallel application in combination with the characteristics of the hardware platform may favour one approach over the other (Buck and Keleher 1998; Levelt et al. 1992).

Page-based

The most common way of perceiving memory is that of a continuous, unstructured linear space of addressable locations. For management purposes, the memory space is often partitioned into pages. Early research efforts resulted in the design and implementation of hardware and software DSM systems that replicated the semantics of page-based memory management. In fact, page-based DSM systems provide parallel applications with a view of the distributed memory that cannot be distinguished from the physically shared memory found on multiprocessor architectures with a conventional operating system (*e.g.*, Linux, Windows 2000, SunOS, etc.).

The sizes of the pages, the consistency semantics (to be discussed shortly), and the caching techniques, if any, may differ between implementations of page-based DSM systems but the main principles are the same. Applications access memory locations via

read and write operations. When a memory location that is not available locally is addressed, the page containing the requested location is fetched from the remote memory transparently to the parallel application. Once the page has arrived, the application can access it as if it was a local page.

Unlike hardware DSM systems, which use specially designed and constructed hardware components to trap the read and write operations on remote memories, software page-based systems usually utilise the virtual paging mechanism of the underlying operating system. Li and Hudak were the first to investigate the virtual paging mechanism of an operating system as the means to implementing a shared virtual memory on a multicomputer (a collection of Apollo workstations) (Li 1986; Li and Hudak 1989).

As with virtual paging techniques, page-based DSM systems favour applications that exhibit spatial locality in memory access. With DSM systems that cache fetched pages, an application is able to access memory locations without always having to pay the penalty of a remote fetch operation. Actually, it is unlikely that an efficient all-in-software DSM system without support for caching could be implemented. With caching, larger page sizes increase the probability for spatially adjacent memory locations to be available when required by an application. However, the chances of false sharing also increase as more than one parallel process may require access to different memory locations on the same page. Different consistency models (Sections 4.2.2 and 4.3) attempt to provide a solution to this problem but not without sacrificing efficiency.

Object-based

A different approach to perceiving memory is that of a shared, structured space. The memory is arranged as a collection of data structures or objects rather than as an unstructured linear space of addressable locations, as it is the case with page-based DSM systems. Both data structures and objects are means of representing state. However, objects encapsulate the state and provide methods as the only way of accessing it. For the purposes of the discussion in this chapter, the terms 'data structure' and 'object' are used interchangeably while referring to the memory layout of a DSM system, unless it is stated otherwise.

There is no fixed granularity for the sharing unit because the size of the objects may vary. As a result, object-based DSM systems do not suffer from the problem of false sharing to the same extent as page-based DSM systems. However, unless special caching techniques are used, the benefit of spatial locality in memory access that page-based DSM

systems feature is lost because only the requested objects are transferred during a memory operation.

4.2.2. Memory Consistency

During the design of a DSM system, a decision has to be made about the memory consistency protocol(s), or memory consistency model(s), that are going to be supported. Although the terms 'memory consistency' and 'cache coherence' are usually used interchangeably in the DSM-related literature, this thesis agrees with the argument presented in (Adve and Gharachorloo 1996) and considers cache coherence to be a subset of memory consistency. A memory consistency protocol is the behavioural specification of the memory system as seen by applications and programmers and not just the approach taken into preserving the coherence of replicated data stored in caches. A memory consistency protocol consists of a set of rules about the way memory should be accessed by applications and a comprehensive description of the guarantees provided if those rules are met. In essence, a consistency model is a contract between the software and the memory (Adve and Hill 1990).

Perhaps, strict consistency is the most comprehensible model, as it resembles the memory model of the von Neumann computer architecture: *"Any read to a memory location x returns the value stored by the most recent write operation to x"* (Tanenbaum 1995). However, the model is extremely difficult and inefficient to implement on distributed memory architectures due to the explicit use of time. Sequential consistency is a model with strict semantics but without the notion of time in its definition. The formal definition of the model, as given by Lamport, specifies that the memory is sequentially consistent when "*the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program"* (Lamport 1979). Although the implementation of sequential consistency on a DSM system is feasible, it is still not an efficient approach due to the imposed sequentiality, or ordering rules, in memory access.

In pursuing better performance, research efforts in the fields of cache design for multiprocessor architectures and that of DSM systems have resulted in alternative, more relaxed consistency models. The proposed models are considered relaxed when compared to the consistency models with strict semantics because they eliminate some of the ordering rules. The memory access operations are not strictly ordered and as a result, performance optimisations can be implemented. Provided applications follow certain rules in the way they access the memory, the semantics of sequential consistency seem to

be preserved. Relaxed consistency models may also support chaotic memory access as required by some type of applications (Protic et al. 1998).

For a comprehensive discussion of issues related to memory consistency, the reader is referred to (Adve and Gharachorloo 1996; Mosberger 1993; Tanenbaum 1995; Zucker and Baer 1992). Additionally, Section 4.3 attempts an in-depth look in a small number of characteristic and widely used consistency models.

4.2.3. Data Access

Another decision that affects the design of a DSM system is the method of accessing the sharing unit (i.e., page or object). The sharing unit may be assigned to a specific processing node throughout its lifetime or it may migrate between the nodes requiring access to it. There may exist only one copy of the sharing unit across a parallel system (Single Reader/Single Writer approach) or replicated copies may be permitted in an attempt to facilitate more parallelism. In the latter case, only concurrent read memory operations may be allowed on the replicated copies (Multiple Reader/Single Writer approach) but there may also be cases where both read and write memory operations are permitted (Multiple Reader/Multiple Writer approach). The choice and the implementation of a memory consistency model greatly depend on the selection of the data access algorithm for the DSM system but the vice-versa is also true.

Associated with the selection of the data access algorithm are other issues that may too have an impact on the implementation and the performance of a DSM system. For example, if access to a sharing unit is required, a method of locating the processing node that currently holds that unit is necessary. If the sharing unit is not allowed to migrate, the method of locating it is usually straightforward. Otherwise, a technique of monitoring the movement of the sharing unit must be devised (*e.g.*, forwarding requests to the last known location, distributed directories, etc.).

A survey of different algorithms for DSM systems may be found in (Stumm and Zhou 1990) while (Protic et al. 1998) and (Tanenbaum 1995) provide a very good starting point for related issues.

4.2.4. Implementation

A software-based DSM system may be implemented by any of the layers in the layered approach to the parallel computing paradigm presented in Section 2.1.6 (page 17). It may be integrated into the operating system, it may be implemented as part of the run-time system, it may be provided by a software library, or it may be incorporated into a

programming language and its compiler. Combined approaches also exist, where the implementation of the DSM system is shared between different layers.

4.2.5. Heterogeneity

A requirement that may be imposed on the design of a DSM system is heterogeneity. There may be cases where a DSM system is going to be used on multicomputers that consist of a collection of heterogeneous hardware architectures. DSM systems with heterogeneous support are not common, as their efficiency has to be compromised to allow for the necessary translation between the different representations of data.

4.2.6. Efficiency

The Impact of the Hardware and the Memory Access Patterns

The performance considerations have been an essential driving force in researching new techniques and design approaches for DSM systems. Undoubtedly, the communication between the processing nodes of distributed memory architectures has been the primary target for the introduced enhancements. In distributed memory architectures, no matter how fast the interconnection hardware may be, an access operation on a remote memory is always slower than an operation on the local memory. Most of the techniques (*e.g.*, caching, memory consistency models, etc.) attempt to reduce the amount of data transferred between processing nodes in order to achieve better performance.

The efficiency of a DSM system design greatly depends on the characteristics of the underlying hardware architecture. A DSM implementation that performs well on a multicomputer with slow processors but high-speed communications hardware may be inefficient when used with fast processors and slow interconnection networks. Of course, the opposite may also be true. The investigation by Buck and Keleher in (Buck and Keleher 1998) attempts to demonstrate the way the hardware architecture may influence the design choices of a DSM system. Their investigation also shows the effect that the memory access patterns of applications may have on the efficiency of a DSM system. Buck and Keleher simulated the execution of four applications using a page-based and an object-based DSM system. Part of their investigation concluded that on hardware architectures where the communication between processing nodes is expensive the page-based approach is more efficient (Buck and Keleher 1998). Their conclusion is in contrast to the general view that object-based DSM designs introduce less communication overhead because of the reduced number of messages they require. However, the result is attributed to the memory access patterns of the four applications used, which exhibit high locality in they way data is accessed.

In an attempt to support the different memory access patterns that parallel applications exhibit, adaptable memory consistency techniques have been devised. For example, Munin (Carter 1995; Carter et al. 1991) offers a variety of consistency models that can be used simultaneously for different part of the shared memory space.

Computation and Communication

An important feature that has recently found its way into DSM systems is the ability to overlap computation and communication. When a remote memory operation is to take place and while a fetch request is in progress, the DSM system may block the processing node until the data is available locally. Alternatively, the DSM system may free the processing node so that other computations can be executed.

Clearly, the allocation of computation work to processing nodes is the duty of a job scheduler and, at first, it may appear unrelated to DSM systems. However, a DSM system may be designed and implemented to work together with the job scheduler in order to make the overlapping of communication and computation feasible.

Scalability

Finally, a concern for designers is the ability of the DSM system to scale to a great number of processing nodes. The main advantage of multicomputer over multiprocessor architectures is their scalability (Section 4.1). Consequently, the design of a DSM system should preserve that advantage and allow parallel applications to benefit from the addition of processing nodes.

In designing a scalable DSM system, centralised points of accessing global information should be avoided as they may cause bottlenecks and result in reduced efficiency. For example, some DSM systems maintain a directory of the location of the migrating sharing units. If the directory is stored on only one processing node, that node may become a 'hotspot' and eventually it will be unable to efficiently serve all the requests that are sent to it.

4.2.7. Discussion

Undoubtedly, more research effort is required on exploring the diverse design approaches as seen above, especially if software DSM systems are to be used on large-scale multicomputer systems. Until now, there have not been any published results of software DSM systems utilising a large number of processing nodes (*e.g.*, greater than one hundred). Possibly, as software DSM systems are tested on larger systems, some of the

proposed techniques and designs will emerge as more suitable for the majority of the parallel applications running on multicomputers.

4.3. Existing Relaxed Memory Consistency Models

As mentioned earlier, consistency models can be viewed as a contract between the memory system and the applications. Memory consistency models consist of a set of rules and requirements that define the functionality of a memory system. During the last decade, a significant number of consistency models have been proposed in the literature (Tanenbaum 1995): strict consistency, sequential consistency (Lamport 1979), pipelined RAM consistency, processor consistency (Ahamad et al. 1992), casual consistency, weak consistency, release consistency (Gharachorloo et al. 1990), lazy release consistency (Keleher 1995; Keleher et al. 1992), entry consistency (Bershad and Zekauskas 1991; Bershad et al. 1993), scope consistency (Iftode et al. 1998).

It is not in the scope of this thesis to thoroughly examine and compare all the consistency models. Performance evaluation of consistency models can be found in (Adve et al. 1996; Keleher et al. 1995; Zhou et al. 1997). Instead, three of the basic models are now described in detail. Release consistency was amongst the first relaxed models to utilise information about synchronisation operations while lazy release consistency is a better implementation of the original model. Entry consistency was the first model designed specifically for a software-based DSM system and it utilised information about the data structures in an application. It is the basis for the consistency model used in NIPDSM (Section 4.5).

4.3.1. Release and Lazy Release Consistency

Release Consistency

Release consistency is a relaxed model (Gharachorloo et al. 1993; Gharachorloo et al. 1990) that was designed as an extension to the weak consistency model. It utilises synchronisation-specific information in order to reduce the amount of data exchanged between processing nodes. To work correctly, release consistency requires that applications adhere to a contract with the memory. Synchronisation constructs, such as locks or barriers, should be used to guard access to shared data.

The operations that may be performed on a lock are acquire and release. An acquire operation informs the memory system that a critical section begins while a release operation indicates the end of that critical section. Access to shared data should only take place within a pair of acquire and release operations. A release operation signals the

memory system to identify the modified data and update all the processing nodes that hold a cached copy of that data. Only after all the cached copies have been updated, the lock can be released and allow another processing node to enter its critical section via an acquire operation.

A barrier blocks the execution of a processing node until all other processing nodes reach the same point. With release consistency and when a barrier has been reached by all the processing nodes, all the shared data are synchronised before execution is allowed to proceed.

The formal definition of the release consistency states that (Gharachorloo et al. 1990):

- *"Before an ordinary read or write access to a shared variable is performed with respect to any processor, all previous acquire accesses must be performed.*
- *Before a release access is allowed to perform with respect to any other processor, all previous ordinary read and write accesses must be performed.*
- *Special accesses (acquire and release) are processor consistent with respect to one another."*[3]

Provided the above rules are not broken by an application, the memory semantics of the sequential consistency model appear to be maintained. However, the strict ordering rules of sequential consistency are relaxed and a more efficient utilisation of the DSM system is possible.

Lazy Release Consistency

A drawback of the original release consistency model is the eager way in which updates to shared variables are propagated to processing nodes. When a release operation on a lock occurs, the updates to shared variables are eagerly transmitted to the processing nodes that already have a cached copy. It is assumed that the receiving processing nodes may need to access the shared variables, which very often may be an erroneous assumption.

Keleher developed an enhancement to the original release consistency model, which he named lazy release consistency (Keleher 1995; Keleher et al. 1992). According to his approach, the updates to shared variables were not automatically propagated to processing nodes on a release operation. Instead, only when a processing node issued an acquire operation the shared variables would be updated. An acquire operation on a lock is seen as the signal that a processing node requires access to the shared variables.

[3] For a description of processor consistency the reader is referred to (Ahamad et al. 1992).

Release consistency was originally developed for the hardware-based DSM system of the DASH multiprocessor. Lazy release consistency was first used in the software-based TreadMarks DSM system (Keleher et al. 1994).

4.3.2. Entry Consistency

Lazy release consistency uses synchronisation operations as signals to the memory system on whether the state of the shared data needs to be updated. When an acquire operation is performed on a synchronisation construct, the modified shared data in memory must be identified and their values synchronised. To that extent, entry consistency (Bershad and Zekauskas 1991; Bershad et al. 1993) is very similar to lazy release consistency. Their main difference is the additional requirement imposed to applications by entry consistency to associate every shared variable with either a lock or a barrier. By associating shared variables to locks, the overhead of identifying and synchronising all the modified data is not incurred, as it is in lazy release consistency, because only the associated variables are examined.

Due to the required explicit association between shared variables and locks, the process of creating an application becomes more troublesome for the programming language compiler and/or the developer. All the shared variables must be identified and linked to a synchronisation construct. It is a difficult and error prone process but if applied correctly it results in the reduction of the communication traffic when compared to lazy release consistency. Additionally, multiple critical sections guarding unrelated shared data may be defined and executed in parallel.

Like all the relaxed consistency models, if an application strictly adheres to the rules, it is given the illusion of a sequentially consistency memory. The formal definition of the entry consistency rules appears in (Bershad and Zekauskas 1991) but it is better expressed in (Tanenbaum 1995):

- *"An acquire access of a synchronisation variable is not allowed to perform with respect to a process until all updates to the guarded shared data have been performed with respect to that process.*
- *Before an exclusive mode access to a synchronisation variable by a process is allowed to perform with respect to that process, no other process may hold the synchronisation variable, not even in nonexclusive mode.*
- *After an exclusive mode access to a synchronisation variable has been performed, any other process' next nonexclusive mode access to that synchronisation variable may not be performed until it has performed with respect to that variable's owner."*

Entry consistency was designed for the Midway DSM system (Bershad and Zekauskas 1991; Bershad et al. 1993), which was implemented as an extension to the C programming language/compiler and it is further described in Section 4.4.2. This thesis considers entry consistency as the most suitable memory model for object-based DSM systems because of the association between shared data and synchronisation constructs. The memory consistency semantics of the object-based DSM system described in Section 4.5 are based on entry consistency.

4.4. Existing DSM Systems

There has been a plethora of hardware, software, and hybrid DSM systems proposed in the literature. A comprehensive presentation of existing systems and their characteristics serves no purpose to the discussion in this thesis. However, the main characteristics of the most influential software-based DSM systems, according to this thesis, are presented. The reader is referred to the books by Protić, Tomašević, and Milutinović (Protic et al. 1998) and Tanenbaum (Tanenbaum 1995) as good starting points for DSM-related concepts and systems.

Midway is the DSM system that utilises the entry consistency model and as such, its design and implementation are of particular interest to the discussion presented in this chapter. Since the proposed all-in-software, object-based NIPDSM system (Section 4.5) uses a variation of the entry consistency model, a comparison between the two implementations is within the scope of this thesis. To that extent, a detailed description of the design and implementation of Midway is considered necessary.

4.4.1. Influential Systems

The following table lists only a representative subset of existing DSM systems and their important characteristics. The presented systems are considered by this thesis to have contributed to the main concepts in the area of software-based distributed shared memory.

DSM System	Notes
Emerald (Jul et al. 1988)	Amongst the first approaches to object-based DSM systems.
IVY (Li 1986; Li and Hudak 1989)	The first approach to implementing an all-in-software distributed shared memory system on commodity hardware workstations. The virtual paging mechanism of the operating system was used to capture memory operations on remote pages.
Linda (Ahuja et al. 1986)	Linda provides a shared tuple space abstraction on distributed memory machines.

DSM System	Notes
Midway (Bershad and Zekauskas 1991; Bershad et al. 1993)	Refer to Section 4.4.2
Munin (Bennett et al. 1990b; Carter 1995; Carter et al. 1991)	One of the first systems to support multiple consistency protocols. The appropriate protocol to be used for a shared variable was indicated by the programmer.
Orca (Bal et al. 1992)	An example of a programming language incorporating the shared memory abstraction. The compiler of the language produced code that could run on distributed memory architectures.
Shasta (Scales et al. 1996)	A system that does not require changes to parallel applications to run on distributed memory architectures. It also features fine granularity of data sharing.
Treadmarks (Keleher et al. 1994)	Perhaps the most influential of the DSM systems. Release and lazy release consistency were designed and implemented for Treadmarks. It was used in the research of various DSM related issues. Currently, it is the only commercially successful DSM system.

Table 4-1: A synopsis of the most influential all-in-software DSM systems

4.4.2. Midway

Midway is a distributed shared memory system that uses data structures of variable length as its sharing units (Bershad and Zekauskas 1991; Bershad et al. 1993). One of the main objectives of Midway is to allow existing and new parallel applications for multiprocessors to run efficiently on multicomputer architectures. Midway requires that existing applications be slightly modified in order for the memory system to work correctly. It is implemented as a run-time library and it requires compiler and programming language support. To that extent, a number of additional keywords are introduced to the C programming language and a modified C compiler is made available.

Parallel applications perceive the shared memory as an unstructured linear space, as it is the case with page-based systems. However, Midway is not considered a page-based DSM system because the sharing unit is the data structure and not the page (see also Section 4.2.1, page 65).

An important characteristic of the Midway DSM system is its primary memory consistency model, entry consistency. Modifications to the C programming language and compiler were introduced to accommodate the requirements imposed to applications by the entry consistency model, as discussed in Section 4.3.2 (page 71). However, Midway also offers applications the option of using processor or release consistency.

An exhaustive investigation of the Midway implementation details is beyond the scope of this thesis. Nevertheless, the designers of Midway have made some interesting choices that are worth noting and contrasted to the design decisions for the NIP object-based DSM system (Section 4.5).

The Implication of Grouping the Shared Variables

In the Midway implementation of entry consistency, programmers are responsible for the identification of the shared variables in an application and the association of those variables with synchronisation constructs. There is a one-to-many relationship between the synchronisation constructs and the shared variables. If a shared variable were associated with more than one synchronisation construct, the correct execution of parallel applications would have been negatively affected. This is true for all concurrent environments with shared resources.

Nevertheless, parallelism may be restricted due to the grouping of variables around a synchronisation construct. For example, concurrent access to the elements of a vector would not be possible if all the elements were associated with the same lock. The solution would be to associate each element with a different lock. However, in Midway there is a run-time overhead with the process of associating a variable with a synchronisation construct. Moreover, if access to the whole vector was required, a great number of acquire and release operations would have been required, resulting in higher run-time costs.

The advantage of having a number of variables associated with one lock is the minimum time a processing node is required to spend in a critical section. Unlike lazy release consistency, before a critical section is entered (i.e., an acquire operation on a lock has taken place) all the required protected variables are updated because they are associated with the lock that guards that section. As a result, no other synchronisation operations are required to take place until the end of that critical section (i.e., a release operation on the lock).

Programmers are further burdened with the task of finding the correct balance between the degree of parallelism that can be exploited in their application and the run-time costs incurred (i.e., association of shared variables to synchronisation constructs and communication between processing nodes during an acquire operation). Programmers have to decide how large the groups of shared variables should be in order to minimise the synchronisation operations between processing nodes while maintaining the high degree of parallelism in their application. Of course, it is a complicated, troublesome, and error-prone task.

Distributed Synchronisation Management

Midway manages the synchronisation constructs (i.e., lock and barriers) separately from the memory caches where replicated copies of the data are maintained. A distributed

queuing algorithm is used to manage the lock operations while a centralised algorithm is used for the management of barriers.

As entry consistency defines, there are two modes of lock access: exclusive and non-exclusive. Every lock in Midway is considered to have a processing node acting as its owner. The ownership of a lock moves from one processing node to another. The last processing node that successfully acquired the lock in exclusive mode is considered its owner. There may be many processing nodes simultaneously holding a lock in non-exclusive mode but only one exclusive mode access is allowed at any time. A lock cannot be accessed in exclusive and non-exclusive mode by different processing nodes at the same time.

The distributed queuing algorithm implemented in Midway for the management of locks is similar to those described in (Forin et al. 1988) and (Lee and Ramachandran 1990). The detailed description of the algorithm is beyond the scope of this thesis. The Midway implementation of the algorithm is comprehensively explored in (Bershad and Zekauskas 1991).

Every processing node is required to maintain information about all the locks defined in an application. Amongst that information, there is a 'best guess' entry about the owner of a lock. When an acquire operation on a lock takes place and if the processing node is not its current owner, a request is sent to the 'best guess.' If the request is for a non-exclusive mode access and a processing node that has already been granted that kind of access for that particular lock is found, the request is granted. Otherwise, the request is forwarded to another 'best guess.' Every processing node maintains an invalidation set with the nodes to which it has granted non-exclusive access. If a request for exclusive access is submitted and after the owner of the lock has been located, all the current non-exclusive access holders must be invalidated first. The current holders are located through the invalidation sets.

The algorithm may produce unnecessary communication traffic, especially during the invalidation process. Moreover, in a large-scale system, a great number of messages may be required until the owner of a lock is located through the 'best guess' entries.

Distributed Cache Management

Midway does not update the state of all the shared variables when a critical section is entered. Instead, only those cached variables whose state has been modified since the last time the critical section was entered are synchronised. In this fashion, unnecessary communication costs are avoided. Midway employs a timestamp-based protocol in order

to identify the inconsistent shared variables since the last synchronisation operation. The protocol is based on the 'happens-before' relationship defined by Lamport in (Lamport 1978).

A timestamp is associated to every shared variable. When an acquire request for a lock is sent from a processing node, the timestamps of the associated variables are piggybacked on the message. The timestamps are compared with those at the node that is granting the lock and the states of the modified variables are sent with the reply.

For the cache management algorithm to function, compiler assistance is necessary. Indeed, the Midway modified C compiler generates code that updates the timestamp of shared variables every time their state is altered by a write memory operation. Evidently, it is a significant run-time cost.

Overlapping of Communication and Computation

A characteristic of entry consistency that is exploited in the Midway DSM is to treat operations on synchronisation constructs separately from each other. A comparison between the definitions of release consistency (Section 4.3.1, page 70) and entry consistency (Section 4.3.2, page 71) reveals that in the former ordering is imposed with respect to processing nodes, while with the latter the ordering is observed with respect to processes (or, threads). The significance of this dissimilarity is the ability of entry consistency implementations to overlap communication and computation.

In release consistency, a processor must be blocked until the state of all the shared variables in the DSM are synchronised because no concurrency is allowed between critical sections guarded by different locks. In Midway, the job scheduler may allow a different critical section to execute in a separate process (or, thread) while another one is waiting for its shared variables to be updated.

4.5. NIPDSM Design Considerations

This section introduces the all-in-software, object-based NIP Distributed Shared Memory system, or NIPDSM (Parastatidis and Watson 1999a; Parastatidis and Watson 1999b). The design choices for NIPDSM are considered and contrasted against the alternatives that were discussed in the previous sections.

4.5.1. Design Requirements

In Section 2.8, the NIP programming model was introduced as a new methodology of parallel application development. The NIP programming model assumes that the memory

of the targeted abstract architecture is organised as a collection of objects whose state is only accessible through their methods. The NIP execution model (Section 2.9) is the targeted abstract architecture for the NIP programming model and as such, it must incorporate a memory system with the required support for objects. Additionally, given that the NIP execution model presents a shared memory abstraction on distributed memory, shared memory, or hybrid hardware parallel system architectures, its memory system should be implemented accordingly.

The NIP distributed shared memory system (NIPDSM) was designed and implemented to provide a memory abstraction that satisfied the above requirements. As a result, NIPDSM offers an object view of the memory that is shared amongst the tasks in a parallel system. Parallel tasks may call methods on objects as if those objects were stored locally. NIPDSM manages the movement and replication of objects around a parallel system and guarantees the consistency of their state. The alternative of moving parallel tasks to the location of an object was ruled out as it reduces the degree of parallelism that can be exploited. Replication improves the performance of parallel applications by allowing multiple tasks to access the state of an object concurrently. This would have not been possible if tasks were moved to the location of the object, where they would probably had to be executed in a sequential manner, depending on the availability of computational resources.

4.5.2. Design Choices

Structure and Sharing Unit

As already mentioned, NIPDSM structures the memory as a collection of objects. Unlike other object-based DSM systems, NIPDSM may have two types of sharing units depending on whether the caching techniques, which will be discussed shortly (Sections 4.7 and 4.8), are enabled. The sharing unit may be an object or a cache block. A cache block contains a group of objects that are transferred together in order to reduce memory access delays.

The object-based memory layout was preferred over the page-based approach. The decision was not only influenced by the requirement of the NIP programming model for an object memory. This thesis regards object-based DSM systems as more suitable for the emerging field of object-oriented, parallel computing, a view that is also supported by Hyde and Fleisch (Hyde and Fleisch 1998).

Object-based systems do not suffer from false sharing and they have the potential to better utilise the communications resources because only the necessary data is transferred

between processing nodes. In page-based systems, whole pages are transmitted that include potentially unnecessary data. However, for many applications that exhibit locality in memory access, the extra data found in pages proves to be advantageous. Unfortunately, there has not been an object-based DSM system to date that has managed to benefit from the same memory access behaviour of parallel applications. This thesis uses NIPDSM as a vehicle in the study of issues related with bringing spatial locality to object-based DSM systems together with other new caching techniques (Sections 4.7 and 4.8).

Consistency Model

Amongst the proposed memory consistency models, a variant of entry consistency was chosen as the most suitable for NIPDSM. The reasons for choosing entry consistency and the changes to the original definition are explored in a separate section (Section 4.5.3).

Data Access

Some page-based DSM systems allow concurrent writers to access a single page in an effort to deal with the problem of false sharing. However, extra overhead is added due to the amount of additional computational work that is required at every synchronisation point. The differences between copies of the same page must by calculated and integrated into one.

NIPDSM, as an object-based system, does not suffer from false sharing and, therefore, does not require multiple writers. There may be cases, though, where concurrent write access to different parts of the same object may be advantageous. More investigation is required to evaluate any possible performance benefits from providing such functionality to an object-based DSM system. A suggestion on how this could be achieved is presented in Section 7.3. For the purposes of this thesis, the data access model in NIPDSM uses the multiple readers/single writer approach.

Implementation

The NIP execution model is implemented as a run-time library, the NIP run-time (Chapter 5). No compiler or operating system modification is required. NIPDSM is a component of the NIP run-time and as such, it is fully implemented at user level. It does not use the operating system or special compiler support to trap access to objects or to manage the consistency of their state. The implementation details of NIPDSM will be discussed in Section 4.6.

4.5.3. NIP Entry Consistency

In order to allow more parallelism to be exploited, NIPDSM supports the replication of objects throughout the processing nodes of a parallel system. Due to the requirements of the NIP programming model, the correct execution of a parallel application can only be guaranteed if strict memory semantics are observed when managing the state of the replicated objects. The choice of the strict or sequential consistency memory models for NIPDSM was rejected due to the difficulties associated with their implementation. Nevertheless, even if the difficulties were overcome, the inefficiency of the models due to the associated communication overheads would have made NIPDSM unusable. Instead, a relaxed consistency model that offers strict semantics was deemed necessary.

From the available relaxed consistency models (Section 4.3), entry consistency (Section 4.3.2) appeared to be the best match for the requirements of the object-based NIPDSM. Entry consistency was designed to utilise information about the synchronisation operations (i.e., critical sections and barriers) and their association to shared variables. The difficult and error prone tasks of identifying and defining the synchronisation operations and the process of associating the shared variables with the synchronisation constructs is left to the programmer or the programming language compiler. Furthermore, as it was discussed in Section 4.4.2, the available exploitable degree of parallelism in an application may be reduced depending on the number of shared variables associated with a synchronisation construct. The implementation of entry consistency in NIPDSM does not suffer from the same problems.

Modifications to the Original Model and Advantages

The NIP programming and execution models do not provide the means for application developers to use synchronisation constructs. There is no provision for locks or barriers in the memory system and no means to define critical sections. In the NIP programming model, there exist only functions, objects, and method calls. In the NIP execution model, there exist tasklets and parallel tasks that may call methods on objects. The `waitOrInline()` method of tasklets may be seen as an implicit barrier for all the potentially parallel tasks represented by one tasklet.

The lack of programmer-accessible synchronisation constructs does not necessarily mean that the entry consistency model cannot be used in NIPDSM. However, some changes are required to the original model, which is further relaxed and adapted to the characteristics of the NIP programming model. The NIP entry consistency is the resulting memory consistency model.

NIP entry consistency defines that every object is implicitly associated with a lock. The lock remains private to the object. Explicit acquire and release operations on the lock are made possible through the interface of the object. The method calls on objects defined at the NIP programming model level are implicitly enclosed within lock operations at the NIP execution model level (Code 4-1).

NIP programming model level (programming language)	NIP execution model level (run-time system)

`object.method()`	`object.lockRead()`
	`object.method()`
	`object.unlock()`

Code 4-1: The implicit enclosure of a method call with lock operations

Unlike the original model that was described in Section 4.3.2, in NIP entry consistency there is an one-to-one association between an object and a lock. Before a method is called on an object, the object's private lock is automatically acquired. The execution of the method cannot proceed until the acquire operation is successful. When the method completes, the lock is automatically released.

As in entry consistency, there are two modes of access to an object: read and write. The methods that do not alter the state of an object are called read methods while those that modify the state are known as write methods. Evidently, read and write methods implicitly acquire the object lock in read or write access mode respectively. The implicit acquire and release operations that surround method calls define the critical sections in parallel applications. The degree of available parallelism in an application is not compromised due to objects being locked even though they are not accessed within a critical section, as it may be the case with Midway, where a group of objects can be associated with just one lock.

Finally, there is no run-time cost in associating an object with a lock as there is with the implementation of entry consistency in Midway. In NIPDSM, the lock is part of the object state. An overhead may be incurred, however, when objects are constructed.

Definition

The changes in the original entry consistency as discussed above require that the rules of Section 4.3.2 (page 71) be slightly modified. The NIP entry consistency model requires that implementations of the memory system should always conform to the following:

- A method cannot be called on an object with respect to a task until all updates to that object have been performed with respect to that task.
- The execution of a write method on an object precludes the concurrent execution of any other write or read method on the same object.

- After a write method has been called on an object, any other method may not be called on the same object until the 'owner' node of the object has been notified.

The first rule says that before a method may be called on an object, the state of that object must be brought up-to-date first. The second rule prevents the concurrent execution of any method on an object while a write method on the same object is in progress. Finally, the last rule defines the invalidation process that has to take place before a write method is called on an object. Subsequent method calls will have to contact the 'owner' of the object before their execution may proceed.

Potential Drawbacks

The NIP entry consistency adapts the original model to the requirements of the NIP execution model. Although the changes that are introduced improve upon the original approach, there may exist some negative effects as well.

A disadvantage of the one-to-one implicit association between objects and locks is the extra communication overhead that may be introduced. With the original entry consistency model, the shared variables associated with a lock were all brought up-to-date at the start of a critical section and no additional communication had to take place between processing nodes until the lock was released. With NIP entry consistency, a series of method calls on different objects may result in several state updates taking place. The additional communication overhead could be significant.

NIP programming model (programming language)

```
object1.method()

...

objectN.method()
```

NIP execution model (run-time system)

```
object1.lockRead()
object1.method()
object1.unlock()
...
objectN.lockRead()
objectN.method()
objectN.unlock()
```

Code 4-2: Consecutive method calls may result in several state updates

The problem is related to the memory access patterns of an application. Code 4-2 illustrates the problem by showing the resulting code according to the NIP execution model semantics for consecutive method calls on *N* different objects. As shown, every method call is enclosed within lock and unlock operations. Clearly, the task shown in Code 4-2 requires access to all the objects. However, the object fetching behaviour, as defined by the NIP entry consistency semantics, causes every object to be transferred independently. In systems like Midway, the programmer can provide a solution to the above potential problem by associating objects with one lock. According to the semantics of the NIP execution model, it is left to the run-time system to address the issue. The

caching techniques introduced in NIPDSM succeed in solving the problem without sacrificing any of the benefits that NIP entry consistency introduces (Sections 4.7 and 4.8).

NIP programming model (programming language)

```
object.method1()
object.method2()
object.method3()
```

NIP execution model (run-time system)

```
object.lockWrite()
object.method1()
object.method2()
object.method3()
object.unlock()
```

Code 4-3: Consecutive method calls of the same access mode and on the same object can be grouped together

Another problem with NIP entry consistency may be the added overhead due to repeated lock operations. If a task calls consecutive methods with the same access mode (i.e., read or write) on one object, then unnecessary run-time cost is incurred. However, all the methods could be enclosed around just one pair of acquire and release lock operations at the NIP execution model level (Code 4-3). This optimisation should be an easy task for a programming language compiler when converting the code from the NIP programming model to the NIP execution model.

4.5.4. Coupling the Synchronisation and Cache Management

Unlike page-based DSM systems, the implementation of the NIPDSM system is not dependent on memory page-fault handling mechanisms. Page-based systems often utilise the memory page-fault handling facilities of the underlying operating system and hardware in order to deal with cache management. When access to a memory location in a non-cached page is detected, the page-fault handler arranges for the page to be retrieved from a remote processing node. However, if the page-fault takes place within the boundaries of a critical section, parallelism is reduced because the task executing that critical section will have to suspend until the page is made available. The granularity of the critical section will increase, therefore, blocking other parallel tasks waiting to enter their critical section from proceeding.

As studies have shown, it is computationally expensive to invoke memory page-fault handlers (Anderson et al. 1991; Appel and Li 1991). The hardware is responsible for trapping a page-fault and then the operating system is informed. The currently active task must be context switched so the page-fault handler can be executed. These operations introduce additional run-time costs.

In addition to the performance issues associated with the page-fault mechanism, most DSM systems have to deal with the cost of lock management. Separate locks have to be

used for the management of the shared data and for the internal implementation of the consistency model and the caching techniques.

This thesis considers the approach to identifying memory access operations that was adopted by Midway as a correct step to reducing the run-time costs associated with trapping memory access operations. In Midway, a lock operation on a synchronisation construct is an indication to the memory system that access to shared data is required. As a result, the underlying page-fault handling mechanism is not necessary. All the shared variables are fetched before access to them is requested. Of course, the implementation of such an enhancement is made possible because of the entry consistency memory model, which requires shared variables to be guarded by a synchronisation construct.

NIPDSM follows a similar approach to Midway in detecting memory operations on objects. A method call on an object is the indication to the memory system that access to the state of that object is required. Unlike Midway, though, NIPDSM has finer control over the choice of which objects need to be cached, as explained in Section 4.7.

Midway is forced to deal separately with the management of the synchronisation constructs and caching. Although the entry consistency semantics are sufficient for the detection of access to shared variables via the operations on synchronisation constructs, extra information is required for the identification of the modified data since the last exclusive mode access. A distributed queuing algorithm is implemented to deal with the synchronisation operations (i.e., lock and barriers) while a timestamp protocol, based on Lamport clocks (Lamport 1978), is employed for the cache management. When a lock is acquired, the timestamp protocol is used to determine the shared variables that have been modified since the last exclusive mode access. For the timestamp protocol to function correctly, Midway requires compiler modifications. For every update to a shared variable, the compiler inserts code that alters the timestamp associated to that variable. A similar approach is taken in page-based schemes where pages that contain modified data are marked as 'dirty' for caching purposes. Obviously, managing the cache in the described way introduces run-time overheads.

Owing to the NIP entry consistency semantics, NIPDSM combines the cache and synchronisation management in an attempt to avoid the additional run-time overheads. In NIPDSM, a method call does not only indicate to the memory system that access is required to an object but also it supplies the essential information on whether the state of that object is going to be modified. Consequently, there is no need for timestamp-based protocols. The object is marked as modified at the time its private lock is acquired in write

access mode. The information on whether the state of an object is altered may be supplied by a programming language compiler based on class information or code analysis, or by the application developer.

The coupling of cache and synchronisation management in NIPDSM results in a simpler design and a less demanding and, perhaps, more efficient implementation.

4.6. NIPDSM Implementation

Having explored the design of the NIPDSM system, this section moves to the description of its implementation. The approach taken in implementing the semantics of NIP entry consistency is first considered. Then, the layout and the constructs of NIPDSM are examined.

4.6.1. Node Managers, Read and Write Proxies

The NIP entry consistency semantics do not specify whether the original copy of an object should have a fixed processing node location or whether it may migrate. In systems like Midway and NIPDSM, where the consistency semantics favour extensive replication of data, the choice between a home-based and a migratory model is more ambiguous than it may be on other DSM systems.

A migratory model like the one implemented in Midway, may introduce great communication costs when the owner of a synchronisation construct must be located or when replicas need to be invalidated (Section 4.4.2). As the Midway designers admit, their approach cannot scale to large systems (Bershad and Zekauskas 1991).

Alternatively, a home-based approach does not suffer from the extra communication overhead but it may convert a particular node to a 'hotspot.' Under the home-based model, a processing node may become overloaded when two or more nodes are repeatedly attempting to gain write access to a number of its objects. All the requests will have to be routed via the owner of the objects. However, the home-based model offers faster resolution of object owners and more efficient invalidation process. A more detailed comparison study of the home-based and migratory models under entry consistency semantics is beyond the scope of this thesis. Instead, the reader is referred to (Protic et al. 1998).

NIP Entry Consistency Implementation

For the implementation of NIPDSM, the home-based model was chosen. The node where an object is created is designated as the manager node (or, just manager) for that

particular object. Every object has its own, unique manager but, naturally, a node may be the manager for more than one object. A processing node that stores a replica of an object in its cache becomes a proxy node (or, just proxy) for that object. There may be two types of proxies: read proxies and write proxies.

In order to implement the NIP entry consistency semantics, the following rules have been defined (it must be mentioned that a processing node may include more than one processor and so it may be able to accommodate the simultaneous execution of multiple tasks):

- The manager node of an object allows local tasks to execute read or write methods on the object, provided no proxies exist for that object.
- The manager node of an object allows local tasks to execute read methods on the object, provided no write proxies exist for that object.
- A write proxy of an object allows local tasks to execute read or write methods on that object.
- A read proxy of an object allows local tasks to execute read methods on that object.
- A processing node can be a manager node, a read proxy, or a write proxy for any number of objects but it can never be more than one of them at the same time for a particular object.
- An object has always a manager node, which remains the same throughout the lifetime of that object. At any one time, an object may have none or one write proxy, or alternatively, none or any number of read proxies.
- More than one task on a processing node may execute read methods on the same object at a particular time. At any one time, there can only be one task throughout the parallel system executing a write method on an object.

Mutable and Immutable Objects

Often, applications create objects whose state, once initialised, remains constant until they are destroyed. NIPDSM makes a distinction between mutable and immutable objects. Mutable objects are those whose state may be altered during their lifetime. The objects whose state is not altered after their initialisation are considered as immutable. Knowledge of immutable objects can be exploited by the memory system to provide a more efficient way of dealing with them. Given that a write method is never called on these objects, the

implementation can avoid the extra overhead of dealing with concurrency related safeguards.

4.6.2. Object Representation and NIPDSM Reference

When an object is created in NIPDSM, it is uniquely identified in the memory system by its NIPDSM reference. The NIPDSM reference acts like a virtual memory pointer in traditional memory systems and provides access to the methods of objects. In the current implementation, the NIPDSM reference consists of (a) the unique identifier of the manager node, (b) a page number, (c) an offset.[4] The three fields comprise a unique path to locating the object representation of an object (the NIPDSM memory structure is described in Section 4.6.3).

The object representation is a data structure that maintains vital run-time information about every object in NIPDSM. There is an one-to-one association between objects and their representations at every node. Cached copies of an object have their own representation. The fields comprising the data structure are summarised in Table 4-2.

Virtual memory pointer	The virtual memory pointer to the local copy of the object in the physical memory
Node type	The type of the node for the object: manager node, read proxy, or write proxy
Lock	The current lock on the object: read, write, or free
Number of local read locks	The number of local tasks that have a read lock on the object
List of proxies	The identification numbers of the processing nodes that are proxies for the object
Queue of requests	The lock requests for the object that could not be satisfied immediately
List of associations	The NIPDSM references of the objects with which this object is associated (further explained in Section 4.8.3)
Access history VM pointers	The pointers required for the participation of the object in the locking history at the node (further explained in Section 4.8.4)

Table 4-2: The fields of the object representation data structure

The virtual memory pointer specifies the location of the object state in the physical memory. The value of the pointer remains fixed throughout the lifetime of the object only at the manager node. On a proxy node, the value of the virtual memory pointer is guaranteed to remain the same only while the object is locked. The last two observations may be used to optimise the performance of the memory system and avoid unnecessary NIPDSM reference to virtual memory pointer resolutions. A lock operation on the object

[4] The current implementation uses 32bit NIPDSM references: 1bit indicating whether the referenced object is mutable, 9bits for the manager node id, 14bits for the page number, and 8bits for the offset. As a result, the current implementation of NIPDSM can support up to 2^{22} objects of any size per node and a maximum of 512 nodes.

returns the virtual memory pointer to the application. The virtual memory pointer can be safely used to access the methods of that object until the unlock operation takes place.

The next five fields are used for the implementation of NIP entry consistency according to the rules presented in the previous section. The last two fields in the data structure facilitate the implementation of two of the NIP caching techniques, as described in Sections 4.8.3 and 4.8.4.

4.6.3. NIPDSM Virtual Object Table

The Midway DSM system was required to support legacy applications with minimum modifications. Thus, the memory in Midway is an unstructured, linear space resulting in a less scalable implementation with complicated cache management. The NIP programming and execution models require parallel applications to be developed in a new way without being concerned about legacy code. The memory system can be implemented to be flexible and scalable while providing an object view of the shared memory.

The memory structure in NIPDSM is based on the Virtual Object Table (VOT) (Figure 4-1). The approach taken in the implementation of VOT is analogous to the common practices followed in traditional virtual page memory managers. A series of tables is used to organise the object representations. A NIPDSM reference defines the unique path in the VOT that leads to the representation of an object. Once the representation of an object is located, its state may be accessed via the virtual memory pointer field.

Every processing node in the parallel system maintains its own virtual object table. The NIPDSM VOT is used in the dereference process of the NIPDSM reference. Each of the three NIPDSM reference fields acts as an index to one of the tables maintained by the VOT. The first table contains an entry for every processing node in the parallel system. The contents of the table are virtual memory pointers, which lead to a second table. Likewise, the second table leads via a pointer to a third table, which contains object representations. Following the series of tables, the desired object representation may be reached and the memory system information about that object as well as its state may be accessed.

The VOT is not fully constructed when the NIPDSM starts. The left most table of Figure 4-1 is always present on all the nodes. The second table is also initialised but only for the entries in the first table that represent existing processing nodes. With the intention of conserving the physical memory, the tables of the third column are constructed lazily as new objects are created or existing ones are cached. When a new

object is created on a processing node it is associated with the next available object representation. If no more are available in the existing tables, a new table of object representations will be created (third column in Figure 4-1). In a similar way, when an object is cached on a processing node and the required table is missing, it is created and it is only removed when all of its object representations are no longer needed.

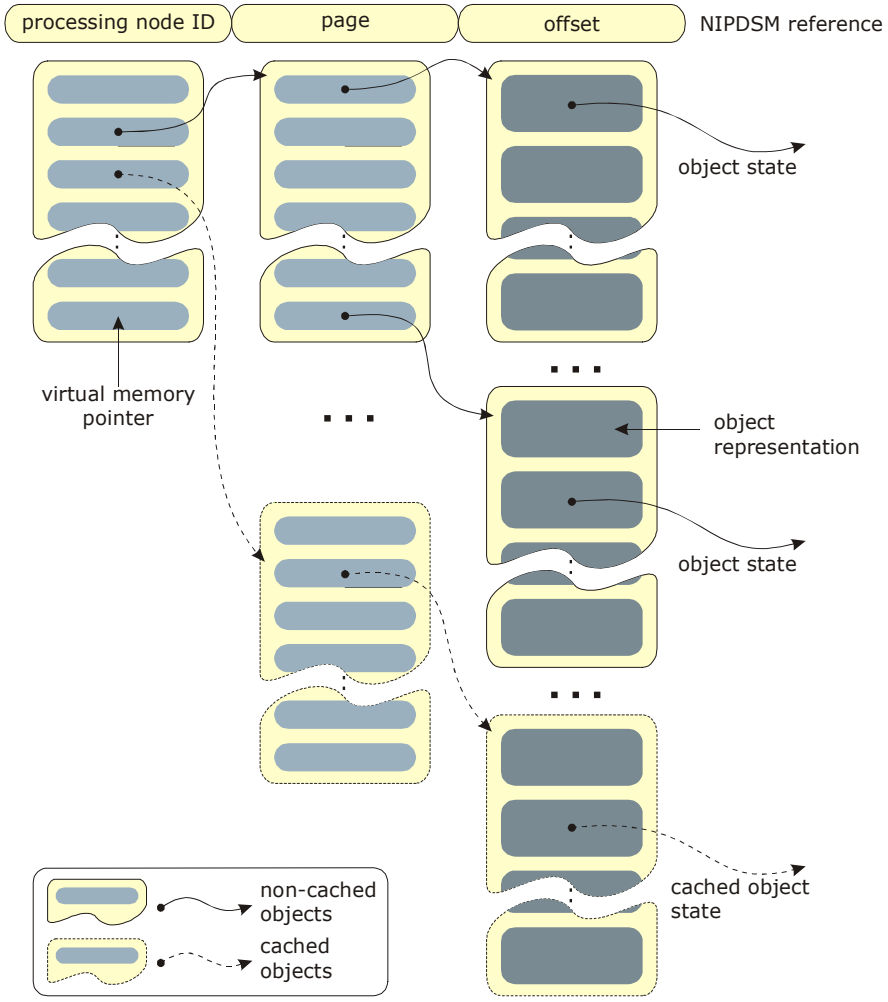


Figure 4-1: The NIPDSM VOT table

The NIPDSM VOT can scale to a great number of processing nodes and objects while making good use of the physical memory on every node. To improve the efficiency in memory access, the different parts of the VOT may be accessed concurrently by more than one task.

4.7. Introducing Caching Techniques in NIPDSM

NIPDSM relieves parallel applications from the burden of defining critical sections and managing memory synchronisation because every method call on an object is implicitly considered as a small critical region. It was implied earlier in the discussion (Section 4.5.3)

that the fine granularity of the critical sections could compromise the efficiency of all the remote object access operations. It was also suggested that NIPDSM incorporates caching techniques that attempt to reduce the associated costs. Indeed, the NIPDSM system has been used as a vehicle for exploring the suitability of new caching techniques for object-based DSM systems. The NIPDSM caching mechanism seeks to introduce new performance benefits for object-based systems. Memory access patterns that depend on locality (*e.g.*, spatial locality and temporal locality) and on dynamic data structures (*e.g.*, tree traversal) have been explored in NIPDSM.

In earlier DSM systems like Midway, it was the developer and/or the programming language compiler that was responsible for dealing with performance issues related to caching by having to explicitly associate objects to synchronisation constructs and carefully define critical sections. In NIPDSM, there is no need for an application to be concerned about the performance costs due to method calls on remote objects or the potential benefits due to memory access patterns.

The NIPDSM caching mechanism is based around the notion of a cache block. A cache block is a group of objects that are transferred together from one processing node to another in order to minimise the number of transfer operations and reduce the negative effects of cache misses. A cache miss occurs when a processing node attempts to call a method on an object that is not stored locally. NIPDSM is responsible for selecting the objects that are included in a cache block. The selection process takes place at run-time and it requires no or minimal assistance from the applications.

Different object selection policies are currently used in NIPDSM, each one targeting a distinct memory access pattern. The methods may be used separately or in combination with each other to fill a cache block. The design concepts behind the new techniques are discussed next but their implementation details are presented later, in Section 4.8.

4.7.1. Temporal Locality

NIP entry consistency is responsible for maintaining the consistency of the replicated objects around a parallel system. The model defines that it is only necessary to invalidate the replicas of an object when a write method is called on that object. The implementation of the NIPDSM system takes advantage of the consistency semantics and maintains a cached copy of an object until it is invalidated. As a result, the memory consistency semantics allow applications to benefit from exhibiting temporal locality in the way they access the objects in the memory.

4.7.2. Spatial Locality

The main disadvantage of object-based over page-based DSM systems has to be the inability to leverage the locality in memory access that many applications exhibit. A study of lazy release consistency and entry consistency concluded that DSM systems based on the latter model also suffers from the lack of support for spatial locality (Adve et al. 1996). In contrast, despite its object-based nature and the use of an entry consistency variant, the caching mechanism in NIPDSM has been designed to benefit applications that exhibit locality in the way they access objects.

Usually, when variables are allocated on an unstructured, linear space, they are placed spatially next to each other. A memory page is likely to contain a number of distinct variables that a task may access. In page-based DSM systems, a page transfer results in many variables being moved together and, thus, if spatial locality in memory access is observed, the number of cache misses is reduced. Of course, the actual number of variables being transferred is determined by the size of the page. A similar approach is implemented in NIPDSM.

Equivalently to the variables in page-based systems, objects in NIPDSM are spatially related. Objects in data structures, like vectors, are allocated spatially adjacent to each other. As it will be shortly described (Section 4.8), in the current implementation of the NIPDSM, two objects are considered adjacent to each other when their objects representations are adjacent in the NIPDSM VOT (Section 4.6.3). Of course, the information about the spatial relationship between objects is kept internally to the NIPDSM system and is not visible to applications. An object that was requested by a processing node is transferred inside a cache block. NIPDSM selects the objects that are spatially related to the requested one and places them inside the same cache block. As a result, the spatially related objects are cached at the remote processing node and they can be accessed immediately.

One may have expected that in order to support spatial locality, NIPDSM would be required to face the problem of false sharing. However, NIPDSM manages to offer the benefits of spatial locality while avoiding the run-time costs of false sharing and without having to use a multiple writers solution. NIPDSM does not include a spatially related object in the cache block unless the same lock access (i.e., read or write) with the requested object is possible. Therefore, if another parallel task has exclusive access to an object, NIPDSM will not try to take that access away until it is necessary to do so.

4.7.3. Dynamic Data Structures and Access Patterns

In a parallel environment, there may be cases where a dynamic data structure is created by a number of parallel tasks. The objects comprising the dynamic data structure may be not spatially related (*e.g.*, the data structure was created at different points in time). In traditional page-based schemes, information about the arrangement of a data structure is not taken into consideration in improving cache performance.

In an object-based DSM system like NIPDSM, the knowledge of the relationship between the objects comprising the data structure can prove beneficial. Indeed, NIPDSM utilises information about the data structures to reduce the number of cache misses. In applications, for example, where a data structure is traversed, it is advantageous to have as many elements of the structure stored in the cache as possible.

For this caching technique to work, NIPDSM requires additional information to be provided about the associations between objects. The tool that is used for the transition from the NIP programming model to the NIP execution model (*e.g.*, an implicitly parallel programming language compiler) may be able to deduce the association between the objects. For example, a node in a double linked list can be associated with its previous and next objects. NIPDSM uses the supplied information when an object, part of the data structure, is accessed. The object is included in the cache block together with other neighbouring—in terms of their location in the data structure—objects. Only if the associated objects can be locked with the same access mode (i.e., read or write) are they also included in the group, in order to avoid false sharing. The cache block is filled with objects through a breath-first traversal of the graph that the object associations create (discussed in Section 4.8.3).

NIPDSM requires applications to provide information about the associations between objects. Help from the compiler or the application programmer is required for the technique to be applied. For example, a compiler could deduce the associations between objects using class related information (Code 4-4).

The correlation between objects need not only be in relation to dynamic data structures. Objects that are logically associated with each other because they are accessed by the same task may form a group, like the objects of the example in Code 4-2 (page 79). Each object is explicitly associated (*e.g.*, by the compiler) with every other object in the group. When one of the objects in the group is requested by a remote processing node, the others are also placed in the cache block and transferred.

NIP programming model level (programming language)

```
class Node
private:
   Node child
public:
   void add(Node&)

treeAddChildren(Node parent)
   ...
   child1 = new Node
   child2 = new Node
   parent.add(child1)
   parent.add(child2)
   ...
```

NIP execution model level (run-time system)

```
treeAddChildren(Node parent)
   ...
   child1 = new Node
   child2 = new Node
   parent.add(child1)
   parent.associate(child1)
   parent.add(child2)
   parent.associate(child2)
   ...
```

Code 4-4: Type information can be used by a compiler to deduce associations between objects

Objects may also be associated together depending on the memory access pattern of an algorithm. For example, there are various ways of accessing the elements of a matrix: row first, column first, diagonally, etc. The objects stored in the matrix could be associated to each other depending on the chosen access pattern.

The caching technique based on associations between objects described above achieves the same outcome as the association of shared variables with a synchronisation construct in the Midway implementation of entry consistency. Unlike Midway, though, the objects are associated with each other rather than to a synchronisation construct. The association of objects with each other allows for more flexibility. In Midway, the whole group needs to be locked before access to one shared variable in a logical group can be granted. Furthermore, a shared variable may participate in only one group while in NIPDSM an object may be part of several logical groups. Finally, Midway cannot support the creation of logical groups based on the access patterns of algorithms.

4.7.4. Recurring Access to Objects

Many applications access the objects in memory in a specific recurring pattern. The objects that are accessed need not be related in any way (i.e., spatially adjacent or part of the same dynamic data structure). When there is no association between the objects, it is difficult to predict which objects should be included in the cache block in an effort to improve performance. If there was a way, though, to observe and record the access behaviour of an application, caching could be improved.

NIPDSM records the object locking operations at every processing node. When an object is requested by a remote processing node, the objects in the access history list following the requested one are also placed in the cache block. Objects that cannot be locked with the same access mode are not included. This caching method benefits

applications that access unrelated objects in a recurring pattern. For example, a task serving a web request may need to access a number of remote objects in order to construct a web page. The objects were associated with each other in a previous request for the same page and, therefore, they can be placed in the same cache block.

4.8. Implementation of the NIPDSM Caching Optimisations

In the final section dedicated to the NIPDSM system, the implementation of the caching techniques is described. The role of the NIPDSM object virtual table and the object representations are explained.

4.8.1. Cache Block

As already mentioned in the previous section describing the caching mechanisms of the NIPDSM system, a cache block acts as the transfer medium of groups of objects between processing nodes. The caching algorithms select the number of objects that are to be transferred together and they place them inside a cache block. Since the objects in the memory may be of arbitrary size, the size of the cache block is used by the caching algorithms as one of the criteria to end the object selection process. If the size of one object happens to exceed the limit imposed by the cache block, then NIPDSM sends the object as is. The object is not split across multiple cache blocks but no additional objects accompany it.

A cache block may not be fully loaded with objects when it is sent from one processing node to another. To avoid groups with a high number of small-sized objects, NIPDSM imposes a limit on how many may be transferred at a time. There is an associated run-time cost with adding an object to a group. If the cache block size is large then the process of adding many small objects may impose a significant run-time overhead. If the maximum number of objects in a group is reached, the caching algorithms stop and the cache block is sent even if it is not full.

The size of the cache block and the maximum number of objects in a group may have a significant impact on the performance of NIPDSM in combination with the memory access patterns of a parallel application. The investigation of the effect that the cache block size in a group may have on efficiency is part of the performance study presented in Chapter 6.

4.8.2. Object Grouping Based on Location

The NIPDSM virtual object table contains the information required to extract locality related relationships between objects. As in page-based schemes consecutive data structures are allocated on neighbouring memory locations, the object representations of successive created objects reside in adjacent locations in the VOT. The location of the representations is used by NIPDSM to minimise object access times for parallel applications that exhibit spatial locality in the way they use the memory.

Figure 4-2 illustrates the formation of a group when an object is requested by another processing node. First, the location of the object representation is located and the locking operation on the object takes place. If the object is successfully locked, NIPDSM iterates through the adjacent representations and attempts to repeat the same locking operation for each one of them. The objects that are successfully locked are included in the cache block and those for which the locking operation was unsuccessful are ignored. NIPDSM continues to add objects in the cache block until one of the following conditions is satisfied:

- The size of the cache block is exceeded.
- The maximum number of objects that may participate in a group is reached.
- There are no more object representations available to be locked.

The technique is more flexible than caching in traditional page-based schemes because it avoids including objects in the cache block that cannot be locked immediately. In this way, NIPDSM avoids the problem of false sharing and the associated inefficiencies.

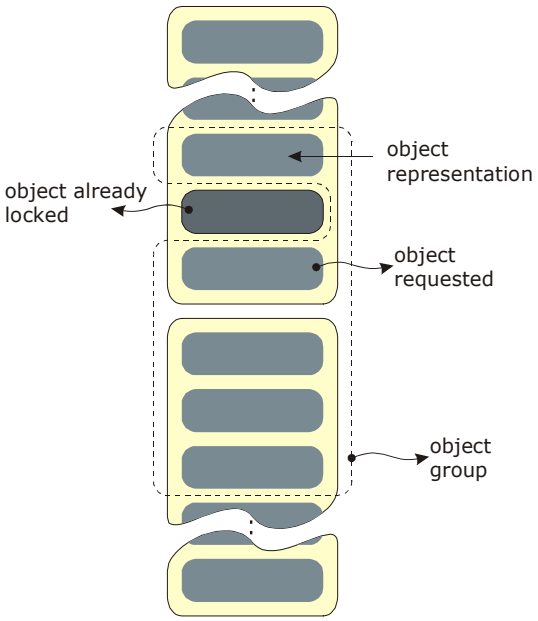


Figure 4-2: Caching based on spatial locality information

4.8.3. Object Grouping Based on Associations

In traditional DSM systems, caching may be ineffective if dynamic data structures are used by applications (*e.g.*, lists, trees, etc.). Usually the elements of such data structures are spread throughout the memory, rendering the caching techniques inefficient. Even in an object-based DSM system like the NIPDSM, access to dynamic data structures may introduce significant delays due to the lack of an efficient caching mechanism. However, the relationship between the elements of a data structure can be valuable if exploited correctly, resulting in the reduction of memory access times.

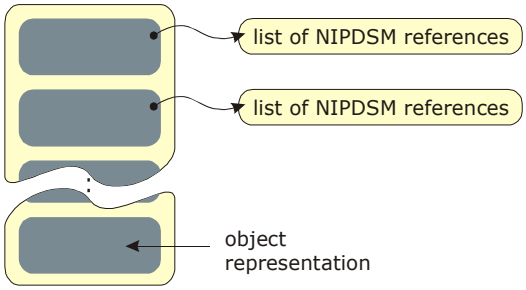


Figure 4-3: Object representations and their list of NIPDSM references of the associated objects

Each object representation in the VOT maintains a list of the NIPDSM references of its associated objects (Figure 4-4). The NIPDSM reference of an object may appear in many association lists. The list of NIPDSM references is maintained on per object basis and can be altered at run-time as dynamic data structures are modified.

If a locking request is submitted for an object, the list of its associated NIPDSM references is traversed and an attempt is made to apply the same locking operation on the corresponding objects. As with the previous caching scheme, the objects that are successfully locked are added to the cache block while those that cannot be locked immediately are ignored.

NIPDSM is not confined in attempting to include into the cache block only the objects that are directly associated with the requested one. Instead, a graph of associations can be visualised with the requested object as the root. NIPDSM iterates through the graph of object associations in a breath-first manner. In some cases, problems may arise from the manner in which the graph is traversed. If a graph-like data structure is traversed depth-first, then the selection of objects included in the cache block may not be optimal. An illustration of the problem and its impact on performance is presented in Section 6.5.2. The problem is not observed when graph-like data structures are traversed breath-first or when list-like data structures are accessed in an iterative manner.

Objects are added into the cache block until one of the following conditions is satisfied:

- The size of the cache block is exceeded.
- The maximum number of objects that may participate in a group is reached.
- There are no more object references in the list to be traversed.

4.8.4. Object Grouping Based on Access History

Applications have to continually incur the same memory access delays when they, repeatedly and in the same order, access a number of objects that are otherwise unrelated (i.e., spatially, temporally, or based on associations). This thesis investigates if the caching mechanism of a DSM system could be improved if the recurring memory access pattern was observed and recorded.

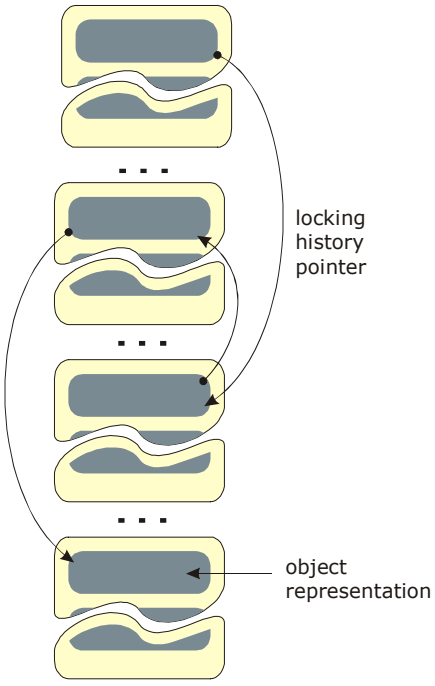


Figure 4-4: Caching based on locking history information

In NIPDSM, when an object is locked, its object representation is automatically added to a locking history list, as in the example of Figure 4-4. Two lists are maintained by the NIPDSM system: one for read and another for write lock operations. If an object is locked by a remote processing node, the locking history list is traversed, starting from the object representation of the requested object, and all of the objects that can be locked with the same access mode are included in the cache block.

There are no additional memory requirements for the two locking history lists to be maintained. The representation of an object includes two virtual memory pointers (Table 4-2, page 83) that point to the representation of the next object in the locking history list.

The caching algorithm avoids the inclusion of more than one copy of the same object in the cache block when a cycle in the locking history list exists. Objects are added to the cache block until one of the following is satisfied:

- The size of the cache block is exceeded.
- The maximum number of objects that may participate in a group is reached.
- The end of the access history list is reached.
- The representation of an object that has already been included in the cache block is reached, in the case of a cyclic access history list.

The locking access history may not capture well all types of recurring patterns. In some cases, for example, if a pattern involves a locking operation on the same object more than once, locking history information is lost. Figure 4-5 presents three different recurring locking patterns that illustrate the behaviour of the caching technique. The object representations of four objects are involved. A number indicates a locking operation on the corresponding object.

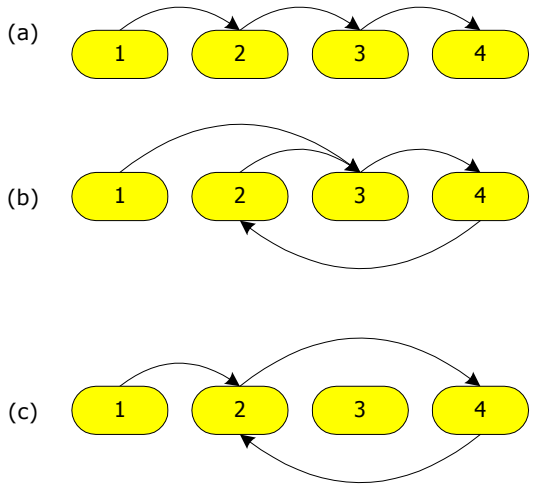


Figure 4-5: Examples of the locking history list

A recurring pattern of the form `1234` results in the locking history list of Figure 4-5a. A locking operation on object `1` will then cause objects `2`, `3`, and `4` to be included in the cache block (if the same locking operation can be applied on them and the conditions for the inclusion of objects in the cache block are satisfied). An access pattern of the form `1234234` also results in all the objects being included in the cache block, even though a cycle exists in the list (Figure 4-5b). However, an access pattern of the form `123424` will only result in three out of four objects to be cached together when a request for the first

object is made because the history list is altered and information is lost after the second time object 2 is locked (Figure 4-5c).

4.9. Discussion

In this chapter, the issues related to the design and the implementation of distributed shared memory systems were studied. The unique characteristics of the NIPDSM were thoroughly explored. In this section, a final discussion and a synopsis of the major claims are presented.

4.9.1. Why Objects

A DSM system provides a convenient and natural way to perceiving the memory of a multicomputer and eases the task of parallel programming. Data movement and communication between processing nodes are no longer a burden to application developers. When compared to the use of message-passing interfaces in programming multicomputers, DSM is more flexible and less error-prone (Lu et al. 1997).

Chapter 2 presented the NIP programming and execution models for the development and execution of parallel applications. Since the NIPDSM implements the memory semantics of the NIP execution model, it is object-based. The NIP programming and execution models aside, this thesis still considers object-based DSM systems more suitable for the future, large-scale multicomputer architectures.

Object-based DSM systems can make better use of the communication resources when compared to page-based approaches and they do not suffer from the problem of false-sharing. Additionally, object-based DSM systems, like the NIPDSM system, may utilise run-time information about the structure of the memory in order to improve performance.

4.9.2. Consistency Semantics

Previous works on DSM systems have proved that in order to achieve efficient execution of parallel applications on multicomputers the memory consistency semantics need to be relaxed. Amongst the several consistency models proposed in the literature, entry consistency (Bershad and Zekauskas 1991; Bershad et al. 1993) was deemed as the most appropriate for NIPDSM. A variant of the original model was designed and implemented according to the memory semantics of the NIP execution model.

NIP entry consistency (Section 4.5.3, page 77) is based on the observation that parallel applications use synchronisation constructs to impose ordering in memory access. Given

that there is no provision for synchronisation constructs in the NIP programming and execution models, NIP entry consistency defines that objects are implicitly associated with one lock. The one-to-one association between objects and locks allows the NIPDSM implementation to have finer control over memory operations.

The simple design of NIP entry consistency and the wealth of implicit information about the way objects are accessed have lead to the unique coupling of synchronisation and cache management. The result is a more efficient utilisation of the computational and communication resources, as explained in Section 4.5.4 (page 79).

4.9.3. Caching

Even in the early stages in DSM research, it was clear that the information about the memory behaviour of applications could be utilised to enhance the caching mechanisms and ultimately improve the performance. Bennett et al. studied the influence in efficiency of the memory access patterns in parallel applications (Bennett et al. 1990a) and concluded that performance could be improved by better cache management. The NIPDSM system was designed and implemented to provide good cache performance for general memory access patterns.

One of the main problems of object-based DSM systems is their inability to support caching for different memory access patterns. Perhaps one the most popular memory access pattern is spatial locality. However, only applications using a page-based approach benefit from exhibiting locality in the way memory is accessed. NIPDSM brings spatial locality to object-based systems in a manner that manages to avoid the problem of false sharing. Additionally, NIPDSM supports temporal locality via the NIP entry consistency semantics. It also collects run-time information about the implicit locking on objects in an effort to benefit from recurring memory accesses. Finally, associations may be defined amongst objects in order to improve the performance of applications that access dynamic data structures.

4.9.4. Midway and NIPDSM

Many of the design ideas of the NIPDSM system were influenced by Midway. However, NIPDSM differs from Midway in a number of significant ways that were mentioned in the discussion to this point. The following table summarises the differences and, at the same time, lists the unique features of the NIPDSM.

NIPDSM	**Midway**
NIP entry consistency: Object-centric, implicit association of objects and locks	Entry consistency: Based on shared variables that programmers need to explicitly associate to a lock

NIPDSM	**Midway**
Coupling of cache and lock management	Separate cache and lock management
Cache support for spatial locality object access pattern	
Cache support for temporal locality memory object pattern through the use of proxies	Cache support for temporal locality memory access pattern through the use of 'owners' and non-exclusive locks
Cache support for dynamic data structures and general object access patterns	
Cache support for recurring object locking	
Does not suffer from false-sharing	Suffers from false-sharing
Object-based view of the memory	Page-based view of the memory
It can be used by programmers as part of a run-time library	It requires changes to the programming language compiler
It does not require a timestamp protocol to identify changes in the state of objects	It requires a timestamp protocol to identify changes in shared variables

Table 4-3: A synopsis of the unique features of the NIPDSM and the way it compares to Midway

CHAPTER 5

THE NIP RUN-TIME SYSTEM

The NIP run-time library is a user-level implementation of the NIP execution model semantics. The library features four distinct services: the communications, the load balancing, the lazy task creation, and the distributed shared memory.

Two of the services incorporated in the library, namely communications and load balancing, have not been examined in this thesis so far. Up to this point, the emphasis of the discussion has been on the novel features of the NIP lazy task creation and the NIP distributed shared memory system. It is not in the scope of this work to thoroughly investigate issues related to communications and load balancing. However, it has been necessary to provide simple implementations for these two services. A fully functional NIP run-time library is required for the evaluation of the NIP lazy task creation and NIP distributed shared memory techniques.

Issues related to the implementation of the library as a whole and the distinct services are explored in this chapter. The aim of the discussion in the following sections is to give an overview of the design and functionality of the NIP run-time library.

5.1. The NIP Execution Model as a Run-time System

In the previous chapters, the NIP programming and execution models for parallel computing were discussed, the NIP lazy task creation (NIPLTC) was presented, and the object-based NIP distributed shared memory (NIPDSM) system thoroughly examined. NIPLTC and NIPDSM were designed to support the task creation requirements and the memory system needs respectively of an implementation of the NIP execution model.

Referring back to Chapter 2, the parallel computing paradigm was illustrated as a set of three layers (Figure 5-1): the design methodology, programming model, and execution model. It was suggested that the last of the layers, the execution model, embodies the abstraction of a parallel architecture. The semantics of the model may be provided by a run-time library, the operating system, the hardware, or any combination of the three. The computational model is the mathematical approach to describing the behavioural aspects and to analysing the performance characteristics of an application that was created according to a certain programming model and it is running on a specific execution model (the reader is referred back to Chapter 2 for the detailed discussion on the layers of the parallel computing paradigm).

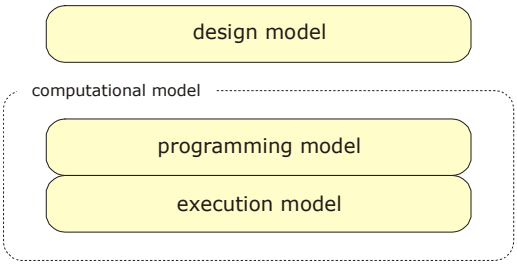


Figure 5-1: The parallel computing paradigm as introduced in Chapter 2

This chapter focuses on the NIP run-time library, whose design and implementation were part of the research work for this thesis. The library was used as a test bed in the evaluation of NIPLTC and NIPDSM and it forms the realisation of the NIP execution model semantics via a set of user-level primitives. The NIP run-time library provides a view of an abstract parallel machine architecture by hiding from the parallel applications and/or implicitly parallel programming language compilers the underlying operating system, any other run-time libraries, and the hardware platform (Figure 5-2). The implementation of NIP does not make any assumptions about the characteristics of the underlying hardware platform. The run-time library can exploit a collection of single-processor and/or multi-processor workstations.

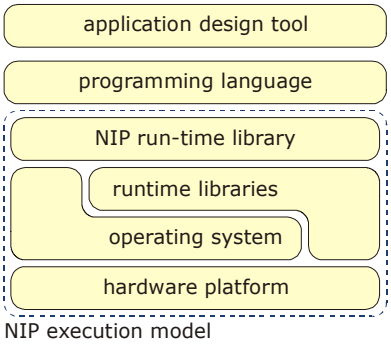


Figure 5-2: The different layers that are hidden by the NIP run-time library, which is an implementation of the NIP execution model semantics

As an implementation of the NIP execution model semantics, the NIP run-time library provides the 'illusion' that there exist an infinitive number of processors and infinite amount of memory structured as a collection of shared objects. The supporting tools for the programming model semantics are responsible for identifying the highest degree of parallelism that may be exploited by the execution environment. That does not suggest that a compiler could not automatically increase the granularity of the some of the identified potentially parallel tasks, which may be too small to ever be executed efficiently in parallel. The NIP run-time library attempts to exploit the identified parallelism in the most efficient manner while considering the availability of the underlying computational resources.

The rest of this chapter focuses on the design of the NIP run-time library (Section 5.2), it provides an overall view of the implementation (Section 5.3), and it describes the four distinct services that are available (Sections 5.4 to 5.7).

5.2. Design

5.2.1. Intended Use

The NIP run-time library is intended to provide the required application execution support for a functional plus objects implicitly parallel programming language (Figure 5-3), like UFO (Sargeant 1993; Sargeant and Kirkham 1994). However, the UFO compiler was not designed and built to utilise the NIP run-time library. At the time this thesis was composed, there did not exist a compiler that could produce code according to the semantics of the NIP execution model. Hence, it was deemed necessary that a C++ interface to the NIP run-time library be made available to developers. The interface is intended to allow developers to explicitly identify the potentially parallel tasks in their parallel applications and to indicate the access operations on objects. The management of

parallelism (i.e., task creation, task execution, task destruction, concurrency control, node-to-node communication, etc.) is left to the NIP run-time library.

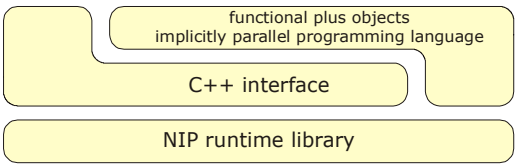


Figure 5-3: Intended use of the NIP run-time library

In the future, a functional plus objects implicitly parallel programming language compiler could be created that utilised the C++ interface (Parastatidis 2000) of the NIP run-time library as the target of a source-to-source translation process. Alternatively, the functionality of the NIP library could be incorporated into the compiler and its code generation routine. Of course, the NIP run-time library could also become the target of a non-functional programming language compiler as long as the requirements imposed by the semantics of the NIP execution model are met.

5.2.2. Overall Structure and the NIP Node

Essential to the design of the NIP run-time library is the notion of a NIP node. During the execution of a parallel application that is built to utilise the NIP run-time library, the underlying parallel system is logically divided into a number of NIP nodes. Each combination of processor(s) and its (their) private, directly accessible memory is considered as a separate NIP node. For example, on a multicomputer that consists of a group of single- and/or multi-processor workstations, each of the workstations is regarded as a NIP node.

The collection of NIP nodes constitutes the execution environment in which parallel applications are hosted. All the NIP nodes cooperate to present parallel applications with the illusion of a single underlying hardware platform featuring physically shared memory and infinite number of processors. There are two types of NIP nodes, the primary and the secondary. There is always one primary NIP node and it is the one used to launch the parallel application. There could be zero to many secondary NIP nodes, which are automatically spawned by the primary NIP node.

In reality, a NIP node is the running executable image of the application. A copy of that image is automatically spawned wherever it is necessary in the parallel platform, as part of the initialisation process of the execution environment.

5.2.3. NIP Node Service

A NIP node is divided into four logical components (Watson 1996) each one of which comprises the implementation of a particular NIP run-time service (Figure 5-4). As their name suggests, the NIP lazy task creation and NIP distributed shared memory services implement the techniques introduced in Chapters 3 and 4 respectively. The design and implementation of these two services are discussed in Sections 5.6 and 5.7. The NIP load balancing service provides two simple algorithms for distributing the available work on a parallel system. The NIP communications service provides a message-passing interface for the exchange of information between multicomputer nodes. Although the research work for this thesis did not involve in-depth investigation of issues related to load balancing and communications, it was necessary that simple implementations of the two services were provided if the NIP run-time library was to be functional. Sections 5.4 and 5.5, introduce the load balancing and communications services and their role in the NIP run-time execution system.

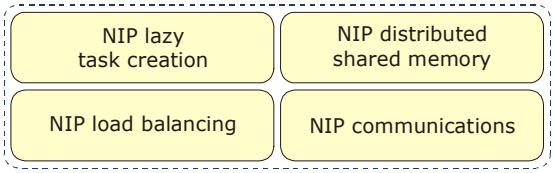


Figure 5-4: NIP run-time library components

5.3. Implementation Overview

5.3.1. Service Task and Workers

As mentioned in the previous section, when a parallel application that utilises the NIP run-time library is launched, a number of NIP nodes are automatically spawned. All the NIP nodes start a service task and a number of worker tasks, or just workers. The service task is of higher priority than the workers are and it deals with all the messages arriving at the NIP node. If there are no available messages to handle, the service task does not consume any computational resources.

The primary NIP node uses one of the workers as the entry point of the application. As more work is made available to the execution environment, more workers are given computation to execute throughout the parallel system. As already discussed in Chapter 3, potentially parallel work is identified via tasklet constructs. One of two work-stealing algorithms distributes the available work across the NIP nodes (Section 5.5).

5.3.2. The Portability Issue

The NIP run-time library has been implemented in the C++ programming language (Stroustrup 1997). Due to the objective of the library, which is to provide a run-time environment for parallel applications according to the semantics of the NIP execution model, one would expect that the implementation be closely tied to the underlying operating system and machine architecture. However, it has been possible to achieve the portability of the library source code across a variety of systems by utilising the ACE object-oriented development framework (Schmidt 1995).

ACE has been ported to a great number of operating systems and it provides, amongst other features, a consistent interface to commonly used system calls without sacrificing performance. Theoretically, the NIP run-time library should be functional on most, if not all, operating systems to which ACE has been ported. However, for the purposes of this thesis, the NIP run-time library was only tested on the WindowsNT, Windodws2000, and Linux operating systems.

5.4. NIP Communications

5.4.1. Design

The NIP communications service is a simple, non-optimised, message-passing interface that allows NIP nodes to exchange messages. It would have been possible to utilise an existing message-passing library, like PVM (Sunderam 1990) or an implementation of the MPI standard (Forum 1994), but there was not one available supporting thread-safe communication primitives and the means to block on multiple active connections waiting for network activity, two design requirements for the NIP communications service.

Due to the nature of the NIP run-time system, it would not have been efficient to use predefined points of synchronisation between NIP nodes where information could be exchanged, similarly to the approach taken by conventional message passing interface systems. For example, a worker executing some computation may require access to an object from the distributed shared memory. For the worker to continue its execution, the object must be first made available locally to the NIP node via the NIPDSM service, if it is not available already. As a result, an access request will be submitted to the appropriate NIP node. It cannot be known before the execution time when such an access request is going to take place. The other NIP node services also require to send or to receive messages in the same unanticipated manner.

On a NIP node, a separate thread could have been spawned for every connection to another NIP node. Each thread would only have to deal with the messages received from one NIP node. However, this approach does not scale well due to the great number of threads required as the number of NIP nodes increases. This is the reason a blocking call on multiple network connections is required. In the current implementation of the NIP communications service, only one thread, the service task, is created on every NIP node to deal with received messages. The thread blocks and it is only resumed to deal with a message that has arrived on one of the established connections with the other NIP nodes.

5.4.2. Communication Between NIP Nodes

Essential to the design of the NIP run-time library and especially the communications service are the notions of the node id and the message. Every NIP node is implicitly associated with a unique node id. The NIP node services exchange information via messages (i.e., objects of type `Message`). The destination address of each message is set to the node id of the receiving NIP node. Broadcast and multicast communication services are also supported by the design but their efficiency depends on the underlying communication protocol used.

Any data that must be transferred from one NIP node to another must be packed inside a message. The message is then given to the NIP communication service, which is responsible for transmitting it to the appropriate NIP node. Of course, the data must be prepared so that it can be transmitted across the channel between the sending and the receiving NIP nodes according to the requirements of the underlying communication protocol.

As already mentioned, there is a high priority service task at each NIP node, which is activated when data arrives from other NIP nodes. The NIP communication service reconstructs the original message from the raw data and routes it to the appropriate NIP node service.

5.4.3. Implementation

The current implementation of the NIP communication service utilises the TCP/IP stack provided by the underlying operating system. A connection-oriented data transfer channel (i.e., a socket-based communication stream) is established between each NIP node. In addition to abstracting the necessary operating system calls for the creation of the communication channels, for the transmission and for the reception of data, ACE also provides a reactor service, which is an event-dispatching mechanism facilitating the

implementation of unanticipated reception of a message. Internally, the reactor utilises the *select(2)* call on UNIX systems or the *WaitForMultipleObjects()* call on Windows platforms.

Unfortunately, the TCP/IP stack is probably not the most efficient communication protocol for parallel computing on networks of workstations. TCP/IP provides services like buffering and error correction that add overhead to the latency of a message transfer operation. A less expensive protocol on a faster interconnection network would probably be better suited for the NIP run-time system (*e.g.*, Layer5 protocol on ATM networks).

An additional run-time overhead in the current implementation is the kernel participation in all the data transfer operations. There exist communication subsystems that could allow the data exchange between two NIP nodes to take place without the involvement of the kernel and therefore offering better performance. The SCI (James 1994) network interface cards and the U-Net (von Eicken et al. 1995) user-level library are examples of solutions to message passing that do not require the involvement of the operating system kernel in communicating data, hence achieving better latency and bandwidth.

Finally, the feature of the NIP communications service that allows NIP node services to use the message abstraction when exchanging data, rather than to deal with operating system communication primitives, introduces yet another overhead. A memory copy operation takes place every time an object is added to a message. The memory used by a message to store objects is expanded dynamically. As a result, the copies of the objects may not be allocated in a continuous part of the memory. For that reason, additional memory copy operations are required to transfer all the objects into a buffer that is maintained by the NIP communications service. The contents of the buffer are transferred through an appropriate operating system call.

Although the implementation of the NIP communication service is by no means optimal, it is required for a fully functional NIP run-time environment.

5.5. NIP Load Balancing

The main objective of the NIP load balancing service is to keep track of the available work on a parallel system. It has to make sure that all the processing nodes are busy but not overloaded. The decision on whether a new task will be created using the NIP lazy task creation technique and then executed in parallel is taken by the NIP load balancing service. Both multiprocessor and multicomputer architectures are supported.

Due to the dynamic nature of the NIP run-time library, a static load balancing service would be insufficient (i.e., the decision on whether a task should be created and/or moved to another processing node is made at compile time). Instead, a dynamic load-balancing algorithm is required. A lot of research work in the area of dynamic load-balancing, or load-sharing, has been published. The reader is referred to (Billard and Pasquale 1997; Eager et al. 1986; Loh et al. 1996; Lüling et al. 1992) as good starting points.

As load balancing is not the subject of this thesis, the NIP load balancing service, like the communications service, was just designed and implemented to support the NIP run-time library and the new techniques it incorporates, namely the NIPLTC and the NIPDSM. Existing load-balancing techniques were adopted for the current incarnation of the NIP run-time library.

5.5.1. Design

The NIP load balancing service features two simple but distinct algorithms that are selected through a command-line switch during the launch of a NIP parallel application. Both algorithms are based on the notion of work stealing. When a NIP node is about to run out of computation to execute, the NIP load balancing service checks to see if there is any work available locally and if not, it requests work from other NIP nodes. For the load balancing algorithms in the NIP run-time library, the notion of 'available work' on a NIP node is equivalent to the 'existence of at least one tasklet.'

One of the load-balancing algorithms is activated when there are no available tasks that could be executed. It is only then that the local tasklet availability queue (Chapter 3) is checked to establish whether a new task could be created. There are no additional load-balancing requirements for multiprocessor architectures. However, on multicomputer systems, if the tasklet availability queue is empty or no tasks can be created from the existing tasklets (i.e., because the `createTask()` calls return `false`) other NIP nodes need be contacted. It is in this way the decision is made on which NIP node to be contacted where the two algorithms that are supported differ.

- The simpler of the two algorithms chooses a random NIP node and sends a 'request-for-work' message to it. If the NIP node receiving the request does not have any tasklets in its tasklet availability queue or no work could be created from the existing ones, a 'request-for-work-denied' message is sent back and a new NIP node is contacted.

- The second algorithm maintains a table with information about all the NIP nodes in the parallel system. The entries in the table indicate whether the corresponding NIP node has at least one tasklet in its tasklet availability queue. In order for the table to represent the most up-to-date information, each NIP node informs the rest about the changes in its tasklet availability queue. Whenever the transition from zero to one available tasklets takes place, an appropriate message is broadcasted to all the NIP nodes. Similarly, when the tasklet availability queue moves from one to zero tasklets, another message is sent.

  When the load-balancing algorithm is activated, the table is searched and the first NIP node to be found with available work is sent a 'request-for-work' message. However, it may be the case that the information maintained in the table is not up-to-date (*e.g.*, a message indicating that the particular NIP node has run out of work is on its way), in which case a 'request-for-work-denied' message is received and the table is updated accordingly.

During the execution of a parallel NIP application, a number of events may cause the activation of the load-balancing algorithm. In all cases, it is the number of available tasks for execution on the NIP node that is first checked and only if more are required is the load-balancing algorithm called. Examples of events that may result in the inspection of the available tasks for execution are:

- A task completes the evaluation of some computation.
- A message arrives from another NIP node indicating the availability of more work.
- A new tasklet is added to the local tasklet availability queue.
- A task needs to wait for an object to be cached or to be unlocked.
- A task has to wait for other tasks to finish before it can proceed (i.e., suspend).

5.5.2. Tasks and the Load of NIP Nodes

The NIP load-balancing service does not feature a task scheduler (or, job scheduler). Instead, the scheduling is left to the underlying operating system as all tasks are mapped to kernel-level threads. Nevertheless, the NIP load-balancing service considers the tasks to be in one of four states: started, running, waiting, and terminated as shown in the task-state diagram of Figure 5-5.

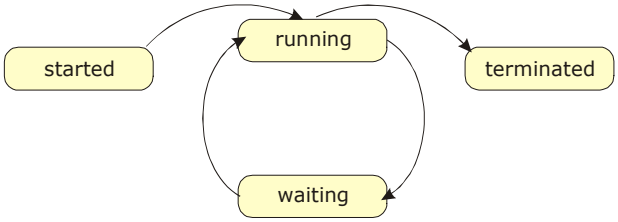


Figure 5-5: The four task states as considered by the NIP load-balancing service

When a task is assigned some computation to evaluate, it is considered to advance from the 'started' state to the 'running' state. When the evaluation of the computation completes, the task moves to the 'terminated' state. In between, a number of reasons may cause the task to be put into the 'waiting' state:

- A call to the `waitOrInline()` method of a tasklet (Section 3.6.1) when there are running parallel tasks created from the same tasklet.
- A lock operation on an object that the NIPDSM does not allow until the object is unlocked.
- A lock operation on an object that it is not cached.

The NIP load-balancing service considers a NIP node busy when the number of tasks in the 'running' state is equal to or greater than the number of locally available processing nodes. The service does not attempt to check the load on the NIP node due to other—unrelated to the NIP application—work that may consume computational resources. Additionally and due to the lack of an integrated task scheduler, there may be cases where a NIP node has more tasks in the 'running' state than available processing nodes. The NIP load-balancing service moves tasks from the 'waiting' to the 'running' state without considering the availability of the computational resources. As a result and if the underlying threads library supports thread pre-emption, then additional run-time cost may be incurred due to context switching as all the tasks in the 'running' state compete for processing power.

5.5.3. Implementation

The current implementation of the NIP load-balancing service utilises kernel-level threads for the execution of the parallel tasks created from tasklets. A user-level threads library that does not support pre-emption could have been utilised to avoid the additional overhead of context switching that may arise in some cases, as mentioned above. User-level threads libraries implement an independent scheduling mechanism to avoid the overhead of kernel-level scheduling. However, many user-level threads library cannot leverage more than one processor on a shared-memory multiprocessor architecture.

Furthermore, operating system calls that block the execution of threads waiting for an event may result in computational resources being wasted, since another thread cannot be scheduled. Lack of pre-emption would have made the implementation of the NIP communications service more complicated and perhaps more inefficient. The service task (Section 5.3.1) depends on pre-emption to deal with arrived messages as soon as possible.

Both the two platforms used for the development and performance evaluation processes of the NIP run-time library, namely Windows2000 and Linux, support thread pre-emption and symmetric multiprocessing. The Linux kernel 2.2.x, however, does not support thread priorities for users without administrative privileges. This results in the service task being assigned the same priority as the workers and, therefore, having to compete for computational resources when a message arrives with any 'running' workers.

Thread Pool

Whenever computational resources become available and a new task is created, either from a local or from a remote tasklet, a thread is required to 'host' the evaluation of the associated computation. Once the evaluation completes, the thread is not needed anymore. In an attempt to avoid the repeated thread creation and destruction operations and their associated run-time costs, the NIP load-balancing service creates a number of threads during the initialisation process of every NIP application. All the threads are created in the 'suspended' state and they are added to a pool that is maintained by the NIP load-balancing service.

When a new task is ready to be executed, a thread is removed from the pool and it is resumed to evaluate the computation associated with the task. When the computation completes, processing resources may become available, hence the number of the tasks required to keep the NIP node busy is checked. If more tasks are required, the load-balancing algorithm is called within the context of the same thread that just completed the evaluation of a previous task. Only after the load-balancing algorithm has finished, is the thread suspended and placed back in the pool.

The thread pool approach reduces the number of thread creation and destruction operations during the execution of an application, therefore resulting in faster execution times. Of course, if during the execution of an application a thread is required while the pool is empty, a new one will be created.

Overlapping Communication and Computation

The current implementation of the NIP load-balancing service does not manage the available processing resources in the most efficient manner in all cases. As described

earlier, if computational resources become available on a NIP node and no local work exists, the load-balancing algorithm will request additional tasks from other NIP nodes. Still, the available processing resources will have to remain idle while the 'request-for-work' message travels to its destination, the recipient of the message processes it, and the response is received. A better solution would be to have a task waiting to take over the processor. This could be achieved by utilising a heuristics-based load balancing algorithm that pushes work to nodes that are likely to run out of work or by attempting to steal work from other nodes before it is needed.

There are cases, however, where the NIP load-balancing service does attempt to overlap the communication with computation. For example, when a task requires access to an object from the NIPDSM and it has to wait until the object is cached, a new task will be created if necessary to take over the just released processing resources. Of course, if there is no local work available, the above problem arises again. The NIPDSM caching techniques attempt to reduce the number of times tasks go to the 'waiting' state because of a cache miss.

Discussion on Efficiency

As it is the case with the NIP communications service, the design and implementation of the NIP load-balancing service is by no means optimal. The service would benefit from a more sophisticated load-balancing algorithm that is aware of the whole execution environment on a NIP node for the better utilisation of the available computational resources. Additionally, an integrated task scheduler would eliminate some of the extra run-time costs due to more than one task being in the 'running' state and competing for processing resources.

Nevertheless, the current NIP load-balancing service is sufficient for supporting the implementation and performance evaluation of the rest of the NIP node services, which consist the main research theme for this thesis: the NIP lazy task creation and NIP distributed shared memory.

5.6. NIP Lazy Task Creation

The purpose of the NIP lazy task creation service is to provide the necessary support for the synonymous technique that was introduced in Chapter 3. The service incorporates the implementation of the tasklet construct and maintains the tasklet availability queue. In addition to supporting application-specific tasklets, the NIP lazy task creation service also

provides a set of predefined tasklets in the form of C++ templates implementing common programming patterns (Parastatidis 2000).

5.6.1. The NIP Tasklet Interface

The NIP lazy task creation service defines the interface of the tasklet construct as a C++ class with pure virtual methods (Section 3.6.1). NIP applications are required to define custom tasklet classes that adhere to the `NIPTasklet` interface. The custom tasklets must inherit the `NIPTasklet` class and provide implementations for the pure virtual methods. The purpose of each of the methods was analysed in Chapter 3 as part of the discussion about the abstract interface of the tasklet construct. The NIP lazy task creation service slightly extends and adjusts that abstract interface.

```
class NIPTasklet
{
public:
   typedef void (*ExecFunction)(NIPTask&);
   NIPTasklet(ExecFunction);
   ~NIPTasklet();
   virtual bool createTask(NIPTask&)  = NULL;
   virtual bool createTask()          = NULL;
   static  void executeTask(NIPTask&);
   virtual void executeTask()         = NULL;
   virtual void returnTask(NIPTask&)  = NULL;
   virtual void waitOrInline()        = NULL;
   void activate();
   void deactivate(bool = true);
protected:
   void beginCriticalSection();
   void endCriticalSection();
   void blockCriticalSection();
   void unblockCriticalSection();
   bool waitForStolenTasks();
};
```

Code 5-1: C++ interface of the `NIPTasklet`

Two additional methods are introduced, `createTask()` and `executeTask()` (i.e., the ones without arguments), to allow for optimisation on shared-memory multiprocessor architectures. When a new task is created from a tasklet because of a request from another NIP node, a `NIPTask` object is automatically instantiated to store essential run-time information. On shared-memory multiprocessor architectures, where only one NIP node exists, the creation of a `NIPTask` object can be avoided. As a result, the run-time overhead associated with the packaging of run-time information into a `NIPTask` object is prevented. The `createTask()` and `executeTask()` are called instead of their `createTask(NIPTask&)` and `executeTask(NIPTask&)` counterparts on shared-memory multiprocessors.

The `executeTask(NIPTask&)` method must be declared as `static` so its virtual memory address can be the same on all the NIP nodes that run the same application executable. It is a necessary adjustment to the interface of the tasklet construct because the state of the particular tasklet instance is not going to be transferred together with the new tasks that are created for execution on remote NIP nodes. According to the C++ standard (Stroustrup 1997), the virtual memory address of a method of an object is calculated in relation to the address of that object. It is therefore necessary to declare the `executeTask(NIPTask&)` method as `static` so that its virtual memory address can be the same on all the NIP nodes running the same application executable. A static method call is shared amongst the instances of the same class but is not allowed direct access to their states. The 'static' qualifier instructs the C++ compiler to give the method global linkage (i.e., it is given a virtual memory address in relation to the application executable and not to the instances of the class). The address of the static method is the required argument for the constructor of the class.

Finally, there is a variation in the semantics of the `createTask()` method in the `NIPTasklet` class and the corresponding method of the abstract type introduced in Chapter 3. The departure from the original semantics was a result of the efficiency considerations in the implementation of the NIP run-time library and the behavioural dissimilarity of some synchronisation constructs under the Windows2000 and the Linux operating systems.

Some custom tasklets may enclose data members that need protection from concurrent accesses. The designer of new tasklets should be aware that the `returnTask()`, the `waitOrInline()`, the `executeTask()`, and both the `createTask()` methods might be called concurrently. Critical sections ought to be defined around any access to the private data member. A new synchronisation construct (*e.g.*, a mutex) could be instantiated and used for the definition of critical sections but it would be more efficient if the lock internal to the `NIPTasklet` could be utilised instead. As already mentioned in Chapter 3, the `beginCriticalSection()` and `endCriticalSection()` methods grant access to the internal lock. The designer of new tasklets should take care and not use the two methods to define critical sections within the two `createTask()` methods because in the current implementation of the NIP lazy task creation service the internal lock has already been acquired by the run-time system. On operating systems that implement re-entrant locks, like Windows2000, it is

not an issue but on systems where the acquisition of the same lock by the same thread is not allowed deadlocks are introduced.

For a similar reason, a Boolean argument has been added to the interface of the `deactivate()` method, which instructs the NIP run-time to remove the tasklet instance from the tasklet availability queue. In Section 3.6.2, the two-level locking process during a task creation operation was described. The lock guarding the tasklet availability queue is already acquired when the `createTask()` method is called. Therefore, it would be troublesome on operating systems that do not support re-entrant operations on synchronisation constructs to call the `deactivate()` method from within any of the two `createTask()` methods without an adjustment. The Boolean argument was introduced to act as an indication to the NIP run-time about the context from which the `deactivate()` method is called.

5.6.2. The Tasklet Availability Queue

The NIP lazy task creation service implements the tasklet availability queue as a double-linked list data structure. The nodes of the list are the tasklets instances allocated on the stack of the tasks that create them. There is no requirement for additional memory to be allocated when a new tasklet instance is added to the list because the two required pointers are part of the state of the tasklet, as described in Chapter 3. Only one variable containing the pointer to the first tasklet in the list and a lock guarding the data structure from concurrent accesses have to be created during the initialisation process of the NIP run-time.

The `activate()` and `deactivate()` methods relate to the addition and removal of the tasklet instance to and from the tasklet availability queue respectively. The two operations involve the acquisition of the lock guarding the tasklet availability queue and the maintenance of the pointers.

5.6.3. The NIP Tasklet Library

In addition to allowing the construction of custom tasklets, the NIP lazy creation service leverages the strength of the C++ templates and offers a selection of parameterised, predefined tasklet classes. The current set of offered templates supports three different patterns of parallelism: function calls, parallel iterative computations, and parallel recursive computations (the three patterns are covered in more detail in Chapter 3). An example of a template class for each of the patterns is presented in Code 5-2.

```
template<class Result, class Argument, Result (*Function)(Argument&)>
class NIPFunctionTasklet

template<class Argument, void (*Function)(int, Argument&)>
class NIPIterativeTasklet

template<class Result, class Argument>
class NIPRecursiveTasklet
```

Code 5-2: C++ template classes for common patterns of parallelism

An instance of the `NIPFunctionTasklet` tasklet class exposes to the NIP run-time a function, the third argument, as a potentially parallel task. A `NIPIterativeTasklet` object is used to represent the parallelism in an iterative computation. The function given to the template as an argument contains the computation associated with each of the iterations. Finally, the `NIPRecursiveTasklet` is used to exploit the parallelism in recursive computations.

The NIP lazy task creation service offers a number of variations of the three template classes presented in Code 5-2. The description of the whole library and the way it is used is beyond the scope of this thesis. Instead, the reader is referred to (Parastatidis 2000) for a thorough explanation of the available tasklets and the way they can be used.

5.7. NIP Distributed Shared Memory

The NIP distributed shared memory service implements the object-based, shared memory abstraction defined by the NIP execution model semantics. The approach to the design and implementation of the service closely follows the discussion on NIPDSM presented in Chapter 4.

5.7.1. Allocation of Objects in the NIPDSM

The NIPDSM service provides a set of template classes and functions in order to facilitate the allocation of C++ objects in the shared memory. The class of any object that is going to be stored in the NIPDSM must inherit from the `NIPShared` class template. Instances of the derived class do not have to be created on the heap of the NIP application. Even if the objects are placed on the stack of a task, they can still be accessed by the rest of the NIP nodes in the parallel system until they go out of scope.

As discussed in Chapter 4, the allocation process involves the possible expansion of the NIPDSM virtual object table, the initialisation of an object representation and the formation of a unique NIPDSM reference, which at the end is returned to the NIP application. The NIPDSM reference is of type `NIPRef<class>` where `class` is the type of the object. `NIPRef` is considered as primitive type and—like `int`, `double`, and

the other primitive types—it cannot be allocated directly in the NIPDSM. An encapsulation class is made available, the `NIPObject<class>`, which allows primitive types and types that are not derived from `NIPShared`, to be stored in the NIPDSM.

5.7.2. Efficiency

The semantics of the NIPDSM system render the use of a page-faulting mechanism unnecessary. Access to the shared objects in the memory is indicated to the NIP run-time through the lock operations on them that all NIP applications are required to issue. The state of an object should not be accessed unless that object is locked first. The separation of the memory access detection mechanism from any operating system service makes the NIPDSM highly portable. Indeed, there was no need to introduce any changes to the implementation of the NIPDSM service when moving between the Windows2000 and the Linux operating systems.

During a locking operation, the availability of the object on the local NIP node as well as its current locking state can be determined. However, in the current implementation, a mutex operation is required whenever an object is locked. As Chapter 6 demonstrates, this introduces a very high cost.

The NIPDSM service implements the locking and caching algorithms that were discussed in Chapter 4.

5.7.3. Object Access

Access to objects in the NIPDSM is made possible through `NIPRef` instances. A `NIPRef` instance acts like a 'smart pointer' (Stroustrup 1997) to objects of a specific class, which must inherit from `NIPShared`. The state and methods of an object is accessed via the C++ arrow operator (`->`). However, the `NIPRef` interface is also enriched with additional operators and methods. The reader is referred to (Parastatidis 2000) for a thorough description of the `NIPRef` interface. Perhaps, the most important of the methods that may be called on `NIPRef` instances are presented in Code 5-3.

The `lockRead()` and `lockWrite()` methods attempt to lock the referenced object. The methods return a virtual memory pointer to the local copy of the state of the object. The pointer can be safely used by the NIP application until the `unlock()` method is called. The use of the virtual memory pointer is an optimisation that was introduced and described in Chapter 4. The optimisation renders the use of the arrow operator (`->`) unnecessary because it always results in a NIPDSM virtual memory table lookup operation, which is computationally more expensive than a virtual memory

pointer de-reference. The arrow operator is still included in the interface, though, for completeness.

```
template<class T>
class NIPRef
{
public:
   T*         lockRead();
   T*         lockWrite();
   void       unlock();
   void       associate(const NIPRef&);
   void       disassociate(const NIPRef&);
   NIPRef<T> operator[](size_t);
   T*         operator->();
};
```

Code 5-3: Part of the `NIPRef` interface

The `associate()` and `disassociate()` methods provide an interface to the management of the associations list, which is maintained on per object basis. The list is used by the NIPDSM object grouping caching optimisation that is based on associations between objects.

Finally, the implementation of the NIPDSM service allows C-style arrays of `NIPShared` objects to be allocated. The subscript operator (`operator[]`) is used when such arrays are accessed.

5.8. Discussion

This chapter introduced the NIP run-time library, a user-level implementation of the NIP execution model semantics. During the design process of the library, four distinct components, or services, were identified. Although no novel techniques were incorporated into the implementation of two of the services—the communications and the load balancing—their inclusion into the library was necessary in order to evaluate the lazy task creation and the distributed shared memory services.

In the previous sections of this chapter, the focus of the discussion was on the implementation details of the NIP run-time services. It has been necessary to concentrate on the implementation issues because the good understanding of the functionality and operational characteristics of the NIP run-time services will be required in the next chapter, where the effect of the NIP lazy task creation and distributed shared memory techniques on the performance of a number of benchmark applications will be thoroughly investigated.

Architecture-related Considerations

Although the portability of the NIP run-time library source code across operating systems was achieved during the implementation process, some idiosyncratic differences between platforms may result in behavioural diversions from what it was specified during the design process.

- The implementation of the kernel threads library in Linux does not support modification of thread-priorities at the non-super user level. As a result, there cannot be a high priority thread dealing with the messages arriving at a NIP node, as the design process for the NIP run-time library specifies. Instead, the thread will have to be of the same priority as the worker-threads that execute computation and compete for processor time. There could be cases where a message is not dealt as soon as it arrives because the thread remains pre-empted until it is given a processor time-slice.

  The latency between a request and a response during the exchange of messages by two NIP nodes may be negatively affected by the thread priority issue on Linux. The NIP lazy task creation technique reduces the effect of the problem by not allowing excessive number of tasks to be created. The result is faster response times because the thread dealing with messages will have fewer threads to compete with for processor time.

  Unlike Linux, the WindowsNT and Windows2000 operating systems do not suffer from the same problem. Therefore, a message arriving at a NIP node will be dealt with immediately.

- The cost of lock operations can be another concern across operating systems. There is a difference in the costs of the operations on synchronisation constructs (*e.g.*, mutex, semaphores, etc.) amongst the two operating systems used for the implementation and evaluation of the NIP run-time library (i.e., Windows2000, Linux). As a result, small parts of the library had to be modified accordingly in order to achieve better performance. The next chapter includes a comparative evaluation of some NIP-related operations between the two platforms.

- Finally, and perhaps more importantly, the load balancing service may be greatly affected by the architectural characteristics of the parallel platform. Parallel architectures, usually with high-speed communications support, may favour load-balancing schemes where tasks are created or moved closer to the location of

the objects they access. In contrast, it may be more efficient on some platforms to move the objects to the computation that operates on them.

The current implementation of the NIP load-balancing service supports only the latter of the two schemes. Data-intensive parallel computations may suffer on platforms with low-latency interconnections, such as Ethernet-based networks of workstations. The NIP distributed shared memory service attempts to reduce the deficiencies, due to tasks accessing objects from different NIP nodes, through its advanced object caching and replication techniques.

CHAPTER 6

PERFORMANCE EVALUATION

The discussion in this chapter focuses on the performance evaluation of the unique features of the NIP run-time environment. The advantages and disadvantages of the techniques studied in the previous chapters are analysed though benchmark applications.

A number of micro-benchmarks are devised to explore the performance characteristics and run-time behaviour of the NIP lazy task creation technique and the NIP object-based distributed shared memory system. Additionally, the performance measurements from the execution of three applications, commonly used in the evaluation of run-time environments for distributed memory architectures are analysed.

The benchmark applications are tested on a physically shared multiprocessor and a distributed memory multicomputer. The results from the performance evaluation demonstrate the effectiveness of the NIP lazy task creation technique and the potential advantages of the NIPDSM caching techniques.

6.1. Introduction

The main objective of this thesis is to explore the characteristics of those unique features that were devised to support the semantics of the NIP execution model. To that extent, the model semantics, the NIP lazy task creation technique, and the object-based NIP distributed shared memory system were studied in the previous chapters. The NIP run-time library, a user-level implementation of the NIP execution model semantics, was also described.

The study continues in this chapter with the performance and behavioural evaluation of the introduced techniques. A number of micro-benchmarks and applications that were built around the NIP run-time library are used as a vehicle for the performance evaluation. Through the analysis of the collected data, the applicability of the introduced NIP lazy task creation technique and NIP distributed shared memory system is examined.

6.1.1. Evaluation Objectives

As mentioned in Chapter 5, the current prototype implementation of the NIP run-time library was built as a test platform for the research purposes of this thesis. The design and the efficient implementation of all the services incorporated in the library were not of a prime research interest to this project. As a result, and given that the performance numbers are collected through the actual execution of parallel applications rather than by the use of simulation, the efficiency of the whole execution environment may not be optimal.

The impact of the load balancing and communications services on performance is indicated where observed. However, the investigation will be primarily concentrating on the efficiency improvements or overheads in the execution of parallel applications due to NIPLTC and NIPDSM. The behavioural characteristics of the parallel applications are also examined.

6.1.2. Real-System Execution vs. Simulation

A combination of a great number of factors may influence the performance characteristics of a parallel application. One has to consider all those attributes of the execution environment that may affect the behaviour and efficiency of an application on a specific platform. In research, a simulation environment is frequently used in the performance evaluation of run-time techniques. For this thesis, it was decided that the quantification

process of the NIP run-time library would take place through the execution of benchmark applications on real parallel platforms rather than via the exercise of simulated execution.

This is because there does not exist a simulation environment that can satisfactorily capture the diversity of performance-related attributes that are found on a parallel platform. Although some simulation environments may consider a combination of the issues, there does not exist one that can deal with all of them (*e.g.*, cost of operating system synchronisation operations, thread-related operations, scheduling, pre-emption, memory access operations, processor cache behaviour, network communication costs, compiler optimisations, etc.).

The evaluation process will not attempt to determine the impact of each possible parameter that may affect the performance of an application execution in a parallel environment, as the one provided by the NIP run-time library. Although of great research interest, it would have been beyond the scope of this thesis to measure every possible detail. Instead, the following sections focus on the NIPLTC and NIPDSM techniques that were explored in the previous chapters.

6.2. Experimental Set-up

As suggested in Chapter 1, this thesis considers parallel platforms based on commodity hardware as emerging architectures for general-purpose, high-performance computing due to their cost/performance ratio potential. Although the NIP execution model semantics were formulated to be platform independent, the experimental process for this research work concentrates on low-cost parallel systems. It should not be assumed, though, that the NIP run-time library could not be used on custom-built, high-performance parallel computer architectures.

6.2.1. Hardware Environment

The main parallel platform used for the execution of the benchmark applications was the Affordable Parallel Platform (APP) of the department of Computing Science, at the University of Newcastle upon Tyne, UK. The APP consists of a collection of commodity hardware-based workstations and an interconnection network. Every workstation of the APP cluster runs an instance of Linux, a freely available operating system. The current incarnation of the APP is made out of eight workstations, each one with a single processor. Each workstation has access to a shared 100Mbit/s Fast Ethernet

interconnection networks. The first column of Table 6-1 presents the main characteristics an APP workstation.

	APP workstation	Linux SMP workstation	Windows2000 workstation	Windows2000 SMP workstation
Processor(s)	PentiumII 233MHz	4 x PentiumIII Xeon 500MHz	PentiumII Mobile 300MHz	2 x PentiumIII 800MHz
Cache	512KB level 2 32KB level 1	512KB level 1	512KB level 2 32KB level 1	32KB level 1
Memory	64MB	512MB	256MB	256MB
Bus speed	66MHz	66MHz	66MHz	133MHz
Swap memory	128MB	128MB	Dynamic	Dynamic
Network	100Mbits shared Fast Ethernet (3Com Boomerg adapters)	N/A	N/A	N/A
Operating system	Redhat 6.2 2.4.0-test7 kernel	Redhat 6.2 2.4.0-test7 kernel	Windows2000 SP1	Windows2000 SP1 multiprocessor mode
Compiler	GCC 2.9.2		Microsoft Compiler 12.00.8804	
C++ wrappers for OS system calls	ACE 5.1.3			

Table 6-1: Profiles of the hardware platforms used for the experiments

The APP cluster is a distributed-memory parallel architecture. The NIP run-time library allows applications to perceive the cluster as a parallel machine with physically shared memory. A shared-memory multiprocessor workstation is also used for the evaluation of the NIP run-time library. The main characteristics of the platform are presented in the second column of Table 6-1.

Finally, two workstations running an instance of the Windows2000 operating system were also used in the evaluation of the NIP primitive operations. The characteristics of the workstations are presented in the third and fourth columns of Table 6-1.

6.2.2. Software Environment

The Linux operating system in the form of the RedHat 6.2 distribution was installed on all the workstations. The 2.4.0-test7 version of the Linux kernel was used during the evaluation process. The support for symmetric multiprocessing kernel option was enabled only on the multiprocessor workstation. For the comparative study of the NIP run-time costs that is presented in the next section, workstations with the Windows2000 operating system were also used.

The NIP run-time library was built with version 2.95.2 of the GCC C++ compiler under Linux. The `-O3 -no-exceptions -no-rtti` compiler options were enabled. Under Windows2000, the Microsoft C/C++ compiler version 12.00.8804 (Visual Studio

SP4) was used with all the optimisations enabled and the exceptions and run-time typing information disabled.

Finally, and as mentioned in the previous chapter, the ACE object-oriented development framework (Schmidt 1995) was used for the implementation of the NIP run-time library. The 5.1.3 version was installed on all the workstations. Table 6-1 summarises the configuration of all the workstations.

6.3. Cost of Primitive Operations

With the intention of determining the efficiency of NIP primitive operations, this section presents a set of small experiments measuring run-time costs. The overhead of the primitive operations, in terms of time and processor cycles, are established for four different processors, two of them running Linux and two running Windows2000. The results from the measurement of the NIP primitive operations are compared to the run-time cost of the equivalent operating system calls.

The cost of a primitive operation is calculated as the average from the repeated execution of the same operation. The frequency of each processor is used to calculate the corresponding cost in processor cycles. Due to the imprecise nature of the timing operations, the presented costs can only be seen as approximations. The small deviations in the results may be attributed to a great number of factors, such as the lack of accurate means for timing, different memory access times, processor cache speed, small kernel differences (i.e., uni-processor version vs. symmetric multiprocessor version), etc. Nevertheless, the results are still of interest to this study because the potential benefits and drawbacks of using the NIP run-time library can be determined.

6.3.1. Operating System Primitive Operations

From the analysis of the measurements presented in Table 6-2 and despite the differences between the processors used, it is evident that the launch of a separate thread of control to execute the same empty function is much more expensive on Windows2000 than it is on Linux. The acquire and release operations on a mutex are faster on Windows2000 but the creation and destruction operations of a mutex are slower. Finally, the creation and destruction of a conditional variable are significantly slower on Windows2000 because ACE has to simulate, using native OS primitives calls, the behaviour of that particular synchronisation construct according to the POSIX semantics (IEEE 1996).

	PentiumII 233MHz (Linux)		PentiumIII Xeon 500MHz (Linux)		PentiumII Mobile 300MHz (Windows)		PentiumIII 800MHz (Windows)	
	usecs	cycles	usecs	cycles	usecs	cycles	usecs	cycles
Function call	0.009	2	0.004	2	0.005	2	0.002	1
OS thread spawn/join	336.930	78505	160.299	80149	654.157	196247	343.334	274667
Mutex acquire/release	0.751	175	0.347	174	0.119	36	0.084	67
Mutex creation/destruction	0.461	107	0.215	108	1.833	550	0.723	578
Condition variable creation/destruction	0.439	102	0.204	102	31.178	9353	15.959	12767
Virtual memory C++ object construction/destruction	2.404	560	1.141	570	1.537	461	0.380	304
Assignment to a data member of a C++ object	0.017	4	0.009	5	0.016	5	0.004	3

Table 6-2: The elapsed time in *usecs* and the corresponding cost in processor cycles of some operating system primitive operations

The costs of creating a C++ object in virtual memory and performing an assignment operation are measured. The construction of an object is found to be slightly faster under Windows2000 over Linux. As would have been expected, an assignment operation on an object data member does not introduce any significant costs to the execution of an application.

6.3.2. NIP Run-time Primitive Operations

There are two possible ways a potentially parallel computation may be executed: inline or as a new parallel task. When the logical degree of parallelism in an application is high, a great number of tasklets may exist. The cost of the tasklet creation and destruction operations can play a significant role in the performance of an application. For that reason, the measurements presented in Table 6-3 were taken into consideration in the implementation of the NIP run-time library.

As described in Chapters 3 and 4, every tasklet and NIPDSM object is implicitly associated with a private lock. In the current implementation of the NIP run-time library, the lock consists of a mutex and a conditional variable. Due to the expensive conditional variable operations on Windows2000, a shared pool of synchronisation constructs is used on every NIP node. Whenever a new tasklet or NIPDSM object is created, a pair of a mutex and a conditional variable is retrieved from the pool. Only the relative smaller cost of the acquire and release operations on the mutex that guards the pool is incurred while the instantiation overhead of a conditional variable is avoided. On Linux, where the acquire and release operations on a mutex are cheaper, there is no need to implement a pool for the tasklet construct. However, a pool of mutex and conditional variables is used

for the NIPDSM objects, as is under Windows2000, in order to avoid exhaustion of resources in case of a large number of objects. Pairs of mutex and conditional variables can be shared between different objects. This consequence is that simultaneous access to the object representation of two different NIPDSM objects sharing the same pair is prevented.

	PentiumII 233MHz (Linux)		PentiumIII Xeon 500MHz (Linux)		PentiumII Mobile 300MHz (Windows)		PentiumIII 800MHz (Windows)	
	usecs	cycles	usecs	cycles	usecs	cycles	usecs	Cycles
Tasklet creation/destruction	4.109	957	1.899	950	0.665	200	0.360	288
Tasklet creation/destruction and function inline	11.486	2676	5.344	2672	1.376	413	0.670	536
Function inline	0.928	216	0.428	214	0.121	36	0.089	71
Tasklet creation/destruction and parallel task creation	59.328	13824	45.432	22716	26.733	8020	13.907	11126
Cost for executing a task from a tasklet on a remote node (fast Ethernet)	614.200	143109	N/A		N/A		N/A	
NIPDSM C++ object construction/destruction	9.850	2295	4.675	2337	10.744	3223	3.797	3038
Assignment to a NIPDSM C++ mutable object (lock and unlock)	2.134	497	0.989	495	0.746	224	0.290	232
Access to a NIPDSM C++ immutable object (lock and unlock)	0.366	85	0.170	85	0.229	69	0.057	46

Table 6-3: The elapsed time in *usecs* and the corresponding cost in processor cycles of some NIP primitive operations

Table 6-3 presents the costs of the NIP run-time primitive operations. The smaller costs of the tasklet creation/destruction and inlining operations under Windows2000 are due to the smaller costs of the mutex acquire/release operations when compared to the equivalent costs under Linux.

As described in the previous chapter, the NIP run-time library uses thread-pooling to optimise the use of resources. As a result, the lazy creation of a parallel task is faster than eager creation. The former only requires that a thread be taken from the thread pool and resumed to execute the new parallel task while the latter requires a new thread to be created. Of course, the overhead of creating the thread pool is incurred during the initialisation of the execution environment. The thread creation cost only need be incurred once and not every time a new parallel task is created. The great overhead of the thread creation and thread synchronisation operations under Windows2000 and the small cost of mutex acquire and release operations result in high speedup for task inlining over eager task creation, as shown in Figure 6-1. Finally, lazy task creation over the shared fast

Ethernet network of the APP is significantly expensive, 1.8 times slower than eager task creation on a local machine because of the network activity that needs to take place.

Based on the results presented in Table 6-2 and Table 6-3, the graph of Figure 6-1 shows the advantages and disadvantages of using the NIP run-time library. Task inlining is significantly faster than eager and lazy task creation especially on Windows2000 where the thread creation costs are higher and the mutex operations are cheaper. Figure 6-1 suggests that applications exhibiting a higher degree of parallelism than what is available by the hardware architecture will perform much better when tasklets are used to identify parallel tasks because of the benefits of inlining.

In contrast to the benefits of the tasklet related operations, the NIPDSM operations introduce significant run-time overhead. A data member assignment operation on a NIPDSM object is considerably slower when compared to the equivalent operation on a virtual memory object. This is attributed to the high run-time cost that the NIPDSM lock and unlock operations introduce. According to the NIP execution model semantics, every operation on an object must be included within a pair of lock and unlock operations (Chapters 2 and 4). As discussed in Chapter 4, though, memory access can be improved by including consecutive calls on the same object within just one pair of lock and lock operations. It should also be noted that the NIPDSM objects might be safely accessed in a concurrent manner. In contrast, the virtual memory objects are not guarded from any form of concurrent access.

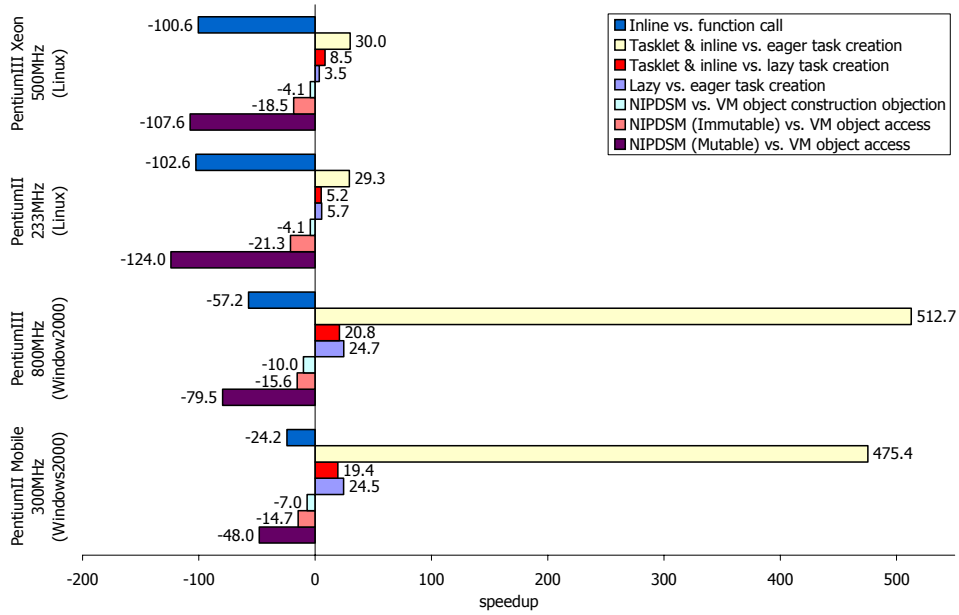


Figure 6-1: The speedup and slowdown of NIP primitive operations over the corresponding operating system operations on four different configurations

The overhead of the NIPDSM operations is due to a number of factors. A lock operation on a NIPDSM object includes:

- **NIPDSM dereference.** The NIPDSM object representation structure of the object to be locked must be located. This involves operations on the NIPDSM reference to identify the correct path to the object's representation in the NIPDSM virtual object table (Chapter 4).
- **Mutex acquire/release operations.** Access to the NIPDSM object representation is critical and needs to be protected. Two concurrent tasks on the same NIP node should not be allowed to access simultaneously the same NIPDSM object representation. This does not imply, of course, that two tasks cannot have a read lock on the same object at the same time.
- **C++ STL operations.** The current implementation of the NIPDSM uses the `list<T>`, `set<T>` C++ Standard Template Library (Stroustrup 1997) classes to maintain necessary information about the state of the object (*e.g.*, the nodeIDs of the proxies, the queue of lock requests, the NIPDSM references of the associated objects, etc.). A NIPDSM lock operation may involve calls to instances of one of those classes, which are probably computationally more expensive than a custom solution, similar to the one adopted for the tasklet availability queue.
- **Node type and lock information check.** The type of the node (i.e., manager, read proxy, write proxy) must be checked to ensure the current NIP node has permission to satisfy a lock request. Finally, the current lock on the object is checked to ensure that the multiple readers/single writer object access model is preserved.

Most of the overheads described above can be avoided for immutable objects only when the locking operation takes place on the manager node of those objects. This is the reason for the difference in the performance of the locking operations presented in Table 1-1 and Figure 6-1.

6.4. NIPLTC Micro-Benchmarks

The investigation into the performance of the NIP run-time techniques starts with the evaluation of NIP lazy task creation. A number of micro-benchmarks are devised to test the applicability of NIPLTC. The benchmarks are executed both on the shared-memory

and distributed-memory architectures. The tests attempt to measure the additional overhead incurred due to NIPLTC but also the speedups that can be achieved.

None of the tests that follow attempt to make a direct comparison between NIPLTC and eager task creation. Eager task creation would have introduced a great execution overhead, as the thread creation costs shown in Table 6-2 (page 115) suggest, for the applications with a great degree of parallelism that are examined next. Of course, the applications could be explicitly written in a manner that better utilised the available computational resources but, then, they would not comply with the semantics of the NIP programming and/or execution models. The benefits of lazy over eager task creation were presented in previous works (Goldstein 1997; Goldstein et al. 1996; Mohr et al. 1991). This section concentrates on establishing the applicability of NIPLTC under different cases of parallel computations.

6.4.1. Iterative Tasklet – Parallel Map – APP

The first of the micro-benchmarks involves the iterative tasklet construct (Chapter 3), an instance of which is used to represent a data parallel computation. The computation involves the application of a function *f* on the elements of a vector. Code 6-1 presents the pseudo code of the data parallel computation, in harmony with the NIP programming model semantics.

Both the size of the vector and the granularity of the function are varied in the following tests. The vector size determines the degree of parallelism and the grain size of the applied function controls the granularity of every potentially parallel task.

```
main()
  Vector<double> vector(X)
  for i = 0 to vector.size()
    f(a[i])
```

Code 6-1: Pseudo code for an iterative computation

Parallel Tasks Executing Single Iterations

By means of an appropriate NIP-aware compiler, Code 6-1 can be translated to Code 6-2, which is consistent with the NIP execution model semantics without the object memory. The parallelism in the computation is expressed through an iterative tasklet. All the iterations in the loop of Code 6-1 are exposed as potentially parallel tasks through just one instance of the NIP iterative tasklet (Code 6-2). However, a lazily created task from the tasklet can execute only one iteration.

The `MapTasklet` tasklet class is not part of the NIP run-time library (Chapter 4). It is a specialisation of the general `Tasklet` type and it is specifically created for the benefit of this micro-benchmark. The `createTask(NIPTask&)` method (Chapter 3) of the tasklet explicitly places the required data into the state of the task being created. Hence, there is no need for the object-based-shared memory of the NIP execution model (i.e., the NIPDSM). In this manner, the performance evaluation of the micro-benchmark can concentrate on issues related to NIPLTC, without the influence of the NIPDSM. A version of this micro-benchmark, which depends on the object memory for data access and uses one of the NIP run-time library provided tasklets, will be employed later in this chapter, in the assessment of one of the NIPDSM caching techniques.

```
main()
  Vector<double> vector(X)
  MapTasklet<double, f> tasklet(vector)
  tasklet.waitOrInline()
```

Code 6-2: The resulting pseudo code from the translation of Code 6-1 consistent to the NIP execution model semantics without object memory

Although the cost of the tasklet related operations were measured and presented in Table 6-3 (page 116), the overall run-time overhead on the computation is also calculated here. The sequential version of the computation of Code 6-1 is implemented in C++ and compared against the NIP version executed on one node. The difference between the execution times of the two versions corresponds to the total overheads introduced due to the NIP run-time related operations.

		Vector size			
		100	**500**	**1000**	**2000**
Function granularity (msecs)	**0.01**	8.92%	7.29%	8.73%	7.17%
	0.03	2.04%	1.69%	1.62%	1.40%
	0.05	1.40%	0.99%	0.92%	0.84%
	0.11	0.69%	0.51%	0.38%	0.65%
	0.27	0.29%	0.26%	0.88%	0.25%
	0.54	0.20%	0.19%	0.15%	0.16%
	1.35	0.14%	0.14%	0.13%	0.15%
	2.69	0.12%	0.14%	0.10%	0.09%
	5.38	0.08%	0.10%	0.12%	0.10%
	8.07	0.20%	0.29%	0.10%	0.08%
	10.76	0.09%	0.08%	0.08%	0.12%

Table 6-4: The execution overhead introduced due to NIP run-time related operations as a percentage of the execution time of sequential version of the computation presented in Code 6-1

Table 6-4 shows the additional overhead incurred from the execution of the NIP version of the computation as a percentage of the C++ sequential version. As the

granularity of the function applied to the elements of the vector increases, the effect of the NIP related operations decreases. This is because the NIP related overheads are fixed. They are only incurred during the construction and destruction of the iterative tasklet instance and during a stealing operation for an iteration (Chapter 3) that is to be executed inline. Therefore, the increase of the function's granularity does not introduce additional overheads.

Next, the speedup achieved from the execution of the NIP version of the computation on the APP is measured (Figure 6-2). For fine granularities, a smaller rate in the efficiency is observed as the number of nodes increases (Figure 6-3). This is due to the great communication overheads on the APP and the high cost of lazily creating tasks on remote nodes, as shown in Table 6-3 (page 116). The number of task stealing operations increases with the number of nodes, thus the introduced run-time overheads are higher. When the granularity of the computation is fine, the impact of these overheads is greater and, as a result, the efficiency is affected.

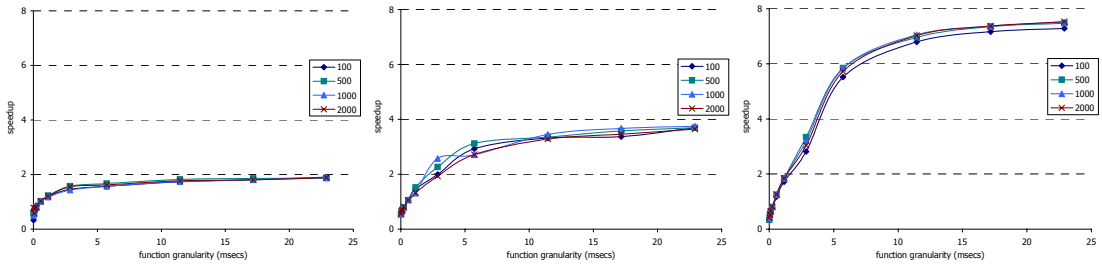


Figure 6-2: Speedup achieved on *2*, *4*, and *8* nodes (vector size: *100*, *500*, *1000*, *2000*)

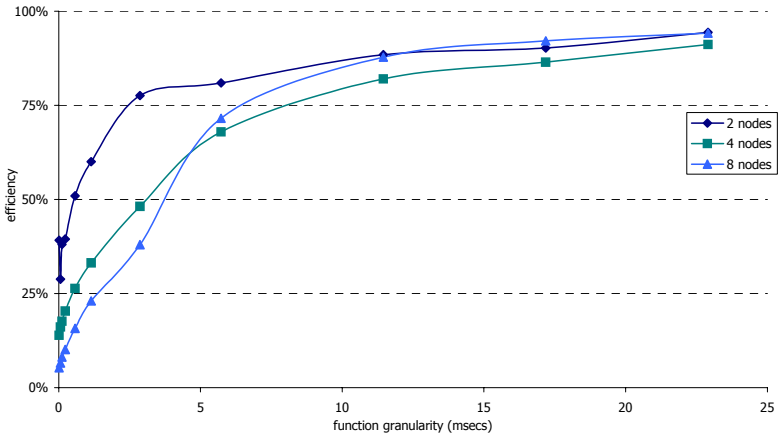


Figure 6-3: Efficiency achieved on *2*, *4*, and *8* nodes (vector size: *2000*)

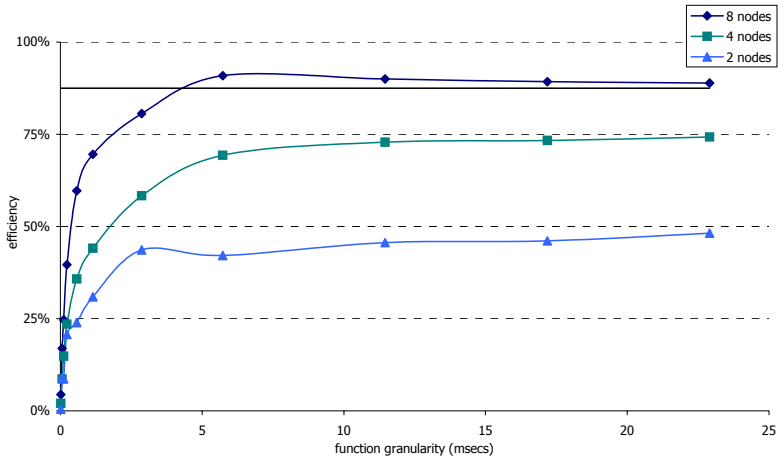


Figure 6-4: % of tasks created out of 2,000 possible for different function granularities and for different number of nodes used (non-optimised iterative tasklet)

In order to support the above argument, the number of lazily created tasks was measured. The percentage of the tasks that were created depends on the granularity of the function and the number of nodes used, as Figure 6-4 suggests. When the granularity of the function is large, the secondary nodes of the parallel system have more chances of stealing iterations from the primary node, which is inlining as many iterations as possible. Furthermore, as the number of nodes increases, more parallel tasks can be executed simultaneously.

It is expected that the potentially parallel work be evenly distributed on the available processing nodes. For example, on *2*, *4*, and *8* nodes, it is expected that *50%*, *75%*, and *87.5%* of the available tasks are lazily created and executed on the available processors respectively. Figure 6-4 presents the percentage of tasks that were lazily created during the execution of the micro-benchmark. When *8* nodes are used, the measured number of lazily created tasks surpasses what was expected (the *87.5%* line of Figure 6-4 demonstrates the anomaly). Since only one node in the parallel system has available work to offer, all the other nodes attempt to steal tasks from it. As that node has to deal with the requests, it does not get enough time to execute its part of the computation (i.e., inline iterations). Therefore, other nodes execute a larger piece of the entire computation.

The results of Figure 6-4 suggest that both NIPLTC and the NIP load balancing service work well together to distribute the work around the parallel system. However, since the additional overhead of lazily creating a task for the execution of only one iteration is significant, the performance of the micro-benchmark is poor for smaller granularities.

Parallel Tasks Executing Groups of Iterations

The number of parallel tasks actually created from the iterative tasklet presented above is very large. Every task that is lazily created only executes one iteration. Chapter 3 proposed an optimisation to the original iterative tasklet by suggesting that a group of iterations could be executed together. In this manner, the granularity of the lazily created parallel tasks could be implicitly increased.

```
main()
  Vector<double> vector(X)
  MapTaskletGroup<double, f> tasklet(vector)
  tasklet.waitOrInline()
```

Code 6-3: Optimised version of the pseudo code presented in Code 6-2

The iterative tasklet of Code 6-2 is replaced by a new specialisation of the tasklet construct. The new tasklet allows a group of iterations to be stolen at a time. The default number of iterations to be included in a group is calculated according to Equation 6-1. If during the execution of an application the number of the remaining, non-evaluated iterations is less than the group size, then all the iterations are included in the same group. The equation divides the total number of iterations into groups, according to the optimal availability of computational resources in the parallel system (it is assumed that every NIP node in the parallel system has an equal number of processors). The size of the group is divided by two to give the NIP node that created the iterative tasklet more chances of inlining iterations. When the iterations are fine-grained, the execution of the application suffers less from the overheads of lazily creating tasks because more iterations can be executed inline, since a smaller part of the entire computation is executed on remote NIP nodes.

$$group\ size = \frac{total\ number\ of\ iterations}{number\ of\ nodes * number\ of\ processors * 2} \qquad \textbf{Equation 6-1}$$

Function granularity (msecs)	Vector size 100	500	1000	2000
0.01	2.98%	1.73%	0.85%	1.03%
0.03	0.70%	0.29%	0.32%	0.11%
0.05	0.59%	0.28%	0.67%	0.18%
0.11	0.31%	0.22%	0.05%	0.06%
0.27	0.18%	0.15%	0.10%	0.10%
0.54	0.13%	0.11%	0.14%	0.10%
1.35	0.19%	0.10%	0.09%	0.20%
2.69	0.09%	0.12%	0.08%	0.07%
5.38	0.06%	0.08%	0.11%	0.09%
8.07	0.19%	0.08%	0.09%	0.08%
10.76	0.12%	0.08%	0.08%	0.10%

Table 6-5: The execution overhead introduced due to NIP run-time related operations as a percentage of the execution time of sequential version of the computation presented in Code 6-3

The optimised version reduces the NIPLTC related overheads because the number of stealing operations is decreased. Table 6-5 presents the run-time overhead as a percentage of the C++ sequential version of the computation. When compared to Table 6-4 (page 119), it is clear that for smaller granularities the overhead is dramatically reduced. There is no difference for larger granularities because the NIP related overhead is minimal when compared to the total computation time.

The grouping approach to stealing iterations from a tasklet dramatically improves the performance of the run-time environment for smaller granularities. The same degree of speedup is achieved but for smaller granularities than was possible with the iterative tasklet of the previous section. Figure 6-5 compares the speedup achieved for function granularities up to *3msecs* for both the grouping-capable and the original iterative tasklets.

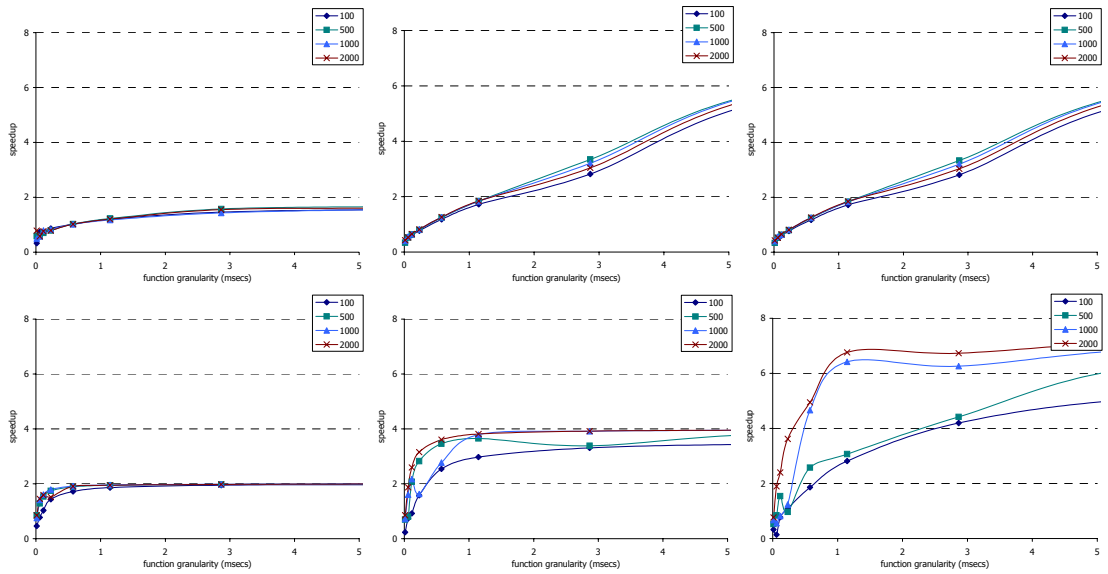


Figure 6-5: Comparison between the speedups achieved when a single iteration (top row) is stolen and when a group of iterations (bottom row) is stolen, for *2*, *4*, and *8* nodes and for fine granularities of the function (vector size: *100*, *500*, *1000*, *2000*)

The nodes of the parallel system are kept busy without having to continuously steal jobs from others because they receive more than one iterations to execute at a time. Despite the implicit increase in the granularity of the parallel tasks, the degree of parallelism of the entire application is not reduced. When a node receives a group of iterations, a new tasklet is immediately created to represent the potentially parallel tasks in the delivered group. Then, the node starts inlining the iterations from the created tasklet while others can steal work from it. This approach also means that there is a better distribution of work in the parallel system. Now, it is not only the node where the iterative tasklet was created that has available work for other nodes to steal. This means that a single node does not become a 'hot spot' for work requests, which results in better performance. This problem was first illustrated in the previous section (Figure 6-4, page 120).

		Nodes sending tasks															
		Original iterative tasklet								**Grouping-capable iterative tasklet**							
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Nodes stealing tasks	**1**		0	0	0	0	0	0	0		0	0	0	0	0	0	0
	2	254.0		0	0	0	0	0	0	0.6		3.3	1.7	1.0	0.3	0.7	0
	3	254.0	0		0	0	0	0	0	0.6	0		20.3	12.3	8.3	4.0	4.0
	4	254.0	0	0		0	0	0	0	0.7	0.3	0		14.0	11.0	6.7	4.7
	5	254.0	0	0	0		0	0	0	0.7	0.3	0.3	0		14.0	10.3	7.7
	6	254.0	0	0	0	0		0	0	0.8	1.3	1.3	1.0	0		16.0	8.3
	7	254.0	0	0	0	0	0		0	0.9	6.7	2.3	2.7	1.3	0		14.7
	8	253.7	0	0	0	0	0	0		1.2	7.0	1.3	2.0	0.7	0	0	

Table 6-6: Average of the tasks stolen from the repeated execution of the tests (number of nodes used: *8*, vector size: *2000*, function granularity: *~23msecs*)

Table 6-6 presents a comparison between the two versions of the iterative tasklet. The exchange of tasks between the nodes of the parallel system is recorded. In the original version, only the primary node (node 1) has work to offer and, as such, it becomes a 'hot spot' of work stealing requests. With the grouping-capable version, the work is better distributed across the parallel platform. It has to be noted that in the optimised version a group does not always contain the same number of iterations. The *total number of iterations* factor of Equation 6-1 may differ between nodes as it depends on the size of the group received.

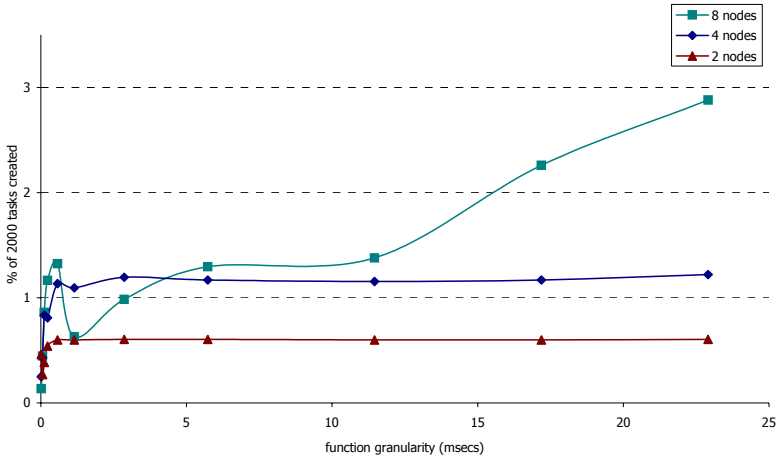


Figure 6-6: % of tasks created out of 2,000 possible for different function granularities and for different number of nodes used (optimised iterative tasklet)

It can also be observed that the number of tasks lazily created from the grouping-capable version of the iterative tasklet is significantly smaller than from the original version. This is attributed to the resulting larger granularity of the created tasks. The comparison of Figure 6-4 (page 120) and Figure 6-6 demonstrates the substantial difference between the numbers of lazily created tasks from the two versions of the iterative tasklet.

When the grouping-capable iterative tasklet is used, the granularity of the lazily created tasks greatly depends on the load balancing algorithm. The algorithm used in the execution of this and all the other benchmarks, is the one described in Chapter 5, where a 'request-for-work' message is sent to the next node (in ascending order) with a non-empty tasklet availability queue. However, the work stolen from secondary nodes is of finer granularity than the work that is stolen from the primary node. This results in fine-grained lazily created tasks as the number of nodes increases, which, in turn, means larger number of tasks (Figure 6-6).

In the grouping-capable version, if the primary node runs out of work to inline, it des not have to wait for the execution of the stolen tasks to complete. Instead, it may steal

back some of the available work in the parallel platform. With the original version, this would have not been possible because there is no other work available in the parallel platform. In the example of Table 6-6, the primary node does not receive any tasks from the other nodes because the execution of all the stolen tasks finishes before the primary node runs out of work.

The smaller number of lazily created tasks with coarser granularity results in better utilisation of the available computational resources. The nodes of the parallel platform do not waste valuable computational time waiting for a new task to arrive, as it is the case with the original version of the iterative tasklet. This observation is justified by the speedups for small function granularities presented in Figure 6-5 (page 122).

6.4.2. Iterative Tasklet – Parallel Map – SMP

The benefits of the iterative tasklet construct are also observed on physically shared memory multiprocessor architectures. As mentioned in Chapter 3, the implementation of the tasklet construct can be fine-tuned to better utilise SMP architectures. The tasks that are created from a tasklet have direct access to the state of that tasklet and, therefore, can also inline part of the associated computation. This approach reduces the number of lazily created tasks and improves performance by avoiding the overheads of creating new tasks.

Since network related overheads are not incurred, NIPLTC performs significantly better on the 4-way SMP than on the APP. Good speedups are achieved for very fine granularities of the function *f* (Code 6-1, page 118). The grouping-capable iterative tasklet results in even better performance (Figure 6-7). This is because the lazily created parallel tasks have a group of iterations to execute and, as a result, there is no congestion on the tasklet's private lock. Every inlining operation requires that the tasklet's private lock be acquired. In the case of the original iterative tasklet, every inlining operation involves only one iteration. In contrast, an inlining operation on the grouping-capable iterative tasklet involves a group of iterations and, therefore, the number of required inlining operations is reduced.

Due to the fine-tuning of the tasklet construct on physically shared-memory architectures, as described in Chapter 3, the degree of parallelism in the application is implicitly reduced when the grouping-capable version of the iterative tasklet is used. All the lazily created parallel tasks inline iterations without creating new tasklets to maintain the degree of parallelism. If the degree of parallelism needs to be preserved, a suitable specialisation of the iterative tasklet could be implemented.

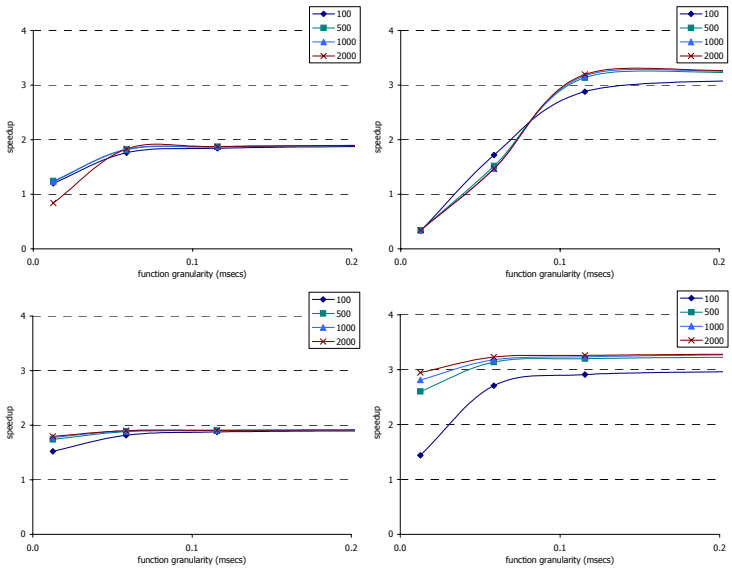


Figure 6-7: Speedups of the parallel map micro-benchmark on the 4-way SMP for the original (top row) and grouping-capable (bottom row) iterative tasklet (*2* and *4* processors, vector size: *100*, *500*, *1000*, *2000*)

Finally, it is observed that the speedup on *4* processors asymptotically reaches *~3.4* and does not increase even with the grouping-capable iterative tasklet. Even for larger granularities of the function *f* (not shown in Figure 6-7), the speedup does not exceed *3.4*. Although this may be attributed to a number of factors, such as smaller executable of the sequential code and therefore better utilisation of the processor cache, Linux thread scheduling, etc., it is very likely that there is a scalability problem with the micro-benchmark on SMP architectures. Only one tasklet is available from which all the lazily created parallel tasks are trying to inline iterations. A grouping-capable iterative tasklet that does not reduce the degree of the parallelism in the application on SMP architectures (i.e., new tasklets are created for every stolen group) could probably tackle the problem of scalability as inlining would take place from more than one tasklet simultaneously.

The performance benefits due to the grouping-capable version of the iterative tasklet demonstrate the strength of NIPLTC and the tasklet construct over previous lazy task creation approaches. In all the benchmark applications that follow, when an iterative tasklet is required, the grouping-capable version will be used.

6.4.3. Recursive Tasklet – Grain

The grain micro-benchmark used for the evaluation of the recursive tasklet (Chapter 3) is the same one used in (Mohr et al. 1991). The grain computation recursively adds up a perfect binary tree. When a leaf is reached, a function is called, which controls the overall granularity of the computation, before the value *1* is returned. As in the original grain benchmark, a tree-depth of *16* is used. The number of leaves in the tree determines the

degree of logical parallelism of the computation given that a separate parallel task could be created to evaluate the function associated with each leaf. The number of potential parallel tasks for a tree-depth of *16* is *65,536*.

Code 6-4 presents the pseudo code of the grain micro-benchmark. A suitable, NIP-aware compiler could transform the program to match the NIP execution model semantics (Code 6-5). The NIPDSM is not used because the grain micro-benchmark does not require access to any objects placed in the shared memory. A more detailed discussion of the recursive tasklet can be found in Chapter 3.

```
grain(int depth)
  if (depth > 0)
    return grain(depth – 1) + grain(depth – 1)
  else
    f()
    return 1

main()
   grain(16)
```

Code 6-4: The grain pseudo code consistent to the NIP programming model semantics

```
grain(NIPRecursiveTasklet& tasklet, int depth)
  if (depth > 0)
    NIPRecursiveTaskletNode grainNode(depth – 1)
    int tmp = grain(depth – 1)
    tasklet.waitOrInline(grainNode)
    return tmp + grainNode.result()
  else
    f()
    return 1

main()
   NIPRecursiveTaskletNode grainNode(16)
   NIPRecursiveTasklet tasklet(grainNode)
   grain(tasklet, 16)
   tasklet.waitOrInline()
```

Code 6-5: The grain pseudo code converted to be consistent with NIP execution model semantics

The additional overheads introduced in the execution time of the grain benchmark by NIP related operations are presented in Table 6-7.

Function granularity (msecs)	APP workstation	SMP workstation
0.01	42.09%	42.65%
0.02	13.45%	13.42%
0.04	5.84%	5.79%
0.06	3.61%	3.59%
0.08	2.67%	2.64%
0.10	2.12%	2.08%
0.15	1.39%	1.37%
0.20	1.01%	1.04%
0.50	0.41%	0.42%

Table 6-7: The execution overhead introduced due to NIP related operations as a percentage of the execution of sequential version of the grain micro-benchmark for the an APP workstation and the SMP workstation for different granularities

The grain micro benchmark is executed on both the APP and the SMP parallel platforms. The speedup achieved on the APP is presented in Figure 6-8 and the percentage of lazily created tasks is shown in Table 6-8. The very good speedups that are achieved on the APP (Figure 6-8), despite the slow network that is used (shared Fast Ethernet 100Mbit/s), are attributed to the following factors:

- Tasks are stolen from the top of the recursive tasklet's queue and, therefore, their granularity is coarse (Chapter 3). As a result, there are only a small number of coarse-grained tasks created (Table 6-8). The NIP nodes are kept busy without having to look for work frequently.
- There is a good distribution of work in the parallel platform. When a task receives a sub-tree of the computation to execute, it creates a new tasklet to represent the potential parallelism in the received job. The primary node does not become a 'hot spot' of work stealing requests.

The grain micro-benchmark performs well on the SMP workstation (Figure 6-9). Only a fraction of the total number of potentially parallel tasks is created. The tasks that are lazily created are of coarse granularity, resulting in better utilisation of resources. The implementation of the recursive tasklet is optimised for SMP architectures, like the iterative tasklet. The same factors that may be influencing the scalability of the iterative tasklet (Section 6.4.2) may also be affecting the recursive tasklet. Figure 6-9 suggests that an upper limit exists to the speedup that can be achieved on *4* processors.

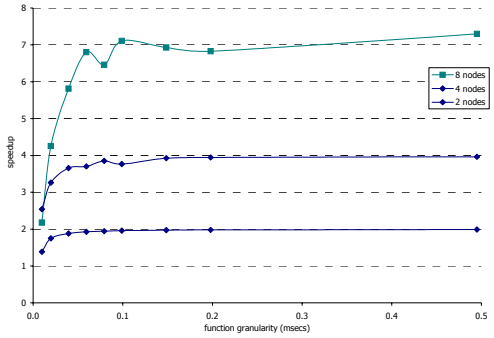


Figure 6-8: Speedups of the grain micro-benchmark on the APP

Function granularity (msecs)	2 nodes	4 nodes	8 nodes
0.01	0.002%	0.012%	0.038%
0.02	0.002%	0.006%	0.055%
0.04	0.003%	0.008%	0.095%
0.06	0.003%	0.015%	0.037%
0.08	0.003%	0.011%	0.044%
0.10	0.003%	0.012%	0.082%
0.15	0.003%	0.014%	0.142%

Table 6-8: Percentage of tasks created out of *65,536* possible on the APP

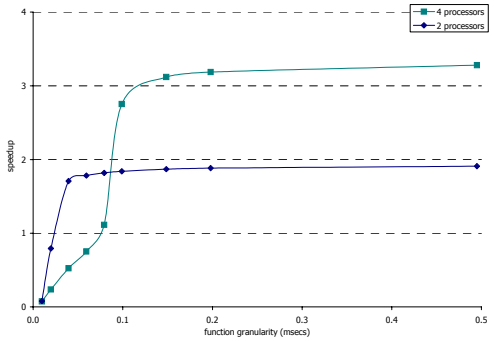


Figure 6-9: Speedups of the grain micro-benchmark on the SMP

Function granularity (msecs)	2 processors	4 processors
0.01	0.003%	0.034%
0.02	0.006%	0.024%
0.04	0.003%	0.024%
0.06	0.006%	0.031%
0.08	0.005%	0.020%
0.10	0.008%	0.024%
0.15	0.003%	0.017%

Table 6-9: Percentage of tasks created out of *65,536* possible on the SMP

6.5. NIPDSM Micro-Benchmarks

The micro-benchmarks of the previous section demonstrated the effectiveness of the NIPLTC technique on both physically shared and distributed memory architectures. However, none of those micro-benchmarks made use of the NIP execution model memory semantics that define the object-structured shared memory. In this section, a series of tests are used to evaluate NIPDSM, the implementation of the NIP execution model memory semantics, and the introduced caching techniques (Chapter 4).

In the current implementation of the NIP run-time library, the cost of accessing an object in the NIPDSM is exceptionally high, as shown in Table 6-3 (page 116). At least one pair of acquire and release mutex operations is required for every lock operation on an object. On proxy nodes, two different mutexes must be acquired and released before a method can be called on a cached copy of the object. Locking operations on immutable objects that take place on the manager node of those objects are not as expensive, since the part of the NIPDSM locking process that guarantees thread-safe access to the NIPDSM VOT is not required.

The cost of a cache miss incorporates the network communication and thread suspension/resumption costs, in addition to the overheads due to mutex operations on

both the proxy and manager nodes. The cost of accessing a cached object is very close to the cost of accessing an object at the manager node (Table 6-3, page 116). The additional overhead is attributed to an extra pair of mutex acquire/release operations that must take place when checking the availability of the required NIPDSM VOT tables. The considerable difference in the run-time overheads between a cache-miss and a cache-hit illustrates the importance of the object grouping caching techniques, which improve the cache-hit rates as will be shown shortly.

The cost of accessing a NIPDSM object is significantly higher when compared to the overhead of accessing an object in virtual memory. This will probably result in memory-intensive applications performing poorly. Better results should be achieved for applications with a high computation/NIPDSM operations ratio.

	Elapsed (usecs)
VM object access	0.017
NIPDSM access through NIP reference	0.094
NIPDSM immutable object access immutable on manager node (lock operation)	0.366
NIPDSM mutable object access on manager node (lock operation)	2.134
NIPDSM cache miss	461.715
NIPDSM cache hit	3.664
NIPDSM proxy invalidation	451.516

Table 6-10: Cost of NIPDSM operations

An additional issue that affects performance of memory-intensive applications is the allocation of objects in the NIPDSM. Objects are not distributed around the nodes of the parallel system when created. Instead, the node that creates an object automatically becomes the manager node for that object. Many applications create all the necessary objects before they initiate any parallel computations. With the NIP run-time, this results in the node creating the objects being overwhelmed with proxy requests.

Even if the implementation were improved, however, the cost of cache misses would remain high due to the overheads of the network related operations. Therefore, one of the main aims of the NIPDSM is to improve caching. To that extent, the analysis of the performance-related results that follows focuses primarily on the applicability of the NIPDSM caching techniques for various memory access patterns.

In the performance evaluation that follows, a specific NIPDSM caching technique is enabled or disabled according to the requirements of a particular micro-benchmark.

However, even when none of the caching techniques is enabled, an object may be found in the cache of a proxy node due to temporal locality. This is because the NIP entry consistency semantics specify that an object remains at a node until it is invalidated (Chapter 4). The micro-benchmarks that follow explicitly invalidate cached objects when required in order to investigate the characteristics of a specific object caching technique. Finally, in the discussion that follows, 'no caching' means that none of the object grouping techniques was used.

6.5.1. Object Grouping Based on Location – Parallel Map

The micro-benchmark of Section 6.4.1 is modified and used in the evaluation of the first of the caching techniques. The version of the benchmark that was used to test NIPLTC did not make use of the memory semantics of the NIP execution mode because no shared memory was assumed. The required data for the execution of an iteration was transferred as part of the `Task` object.

The code of the micro-benchmark that is consistent with the NIP execution model semantics is presented in Code 6-6. The `NIPIterativeTaskletGroup` tasklet is part of the NIP run-time library. The only data exchanged between nodes as part of the lazy task creation process is the argument to the function that is executed for each iteration, a `NIPRef<double>` instance in this case. The availability of any data is guaranteed through the NIPDSM locking operations. For every iteration in the loop, the required element from the vector must be locked before it can be accessed (Code 6-6).

```
typedef NIPObject<double> Double

iter(int i, NIPRef<Double> vectorRef)
  Double& d = vectorRef[i].lockRead()
  f(d)
  vectorRef.unlock()

main()
  NIPRef<Double> vectorRef = NIPDSMNewObject<Double>(Mutable, N)
  NIPIterativeTaskletGroup<NIPRef<Double>, iter> tasklet(N, vectorRef)
  tasklet.waitOrInline()
```

Code 6-6: Pseudo code for the parallel map micro-benchmark consistent with the NIP execution model semantics

The application of Code 6-6 exhibits spatial locality in memory access. This is because the elements of the vector are spatially adjacent to each other and they are accessed in an iterative manner. The object grouping caching technique was designed and implemented to favour this particular memory access pattern.

The following series of speedup graphs illustrate the impact of the caching technique to the performance of the micro-benchmark (Figure 6-10). The first row of graphs shows

the speedups achieved for *2*, *4*, and *8* nodes without the use of any caching techniques. In contrast, the graphs of the bottom row are generated from the execution of the micro-benchmark with object-grouping based on location enabled.

The performance of the micro-benchmark is improved when the caching technique is enabled. Due to the overheads associated with the NIPDSM, the speedups attained are worse, but not to a great extent, than what was achieved with the earlier version of the micro-benchmark (Figure 6-5, page 122). Since the NIPDSM related overheads are fixed, the executions of the micro-benchmark for large granularities are affected less. For example, for a granularity of *22.91msecs* and vector size of *2,000* on *8* nodes (this configuration is not shown in any of the graphs of Figure 6-10), a speedup of *7.3* is achieved, which compares to the speedup of *7.6* that was possible with the non-NIPDSM version. For the same configuration (i.e., function granularity, vector size, and number of nodes) but without object grouping enabled, a speedup of only *6.3* is achieved. The micro-benchmark performs poorly when caching is disabled because the tasks on secondary nodes are repeatedly suspended. This is because the execution of every iteration has to suspend on a locking operation until the object being accessed can be fetched from its manager node, which in this case is always the primary node. When the granularity of the function is fine, the additional run-time cost of the cache miss overwhelms the execution time. Object grouping reduces the number of times the execution has to wait.

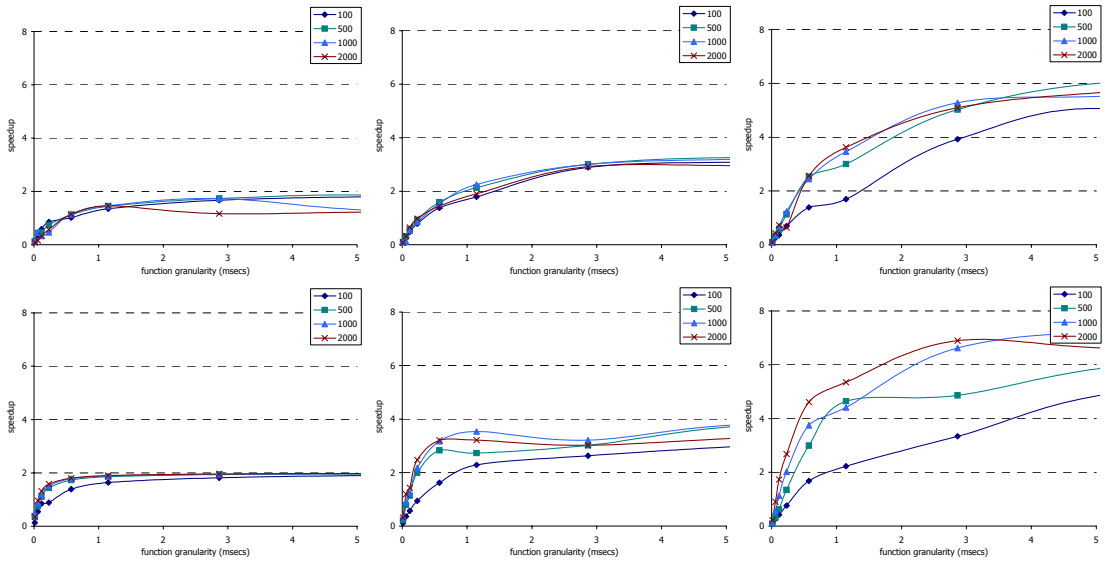


Figure 6-10: Comparison between the speedups achieved when a caching technique is not used (top row) and when a object grouping based on location (bottom row) is used (number of nodes: *2*, *4*, *8*; vector size: *100*, *500*, *1000*, *2000*)

The graphs of Figure 6-10 give the impression that as the number of nodes increases, the advantages due to object grouping based on location become less obvious. There seems to be an association between the number of nodes, the degree of logical parallelism

and the speedup achieved. A greater number of nodes result in poorer speedups for smaller vectors.

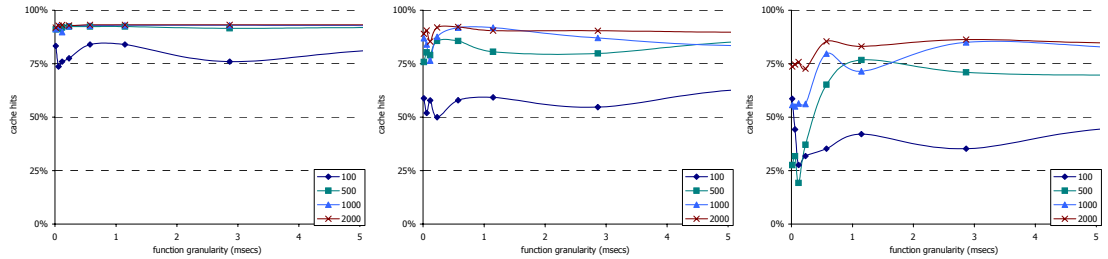


Figure 6-11: Cache-hit rates for the map micro-benchmark on *2*, *4*, and *8* nodes and for different granularities of the function (vector size: *100*, *500*, *1000*, *2000*)

The total number of available potentially parallel tasks in the system is fixed for this micro-benchmark and it depends on the size of the vector used. As the number of the available processing nodes increases, the available work is split to smaller groups through NIPLTC and distributed across the parallel system. The smaller number of iterations on a node results in a lower cache-hit rate. Consequently, the total cache-hit rate on the parallel system is reduced, which results in reduced performance. Figure 6-11 confirms the above discussion. For larger vectors, the number of iterations executed on each node is enough to keep the cache-hit rate high.

The decrease in performance due to the increase in processing nodes for the parallel map micro-benchmark would have been the same with any page-based caching scheme. The high cache-hit rates that have been achieved demonstrate that object grouping based on location can benefit applications exhibiting spatial locality in memory access.

6.5.2. Object Grouping Based on Associations – TreeSum

Object grouping based on associations is the NIPDSM caching technique that attempts to improve the memory access times for applications that traverse through dynamically created data structures, like lists, trees, etc. NIPDSM can make use of information about the layout of the data structures at run-time, in order to increase the cache-hit rates. Applications are required to explicitly identify the associations between objects (Chapter 4). The identified associations are used when NIPDSM chooses the objects to be included in a cache block.

A new micro-benchmark was devised to test the behaviour of the caching technique. It is assumed that a tree of arbitrary depth is created dynamically over time and that this results in the nodes of the tree being spread throughout the memory (i.e., the nodes are not spatially adjacent). A node is a NIPDSM object and, therefore, it needs to be locked

before its state can be accessed. Each node has four children and stores the value *1*. During the construction of the tree, every node is associated with its children.

Once the tree has been initialised, a task is created on a secondary node to calculate the summary of all the values in the tree. In order to allow the performance evaluation to concentrate on the NIPDSM technique, the NIPLTC is not used in this micro-benchmark.

Since each node in the tree is accessed only once and the nodes are not considered to be spatially adjacent, every locking operation would cause a cache-miss even when the object grouping based on location caching technique is enabled (i.e., cache-hit rate of *0%*).

The treesum micro-benchmark was executed for a varying number of tree nodes and cache block sizes. The achieved cache-hit rates range between *~36%* and *~93%* (Figure 6-12). Figure 6-12 suggests that the cache-hit rate depends on both the depth of the tree and the size of the NIPDSM cache block.

The size of each of the nodes in the tree is *64* bytes. Table 6-11 presents the total number of objects for the different tree depths. The entire tree is transferred from the primary node to the secondary node, where the tree traversal algorithm is evaluated. The number of objects is the same as the total number of locks required for the entire tree to be traversed. As the size of the cache block increases, more objects can be transferred together during a locking operation. However, as Figure 6-12 demonstrates, there are cases where for the same number of objects (constant tree depth) a larger cache block does not result in cache-hit rate increase.

There does not seem to be a clear link between the effectiveness of the NIPDSM caching technique in terms of cache-hit rate, the cache block size, and the depth of the tree. The reason for the results of Figure 6-12 can be found if the behaviour of the NIPDSM caching algorithm is considered. Figure 6-13 presents an example that illustrates the correlation between the tree size, the cache block, and the cache-hit rate. Two quadruple trees are shown of size *3* and *4* respectively. If it is assumed that a NIPDSM cache block can only accommodate four objects, then the cache-hit rates for the two trees are *19%* and *78.8%* respectively. The significant difference in the cache-hit rates between the two trees is attributed to the manner in which NIPDSM places associated objects in the cache block.

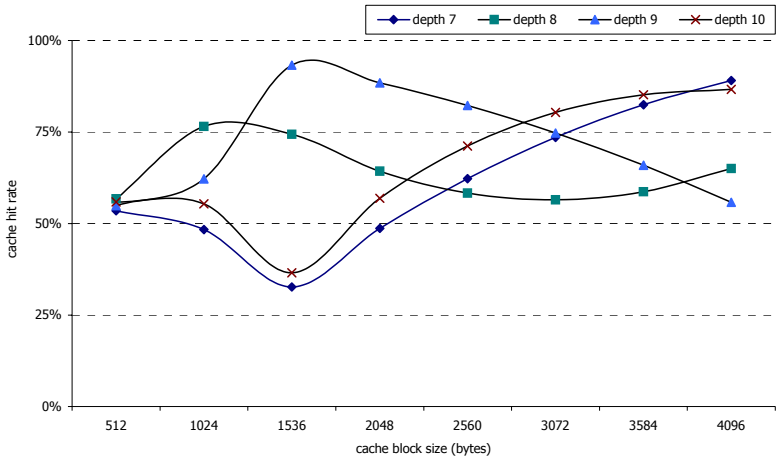


Figure 6-12: Cache-hit rate for the treesum micro-benchmark for different tree depths and cache block sizes

Tree depth	Number of tree nodes
7	5,461
8	21,845
9	87,381
10	349,525
Node size	64 bytes

Table 6-11: Number of objects transferred (and read lock operations) for each tree depth and size of each tree node

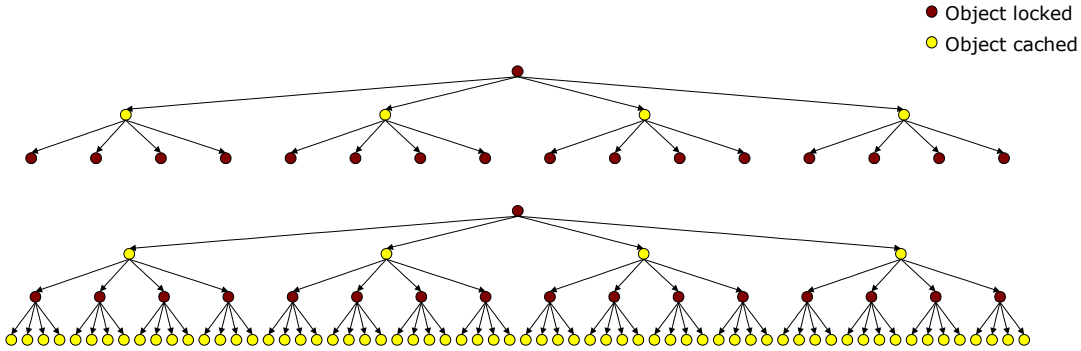


Figure 6-13: Example of object grouping based on relations and the association between the tree-depth and the number of cache hits

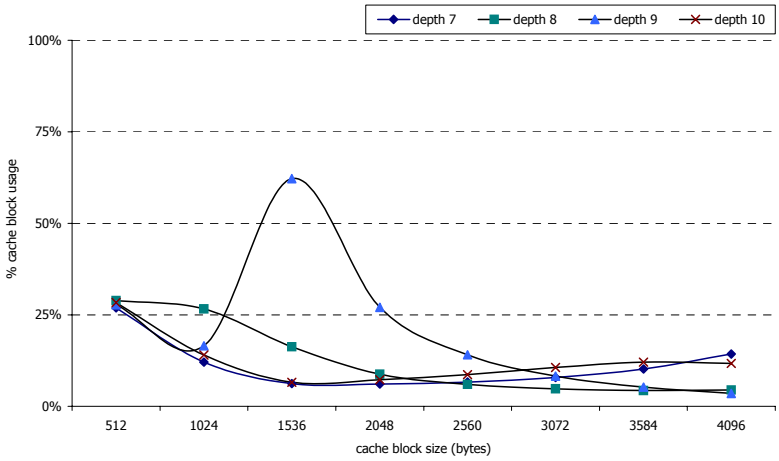


Figure 6-14: Cache block usage for the treesum benchmark

As Chapter 4 described, the associations between NIPDSM objects formulate a graph. The algorithm used to place objects into the cache block traverses the graph of associations in a breadth-first manner. This means that if the last objects to be placed in

the cache block are those at the `depth-1` level, then all their children (i.e., the leaves) will have to be cached on their own because they are not associated with any other object (first tree of Figure 6-13). If, however, when an object at `depth-1` is placed in the cache block and there is still space for other objects, its children will also be included (second tree of Figure 6-13).

Different cache block sizes result in distinct object caching behaviour. To illustrate this, the cache block usage is calculated and presented in Figure 6-14 for different cache block sizes. The total number of objects that fit into a cache block can be determined since the size of each tree node is known (Table 6-11). As it is shown in Figure 6-14, the cache block is not fully used.

Despite the variations in the cache-hit rate and the non-optimal use of the cache block, good cache-hit rates were achieved (Figure 6-12). This demonstrates the effectiveness of object grouping based on associations when data structures are accessed because the significant overheads of cache-misses (Table 6-10, page 128) will not be incurred to the same extent as when the cache-hit rate is low. The cache-hit rate may be low or even zero when none of the object grouping caching techniques is enabled or when object grouping based on location is used, since the tree nodes are not spatially adjacent. The object grouping based on locking history (examined in the next section) would not have increased the cache-hit rate either, since the tree nodes are only accessed once (i.e., there is no recurring object access).

The observed problems were due to the mismatch of the tree traversal and NIPDSM object grouping algorithms. A tree with fewer children per node would suffer less from the problem discussed above, while the cache-hit rate for other data structures, like lists, will be significantly better (i.e., the maximum use of the cache block is achieved).

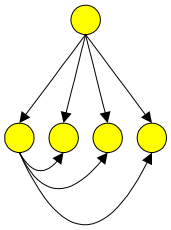


Figure 6-15: Object associations in order to improve the cache-hit rate

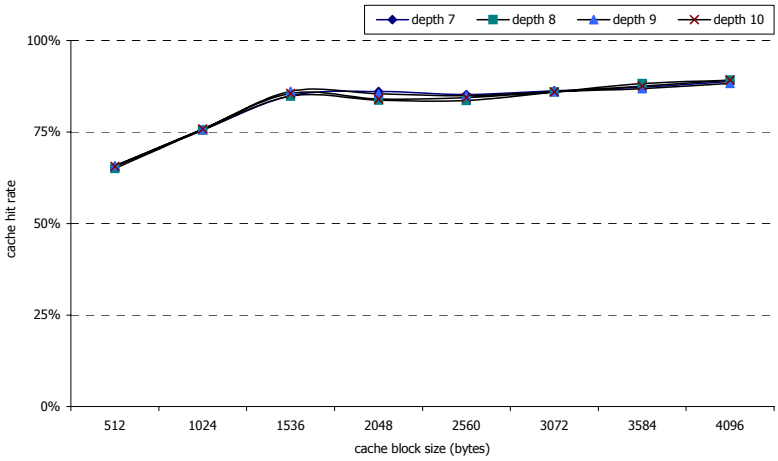


Figure 6-16: Cache-hit rate for the optimised treesum micro-benchmark for different tree depths and cache block sizes

If the cache-hit rate for this algorithm was to be improved, additional associations between objects are required. Besides the associations of each tree node with its children, the left most node in each subtree is associated with the nodes with which it shares its parent (Figure 6-15). The cache-hit rate is improved and it is almost identical for all the tree depths.

To further illustrate the benefits of the object grouping based on associations caching technique, the cache-hit rate achieved from the execution of a micro-benchmark similar to treesum was also measured. In the list-iteration micro-benchmark, a task is created on a secondary node to iterate through the nodes of a list data structure, the nodes of which are spread in memory (i.e., the nodes of the tree are not spatially adjacent). Every node of the list is associated with its next node (Figure 6-17). The cache-hit rate achieved is shown in Figure 6-18.

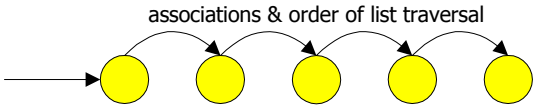


Figure 6-17: An example of a list data structure and the associations between nodes

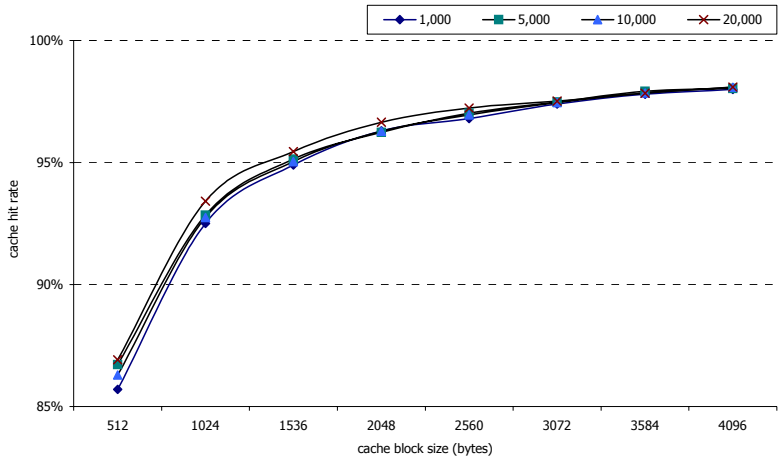


Figure 6-18: Cache-hit rate for the list-iteration micro-benchmark (number of elements in the list: *1,000*, *5,000*, *10,000*, *20,000*)

The object grouping technique could be used for objects that are associated with each other even if when they are not part of the same data structure. The graph of associations between NIPDSM objects could identify a memory access pattern in an application or even explicit links between otherwise unrelated objects. The matrix multiplication benchmark investigates the use of object associations in defining memory access patterns (Section 6.6.1).

6.5.3. Object Grouping Based on Access History – Tree Search

The final NIPDSM caching technique to be evaluated is the object grouping based on access history. A new micro-benchmark was devised to test the technique. A sorted binary tree of *1,024* objects is created and a search algorithm is used to locate a random object in that tree. The search algorithm is repeated *1,000* times for a different random object each time.

A task is explicitly created using appropriate NIP run-time library calls on a secondary node. As a result, none of the objects required by the search algorithm are available locally on that secondary node. After the completion of the search algorithm, all the objects that have been fetched from the primary node are flushed from the secondary node's cache. The next invocation of the search algorithm will not find any objects available on that secondary node.

The random object used as a criterion in the search algorithm is chosen from one of the following pre-specified ranges in turn: the first $1/2, 1/16, 1/32, 1/128$ part of the sorted range of leaves. In order to allow the performance evaluation to concentrate on the NIPDSM technique, the NIPLTC is not used in this micro-benchmark. Furthermore, the elements

of the tree are assumed not to be spatially adjacent to each other. Hence, enabling the object grouping based on location technique would not result in any cache-hit rates.

As the range from which the object is randomly selected becomes narrower, the chances the search algorithm will access objects in the same order increases. The top part of the tree that is common for all runs of the search algorithm includes more objects. As an example, the first four steps of the micro-benchmark for *16* leaves are illustrated in Figure 6-19. When object *5* is to be located, none of the objects along the route is included in the cache block as part of a grouping operation, resulting in zero cache-hit rate. However, NIPDSM records the order in which the objects are accessed and uses that information in the next run. If object *7* is to be located by the search algorithm, the lock on object *a* will result in objects *b* and *e* to be cached as well (the cache-hit rate is *2/5*). Although the link from object *e* to object *i* is lost (Section 4.8.4, page 90), the link from object *i* to object *5* is maintained. As more instances of the algorithm are executed, the locking history information is built up. In Figure 6-19, the random object was chosen from the first half of the set of leaves. This resulted in objects *a* and *b* always being locked, during an execution of the search algorithm. If the object was selected from the first quarter of the same set, objects *a*, *b*, and *d* will always be cached as a group.

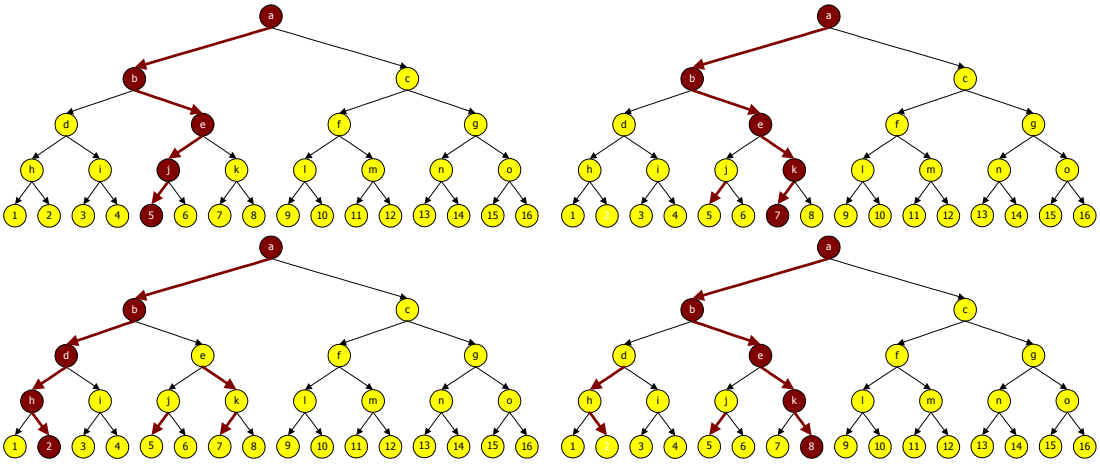


Figure 6-19: Example of locking operations in the tree search micro-benchmark

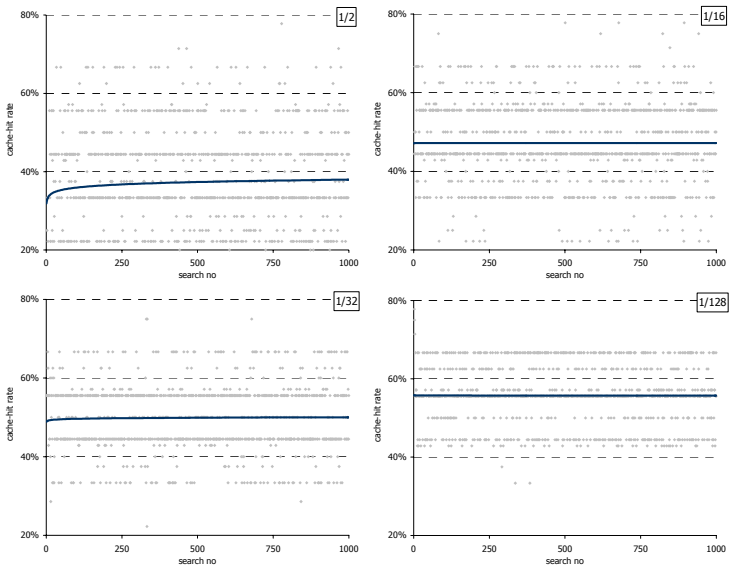


Figure 6-20: Cache-hit rates of object grouping based on access history for the tree search micro-benchmark (binary tree leaves: *1,024*; random objects used as criteria for the search algorithm are selected from the $1/2, 1/16, 1/32, 1/128$ of the set of leaves)

Figure 6-20 presents the results from the execution of the micro-benchmark. A dot in the graph indicates the cache-hit rate achieved for one invocation of the search algorithm. The line represents the trend of the cache-hit rate. As expected, a narrower range of objects from which a random one is chosen results in higher cache-hit rates (the trend lines of Figure 6-20) because more objects are accessed in a recurring manner. The cache-hit rates for the first *50* repetitions of the algorithm are shown in Figure 6-21. Finally, Figure 6-22 presents the evolution of the cache-hit rate as the range from which the random objects are chosen becomes narrower.

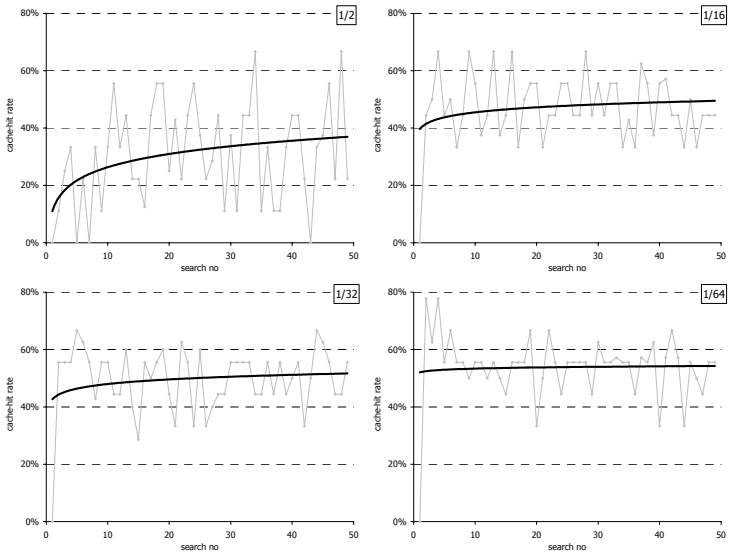


Figure 6-21: Cache-hit rates of object grouping based on access history for the tree search micro-benchmark for the first 50 repetitions of the search algorithm (binary tree leaves: *1,024*; random objects used as criteria for the search algorithm are selected from the $1/2, 1/16, 1/32, 1/128$ of the set of leaves)

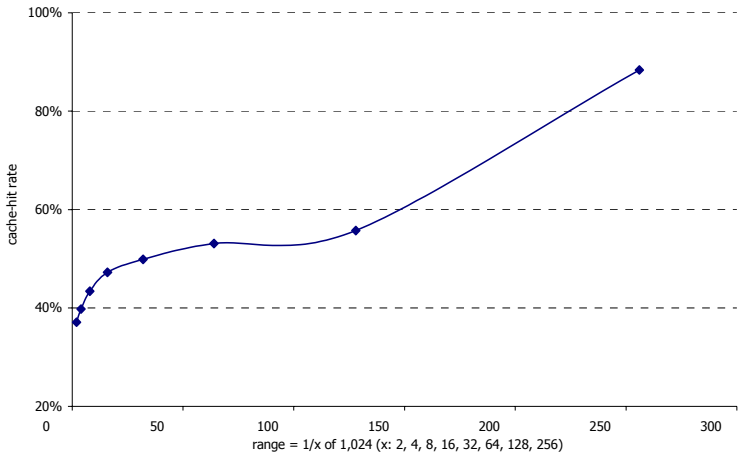


Figure 6-22: Evolution of the cache-hit rate when the range from which the random objects are chosen becomes narrower

The cache-hit rates achieved from the execution of the micro-benchmark demonstrate the applicability of the object-grouping based on access history for applications that exhibit recurring behaviour when accessing objects in memory, such as implementations of search algorithms, web or database servers, numerical applications, etc.

6.6. Applications

After the separate examination of the NIP run-time techniques, the discussion moves to the analysis of the results obtained from the performance evaluation of three applications, with both NIPLTC and NIPDSM activated. All three applications utilise the grouping capable iterative tasklet for identifying parallelism.

In the discussion that follows, 'no caching' indicates that none of the object grouping techniques was used to improve the cache-hit rate. However, when individual objects are cached, even if none of the object grouping techniques is used, temporal locality can still be exploited. According to the NIP entry consistency semantics, a proxy node maintains an object in its cache until it is invalidated (Chapter 4).

6.6.1. Matrix Multiply

Design

The first application to be examined implements a matrix-by-matrix multiplication algorithm. A matrix is regarded as a two-dimensional array. If the dimensions of the matrix are `N` rows by `M` columns, then `M+1` NIP references are required to represent a `Matrix` instance. Since the subscript operator (`operator[]`) is available for NIP references and it offers a similar behaviour to the subscript operator on virtual memory

pointers, it is not necessary to store all the possible NIP references to the matrix elements (`N*M` in total) (Table 6-12).

NIP reference to row		
	[0]	NIP reference to column 0
	[1]	NIP reference to column 1
	.	
	.	
	.	
	[N – 1]	NIP reference to column N - 1

NIP reference to column X (x = 0 ... N – 1)		
	[0]	NIP reference to matrix element
	[1]	NIP reference to matrix element
	.	
	.	
	.	
	[M – 1]	NIP reference to matrix element

Table 6-12: NIP references required for representing the layout of a matrix

```
template<class T>
class Matrix : public NIPShared<Matrix<T> >
{
public:
   Matrix(size_t, size_t);
   ~Matrix();

   typedef T                                MatrixElement;
   typedef NIPRef<MatrixElement>            ColumnRef;
   typedef NIPRef<NIPObject<ColumnRef> > RowRef;

   RowRef row(size_t);
private:
   Matrix();
   RowRef  _rowsRef;
   size_t  _m, _n;
};

typedef Matrix<NIPDouble> MatrixOfDoubles;
```

Code 6-7: The `Matrix` class

The `Matrix` C++ interface is presented in Code 6-7. Since `NIPRef<T>` is considered a primitive type, its instances cannot be allocated directly into the NIPDSM. A wrapper template class (`NIPObject<T>`) is made available by the NIP run-time that may be used when primitive types are to be allocated in the NIPDSM. For the purposes of this test, the elements of the matrix are of type `NIPDouble`, a type provided by the NIP runtime library and which is equivalent to `NIPObject<double>`.

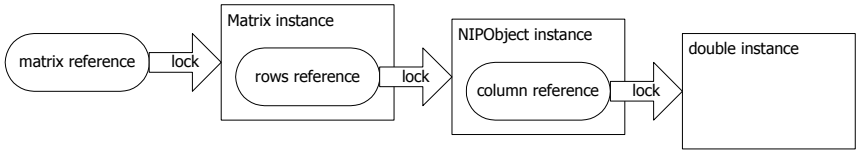


Figure 6-23: Series of locking operations when accessing a `Matrix` element

In order to be consistent with the NIP execution model memory semantics, three NIPDSM locking operations are required before an element of a matrix instance may be accessed (Figure 6-23). The significant run-time overheads of the NIPDSM locking operations (Section 6.3.2 and Section 6.5) overwhelm the computation, as the results presented later in this section demonstrate. A total number of $3 \times N \times M$ lock operations are required when all the elements of a matrix of size $N \times M$ are accessed. The number

of lock operations for the multiplication of two matrices of size $N \times M$ and $M \times L$ is given by Equation 6-2. However, it is reasonable to assume that a compiler can optimise the access to the matrices by reducing the number of locks required. For example, when the elements of an entire column are accessed, the `NIPObject` containing the NIP reference to that column need not be locked every time (second lock in Figure 6-23). In the implementation of the matrix multiplication application that is used here, the number of locks required is reduced and is given by Equation 6-3.

$$2 \times 3 \times N \times L \times M + 3 \times N \times L \qquad \textbf{Equation 6-2}$$

$$5 \times N + 3 \times N \times L \times M + N \times L \qquad \textbf{Equation 6-3}$$

An alternative design of the `Matrix` class could have eliminated part of the run-time overhead and significantly improve performance. The value of the matrix elements rather than their references could be stored in a vector object. Then, only one lock operation would be required when accessing a whole row or column of the matrix (depending on the way the matrix is represented). However, this alternative approach introduces two important problems:

- The manner in which the application can access the matrix object is restricted to either row-wise or column-wise.
- If exclusive access (i.e., a write lock) to an element of the matrix were required, a whole vector would have to be locked. As a result, the application's logical degree of parallelism would have been compromised because not all the elements of the vector could be accessed concurrently. Furthermore, false sharing is introduced. If two or more parallel tasks require concurrent access to different elements of the matrix that belong to the same column (or row), the vector object representing the column (or row) will have to move from one node to another, further compromising performance.

In the performance evaluation of the matrix application that follows, the first design for the `Matrix` class was used (Table 6-12, Code 6-7, Equation 6-3).

Performance Analysis

The NIP version of the matrix multiplication was executed on the SMP using *1*, *2*, and *4* processors. The measured execution times are compared to the sequential C++ version of the same application. The one-processor test shows the significant overheads due to object locking. The observed slowdowns are reduced as the matrix size increases. This is because a number of lock operations ($5 \times N$ in Equation 6-3) are performed outside the

dot product loop of the matrix-by-matrix multiplication. As the granularity of the dot product increases, the relative impact of the run-time overheads due to those lock operations is reduced.

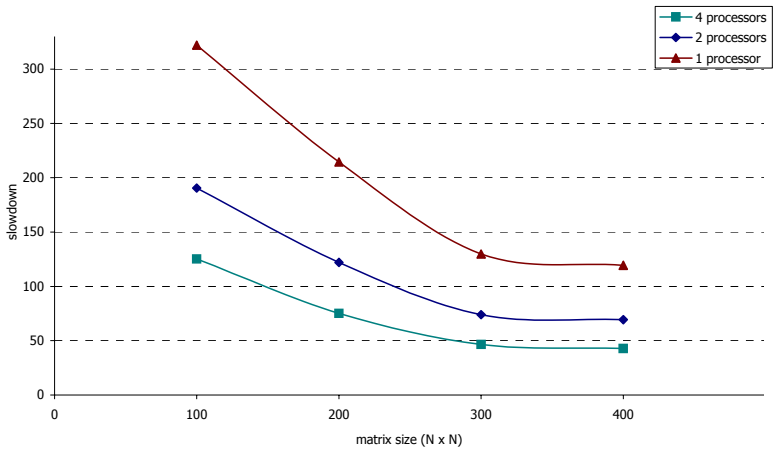


Figure 6-24: Execution slowdowns of the matrix multiplication application on the SMP due to object locking when compared to the sequential C++ version

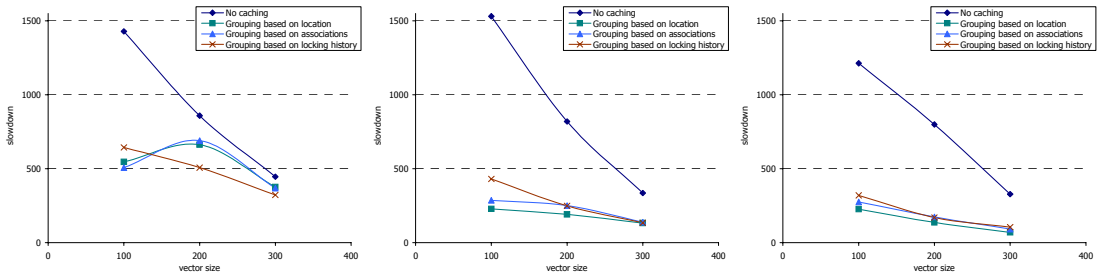


Figure 6-25: Execution slowdowns of the matrix multiplication application on the APP when compared to the sequential C++ version (number of nodes: *2*, *4*, *8*; vector sizes: *100*, *200*, *300*)

Having observed the significant slowdowns on the SMP, the discussion moves to the analysis of the results collected from the execution of the matrix multiplication application on the APP. The performance is expected to be even poorer on the APP due to the additional network related overheads. Indeed, the slowdowns on *2*, *4*, and *8* nodes are extremely high, as presented in Figure 6-25.

When any of the caching techniques is enabled, the performance is dramatically improved, especially for smaller matrix sizes, compared to the execution of the application with 'no caching.' Cache hits due to temporal locality are still observed. All three NIPDSM caching techniques are in turn used for the execution of the application.

When the object grouping based on locking history technique is enabled, NIPDSM records the locking pattern as the elements of the matrix are accessed and it uses the information to improve caching. The locking access pattern can also be explicitly defined through object associations. Each element of a matrix is associated with the next element in the same matrix that is to be locked during the dot product loop.

Even with the caching techniques enabled, the application performs very poorly when compared to the sequential C++ version. This may be attributed to the following factors:

- A task has to suspend whenever an element of the matrix is accessed on a secondary node. Since the grouping-capable tasklet is used, the number of dot product calculations that take place on the secondary nodes is high. Therefore, the computation is overwhelmed by task suspensions and proxy requests.
- Even when caching is enabled, a task has to suspend on every write lock operation. The default behaviour of the NIPDSM is not to attempt to improve cache-hit rates for write lock operations by write locking groups of objects. This is because it is assumed that the read/write object-locking ratio in applications is high. If a cache block was filled with write locked objects in response to a write proxy request, a great number of invalidations would have been required by subsequent locking requests that are initiated on different nodes.
- The sequential version of the application can better utilise the processor cache because of the small size of its executable image.

The performance of the application on the APP is also influenced by the distribution of work on the parallel platform. As discussed in Section 6.4.1 (page 118), the number of lazily created tasks on a node does not necessarily determine the actual work performed at that node when the grouping-capable iterative tasklet is used. However, the number of dot product computations executed on a node can be determined through the write lock operations on that node (Figure 6-26). A write lock operation only takes place once for every dot product computation (i.e., when the result is stored to appropriate element of the results matrix). The positive effect of the caching operations is illustrated by the even distribution of work around the platform.

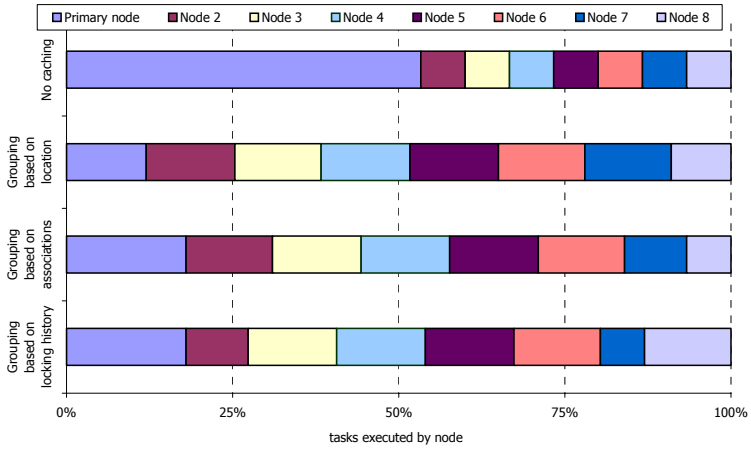


Figure 6-26: % of tasks executed at each node (vector size: *250*x*250*, number of nodes: *8*)

When the object grouping based on location caching technique is used, a number of proxy invalidations take place. This is because objects belonging to the results matrix are cached on secondary nodes due to a read lock operation on a spatially adjacent object. On page-based schemes, the problem of false sharing would have arisen in this case because different type of access (i.e., read or write) would have been required for objects placed in the same page. NIPDSM avoids the problem by only invalidating the required objects. This is indicated by the very small number of object invalidations that take place during the execution of the matrix multiplication application when the object grouping based on location caching technique is enabled (Table 6-13).

		Matrix size		
Number of nodes		**100**	**200**	**300**
	2	0%	0%	0%
	4	0.045%	0.002%	0.001%
	8	0.177%	0.019%	0.004%

Table 6-13: Percentage of cached objects that were invalidated

The analysis of the performance results in this section demonstrates the benefits of both the NIP lazy task creation and NIP distributed shared memory system caching techniques. The work in the matrix multiplication application is lazily distributed across the parallel platform and the caching techniques significantly improve performance. However, due to the high NIPDSM locking operations on the APP and the memory intensive nature of the application, no speedups could be achieved. The affect of the NIPDSM locking operations on performance is further discussed at the end of this chapter.

6.6.2. Barnes-Hut

Design

The Barnes-Hut application is the implementation of a simulation algorithm for a collisionless system of bodies (*e.g.*, molecules, planets, stars, etc.). A new position for every body in the system is calculated during each step of the simulation. The new body position is determined by the forces applied to it by the other bodies in the system, its velocity, its acceleration, etc. A step of the simulation requires that the interactions between all the bodies in the system be considered. The Barnes-Hut algorithm reduces the complexity of the simulation from $O(N^2)$ to $O(N\log N)$ by taking into consideration the distance of the bodies from each other. Groups of bodies that are 'far enough' from a particular body are regarded as one (Barnes and Hut 1986). The algorithm is split into three distinct steps:

- The Barnes-Hut algorithm is based on a quad- or oct-tree data structure, for the 2D and 3D cases respectively. The trees are created by the recursive decomposition of the space containing the bodies into four squares (2D space) or eight cubes (3D space) until each node in the tree contains at most one body. The tree nodes that contain a body or other tree nodes are called cells.
- Once the tree has been created, the interactions between the bodies in the system are calculated. For each body in the system, the tree is traversed and the distance from each cell in the tree is examined. If the distance between a cell and the body is within the critical radius (i.e., 'far enough' criterion), the children of the cell are traversed. If the distance from the cell is long enough, the interaction between the body and the cell can be calculated.
- Once all the interactions have been calculated, the new position of the bodies may be determined. The tree is destroyed before the algorithm can be repeated starting from the first step.

The Barnes-Hut algorithm is often used in DSM related works due to its irregular memory access pattern. The dynamically created tree structure exposes the limitations of the caching techniques because the benefits due to spatial or temporal locality, in the way memory is accessed, are limited.

The NIP version of the Barnes-Hut algorithm uses NIPDSM objects to represent bodies and cells. The bodies of the system are stored in a vector throughout the simulation. An oct-tree is constructed during the first step of the simulation (3D space). The nodes of the tree are associated with their children in an attempt to improve the access times when the object grouping based on associations caching technique is enabled. The second step of the algorithm requires that the interactions of every body in the vector be calculated. The potential parallelism in the second step is exposed through a grouping-capable iterative tasklet. The tasklet used in the current implementation inlines groups of iterations, rather than one, at a time. Finally, the tree is destroyed during the third step. The first and the third step always take place on the primary node since any potential parallelism in them has not been exposed. As a result, all the bodies that are cached on secondary nodes are invalidated during the third step because their state is updated (i.e., the result from the interactions is used to determine the new position).

The initial position of the bodies is randomly selected before the first step of the simulation and it is the same for all the tests. The oct-tree is different for every step of the simulation as the bodies change positions. The total number of locks required during

every step dramatically increases for larger number of bodies. For example, an average number of *~468,000* read and *~154,000* write locks per step are required for *512* bodies. The corresponding number of locks for *8,192* objects are *~61,000,000* and *~20,000,000* respectively.

Performance Analysis

The performance characteristics of the Barnes-Hut application are recorded separately for each step of the simulation. Due to the great number of locking operations required for every step of the simulation, the run-time overheads are dramatically increased resulting in significant slowdowns. The run-time overhead is measured to be approximately *11%* of the execution time of the sequential version (number of bodies: *512*, *1024*, *2048*, *4096*). However, when the application is run on more than one node, the cost of accessing non-cached objects affect performance to a greater extent, as is shown below.

Figure 6-27 presents the measured slowdowns on *2*, *4*, and *8* nodes for different number of bodies. In Figure 6-28, the same results are shown in finer detail only for the larger number of bodies. When the granularity of the computation is fine (i.e., smaller number of bodies), the slowdown on a larger number of nodes is higher. This is attributed to the overwhelming runtime costs of object replication to secondary nodes when compared to the total computation time. As the granularity increases, the NIPDSM related overheads become comparably less evident.

The cache-hit rates achieved from the execution of the Barnes-Hut application on the APP are presented in Figure 6-29. The cache-hit rate is exceptionally high because of the temporal locality in memory access that the application exhibits. During a body-to-body or body-to-cell interaction, every body or cell is locked a great number of times. Higher rates are achieved when object grouping based on location is enabled.

It would have been expected that higher cache-hit rates would yield improved application performance. However, this is true only for larger number of bodies (Figure 6-28). When Figure 6-27 and Figure 6-29 are compared for smaller number of bodies, it is clear that the significant difference in cache-hit rates does not result in performance improvement over the execution of the application when none of the caching techniques is enabled. In order to account for this lack of improvement, the memory access pattern of the Barnes-Hut application must be considered.

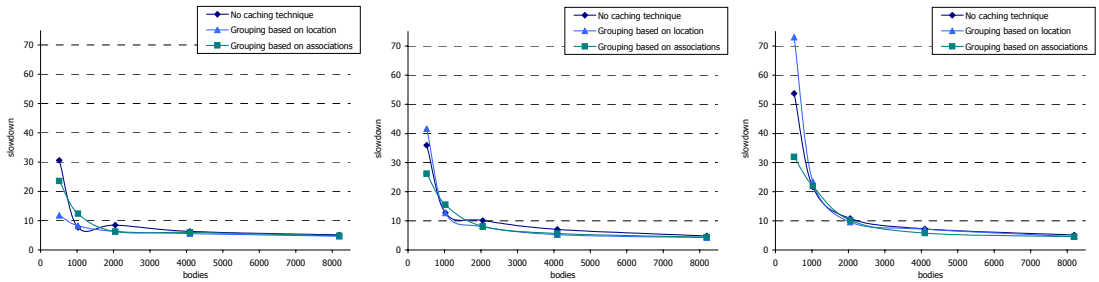


Figure 6-27: Slowdowns of the NIP version of the Barnes-Hut application on the APP for different caching techniques (each graph represents a different number of nodes: *2*, *4*, *8*; number of bodies: *512*, *1024*, *2048*, *4096*, *8192*; object and cache block sizes are presented in Table 6-14)

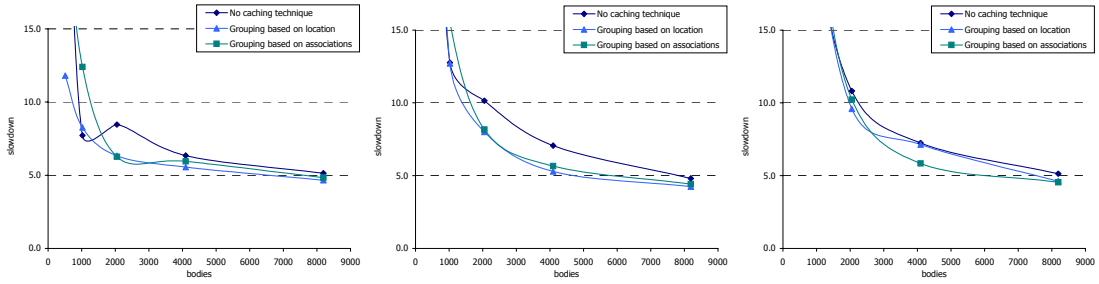


Figure 6-28: Slowdowns of the NIP version of the Barnes-Hut application on the APP for different caching techniques (each graph represents a different number of nodes: *2*, *4*, *8*; number of bodies: *2048*, *4096*, *8192*; object and cache block sizes are presented in Table 6-14)

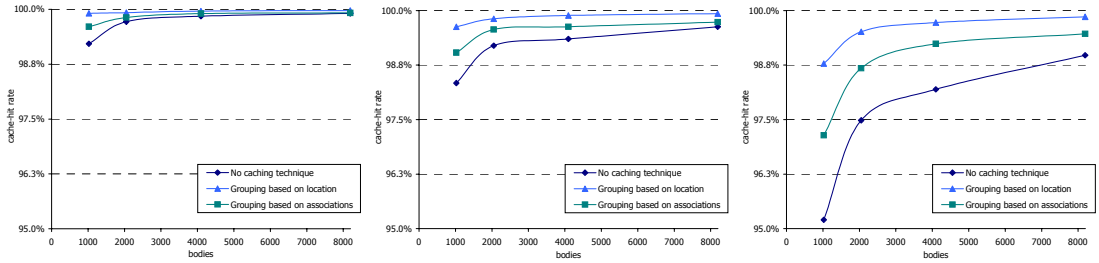


Figure 6-29: Cache-hit rates of the NIP version of the Barnes-Hut application on the APP for different caching techniques (each graph represents a different number of nodes: *2*, *4*, *8*; number of bodies: *512*, *1024*, *2048*, *4096*, *8192*; object and cache block sizes are presented in Table 6-14)

Body NIPDSM object	128 bytes
Cell NIPDSM object	144 bytes
NIPDSM cache block	2,048 bytes

Table 6-14: Sizes of the `body` and `cell` objects and the NIPDSM cache block

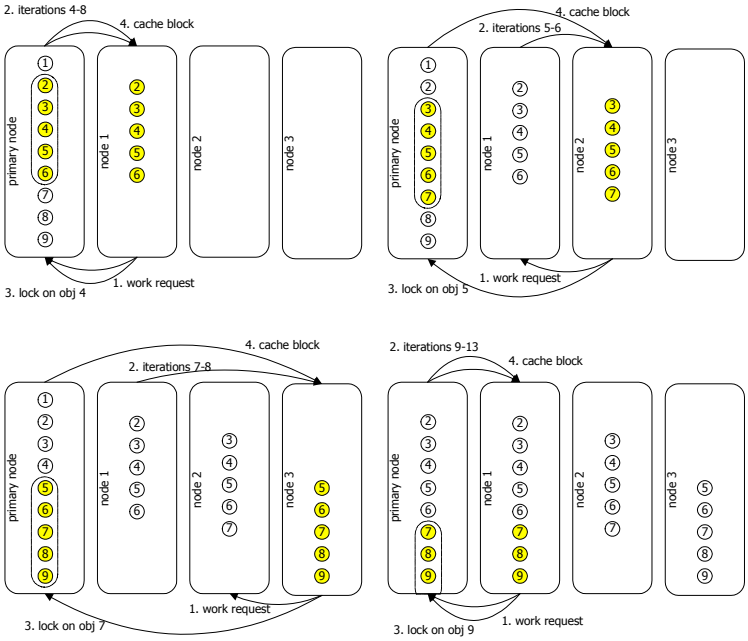
The bodies in the simulated space are created together during the initialisation of the application, thus being spatially adjacent to each other. As mentioned earlier, the parallel computation of the second step is represented by an iterative tasklet. Each iteration calculates the interactions of one body. The calculation involves the traversal of the oct-tree and the evaluation of the necessary body-to-body or body-to-cell interactions. At the end of each interaction, the state of the body is updated and, hence, a write lock operation is required. However, the write lock operation on an object is always preceded by a series of read locks because its state is read a number of times before it is altered.

This access pattern and the caching technique that is used influence the performance of the application, as explained next.

When an iteration is executed on a secondary node, it will have to block when the body for which the interactions are calculated is not cached. When the object grouping based on location caching technique is enabled, the locking operation on a body causes the adjacent objects (bodies or cells) to be cached as well. As the task waits for the requested object to be fetched, a new local task may be created to inline part of the available iterations. Additionally, another node may steal a group of iterations. In either case, it is very likely that the new task created will have to suspend on a lock operation because the required body for the first lock of the iteration is not cached.

Figure 6-30 illustrates how most of the bodies in the simulated space are quickly replicated in the parallel platform, when the object grouping based on location caching technique is enabled, and why a high cache-hit rate is observed. The large number of replicated bodies, though, results in a higher rate of invalidations (Figure 6-31), which compromises performance for smaller granularities of the entire computation (i.e., smaller number of bodies). The example of Figure 6-30 is only a simple demonstration of the NIP-related operations that take place. The complexity of the actual execution behaviour of the application is much greater due to the combination of load balancing, NIPLTC, and NIPDSM.

In addition to the above discussion, it should also be noted that the recursive way in which the oct-tree is constructed results in cells at the top part of the tree being spatially adjacent, further contributing to the increase in the cache-hit rate.



Example description - (Four nodes are used to simulate a system of 40 bodies. The cache block is assumed to accommodate a maximum of five objects.) Node 1 requests work from the primary node (step 1). When the work arrives (step 2), node 1 attempts to acquire a read lock on the body of the first iteration. This results in a proxy request being sent to the primary node (step 3) and a cache block full of bodies is returned (step 4). In the mean time, or even after the cache block has arrived, node 2 steals some work from node 1. As a result, the copies of another group of objects will be cached on node 2. In the same manner, objects will be cached on node 3. As the execution of the calculations for body 4 finishes on node 1, the active task will have to be suspended until the stolen tasks by nodes 2 and 3 have been completed. A new task is created from the primary node 1 to keep node 1 busy causing yet another group of objects to be cached.

Figure 6-30: Caching of bodies on a parallel platform of four nodes when the object grouping based on location technique is enabled

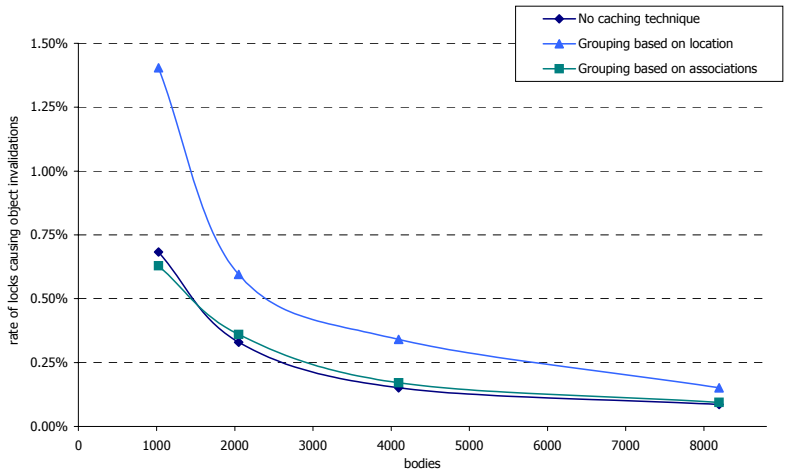


Figure 6-31: Locks causing object invalidations from the execution of the Barnes-Hut application on *8* nodes (number of bodies: *1024*, *2048*, *4096*, *8192*; object and cache block sizes are presented in Table 6-14, page 143)

Figure 6-31 suggests that the object invalidation rates are almost identical when no caching technique is used, and when object grouping based on associations is used, despite their different cache-hit rates (Figure 6-29, page 143). In both cases, a lock operation on a non-cached body results in only that body being fetched. This is because in the former case, no object grouping takes place at all and, in the latter case, object

grouping takes place only when elements of the oct-tree are accessed, since only the cells in the oct-tree contain associations with bodies.

As explained in Section 6.5.2 (page 131), the algorithm used by NIPDSM to group objects based on the graph of associations is breath-first and it does not match the tree depth-first traversal that is used in the Barnes-Hut algorithm. This contributes to the lower cache-hit rate of object grouping based on associations caching technique.

In the discussion up to this point, the effect of the caching techniques on the distribution of work in the parallel platform and the granularity of the lazily created tasks were not considered. If the write lock operations on bodies are regarded as a measure of the work executed on a node, a view of the load distribution can be generated (Figure 6-32).

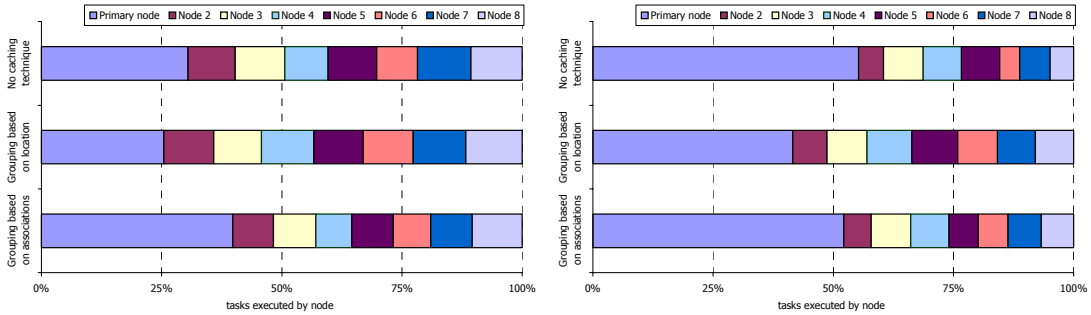


Figure 6-32: Write locks executed on each node during the execution of the Barnes-Hut application on *8* nodes (each graph represents a different number of bodies: *1024*, *8192*; object and cache block sizes are presented in Table 6-14, page 143)

The graphs of Figure 6-32 suggest that a very large share of the overall computation is executed on the primary node. This is due to the benefits of accessing immutable objects at the manager node (Table 6-10, page 128). The cells of the oct-tree are constructed as immutable objects since, once initialised, their state is not altered. The primary node does not suffer from the overheads of all the locking operations on cells. Therefore, the interaction calculations are executed significantly faster. Not all of the work that is executed on the primary node, however, can be attributed to iteration inlining from the original tasklet. Since the secondary nodes suffer from the overheads of remotely accessing objects, the primary node manages to steal back some of the iterations that were stolen from it and executes them locally.

Unfortunately, due to the dynamic nature of the NIP run-time and the use of a grouping-capable tasklet, it is not possible to determine the exact granularity (i.e., the number of iterations) of each lazily created task. However, the total number of lazily created tasks gives an indication of the granularity, as a smaller number of lazily created tasks suggests more inlining operations and, therefore, coarser granularity (Table 6-15).

	Number of bodies				
	512	**1024**	**2048**	**4096**	**8192**
No caching techniques	14.0%	5.7%	4.5%	2.7%	1.6%
Object grouping based on location	10.7%	4.6%	3.7%	3.7%	2.2%
Object grouping based on associations	10.2%	4.1%	4.2%	3.6%	1.7%

Table 6-15: Percentage of lazily created tasks per step out of the maximum possible (number of nodes: *8*, object and cache block sizes are presented in Table 6-14, page 143)

On most page-based DSM systems, the kind of memory access pattern that was described above would have resulted in page-thrashing due to false sharing, as concurrent access to more than one body in the same page would not be possible. NIPDSM avoids the thrashing problem for object groups because of the fine granularity of the sharing unit. However, during the second step of the algorithm the state of an object may be accessed both by read and write methods, causing false-sharing at the object-level. Although this is not as computationally expensive as page-thrashing would have been, it still introduces a significant overhead. In Chapter 7, the concept of a locking technique that allows distinct parts of the same object to be locked in different access modes is outlined.

The implementation of the Barnes-Hut application used in these experiments was not optimal. It was assumed that an implicitly parallel programming language compiler generated the code. The only optimisation used was the one described in Chapters 4, where consecutive method calls can be enclosed within on pair of lock/unlock operations. However, it should be reasonable to assume that a compiler could do better than that. A body is read locked and unlocked for every cell in the oct-tree that is accessed during the second step because the distance between the body and the cell must be calculated. The body data members that are accessed during this calculation remain constant throughout the second step of the algorithm. Based on this observation, two optimisations could be introduced:

- Since the body is unlocked between body-to-cell distance calculations, a write lock on another node might invalidate the cached copy (i.e., object false-sharing). The next read lock operation would cause a cache-miss. If the body was split to two parts, one that is accessed by read-only methods and another that is accessed only by write methods, this unnecessary invalidation could be avoided.
- Given that the same part of a body's state is read for every body-to-cell and body-to-body interaction, it would be a significant improvement if the locking

operations on the body were avoided. A copy of the required state is obtained and used for subsequent calculations.

The efficiency of the application is improved due to the introduced optimisations and speedup was obtained when compared to the sequential C++ version of the Barnes-Hut application (Figure 6-33).

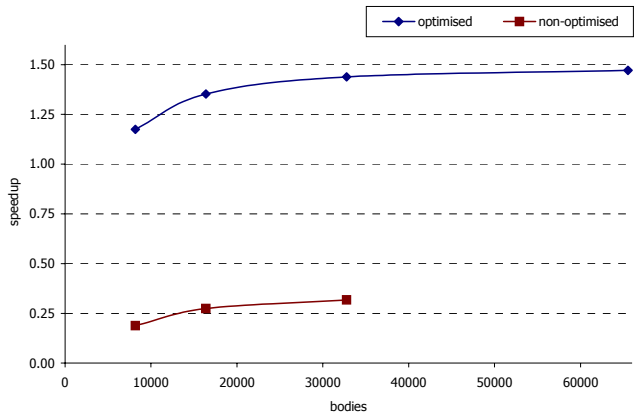


Figure 6-33: Speedup improvement of the Barnes-Hut application on *8* nodes with the object grouping based on location caching technique enabled (number of bodies: *8192*, *16384*, *32768*; *65536*; object and cache block sizes are presented in Table 6-14, page 143)

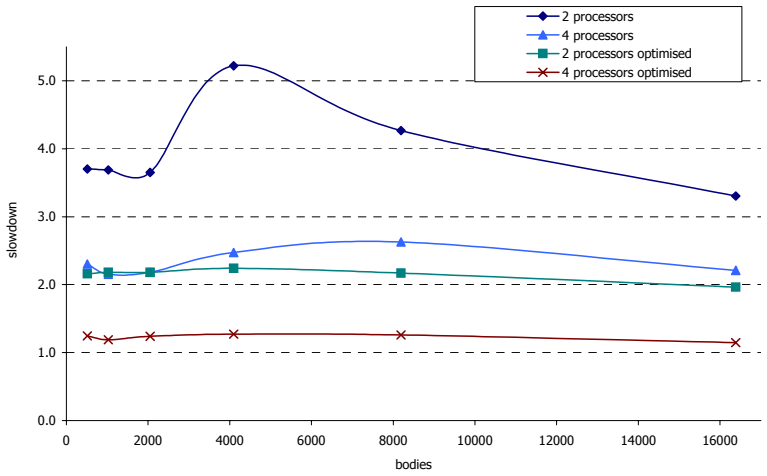


Figure 6-34: Slowdowns of the Barnes-Hut application on the SMP workstation (number of bodies: *512*, *1024*, *2048*, *4096*, *8192*, *16384*, *32768*)

Finally, the slowdowns of the Barnes-Hut application on the SMP workstation are presented in Figure 6-34. The optimised version is the one described above that reduces the number of locks required. Although the execution times do not suffer from the overheads of cache misses, the achieved performance is poor. This is attributed to the following reasons:

- Since bodies are mutable objects, a locking operation on a body always requires a mutex to be acquired. The overhead of accessing a mutable NIPDSM object is presented in Table 6-10 (page 128).

- More than one thread may attempt to read the state of the same body. Although both will be granted access to the state of the body, NIPDSM only allows one thread to update the object representation of that body (i.e., the number of local read locks must be updated). As a result, the execution of a thread may be blocked until exclusive access to the object representation can be granted.
- The granularity of each iteration that calculates body-to-body and body-to-cell interactions is relatively fine. For example, the average granularity of one iteration of the sequential C++ version is *4msecs* for *16,384* bodies. Therefore, the NIPDSM related costs overwhelm the computation.

6.6.3. Travelling Salesperson Problem

Problem Description and Implementation

The final application used in the performance evaluation of the NIP run-time library is the travelling salesperson problem (TSP). The definition of the problem is as follows: *Given a set of cities and travelling costs for each pair of cities, find the best route visiting each city only once.* The problem considered here is asymmetric, which means that the travelling cost from city *i* to city *j* is not assumed to be the same as the cost from city *j* to city *i*.

```
Route       bestRoute
CostsMatrix costs

routeCostForCity(rootCity)
  Route route
  createRoute(rootCity)
  if (route.cost < bestRoute.cost)
    bestRoute = route

createRoute(rootCity, route)
  for city = 1 to N except rootCity
    if city not visited
      route.cost +=
            costs(rootCity, city)
      createRoute(city, route)

main()
  for city = 2 to N
    calculateRouteCostForCity(city)
```

Code 6-8: Pseudo code for the sequential version of TSP

```
RouteNIPDSM  bestRoute
CostsMatrix  costs

routeCostForCity(rootCity)
  Route route
  createRoute(rootCity)
  bestRoute.lockRead()
  bestRouteTmp = bestRoute.cost
  bestRoute.unlock()
  if (route.cost < bestRouteTmp.cost)
    bestRoute.lockWrite()
    if (route.cost < bestRoute.cost)
      bestRoute = route
    bestRoute.unlock()

createRoute(rootCity, route)
  for city = 1 to N except rootCity
    if city not visited
      costs(rootCity, city).lockRead()
      route.cost +=
            costs(rootCity, city)
      costs(rootCity, city).unlock()
      createRoute(city, route)


main()

NIPIterativeTasklet<routeCostForCity>
                          route(2, N)
  route.waitOrInline()
```

Code 6-9: Pseudo cost for the NIP version of TSP

For the purposes of this thesis, a brute force algorithm for solving the problem was utilised. The costs of all possible routes, given an initial city, are calculated. When the

calculated cost for a particular route is found to be less than the current minimum, the current best route and the associated minimum cost are updated.

A matrix with the travelling costs between each pair of cities is created. The state of the matrix elements is not altered once initialised (i.e., the elements are immutable objects). Only the current best route and minimum cost are updated when necessary. The two versions of TSP in pseudo code are presented in Code 6-8 and Code 6-9. In the NIP version of TSP, if the calculated cost for a route is found to be less than the currently best route, another comparison must take place after the write lock has been acquired. This is required in order to guarantee the correctness of the result. Finally, the first city from a set of N is considered as the start of the route.

Performance Analysis

The complexity of the TSP algorithm does not allow large instances of the problem to be considered. The cost of a total number of $(N-1)!$ routes must be calculated for N cities. However, the smaller instances of TSP are sufficient to study the behaviour of the NIP version on the APP.

On the APP, the route costs matrix will be progressively cached on secondary nodes through locking operations. Due to the small number of cities, only few locks are required for the whole matrix to be cached on a particular secondary node, when object grouping based on location is used. This is because the elements of the matrix are spatially adjacent to each other.

Data structure holding best route and minimum cost objects	232 bytes
Cost between two cities object	32 bytes
Cache block size	2,048 bytes

Table 6-16: Sizes of the `body` and `cell` objects and the NIPDSM cache block

Figure 6-35 presents the speedups achieved when compared to the sequential C++ version of the TSP application on the APP. An optimised version of the application is also presented. In the optimised version, the locking costs for accessing the elements of the matrix are avoided because a copy of the matrix is distributed to all the secondary nodes at the start of the computation. Access to the best route and minimum cost objects, though, still requires a NIPDSM locking operation. The optimised version is presented only as a demonstration of the impact on performance that the NIPDSM locking operations have. Since the majority of the locks are avoided in the optimised version, the achieved speedups are very good (Figure 6-35).

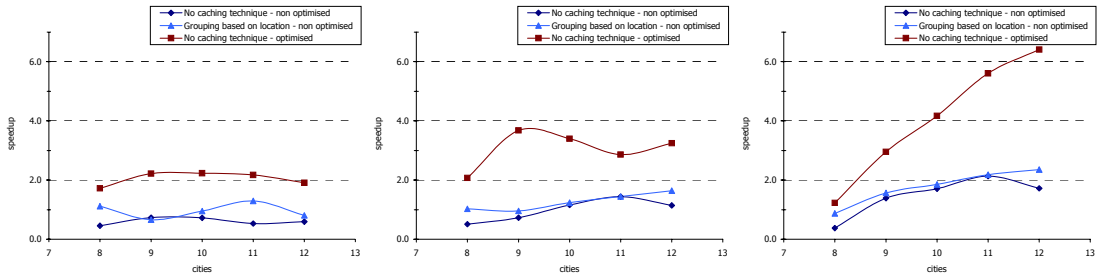


Figure 6-35: Speedup the NIP version of the TSP application on the APP (each graph represents a different number of nodes: *2*, *4*, *8*; number of cities: *8*, *9*, *10*, *11*, *12*; object and cache block sizes are presented in Table 6-16)

For the non-optimised version of TSP and when object grouping based on location is enabled, performance is only slightly improved over the execution of TSP with none of the caching techniques enabled. This is attributed to the high temporal locality in memory access that TSP exhibits. The cache-hit rates in both cases approaches *100%*. Once the immutable objects of the route costs matrix are cached, they are accessed a great number of times. Furthermore, since the great majority of the total number of object locking operations are read locks on immutable objects, the chances of parallel tasks attempting to access the same object at the same time are minimised. This results to a very small percentage of locking operations causing object invalidations, as presented in Figure 6-36.

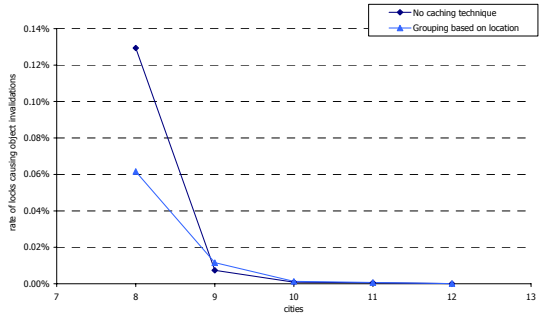


Figure 6-36: Rate of invalidations in the execution of the NIP version of TSP on the APP (number of nodes: *8*; number of cities: *8*, *9*, *10*, *11*, *12*; object and cache block sizes are presented in Table 6-16)

Only the cached copies of the object storing the best route and its cost that may be kept on secondary nodes need to be invalidated during the execution of the TSP application. The percentage of object invalidations decreases as the granularity of computations increases. This is because the increase in the total number of write locks required for the entire computation is disproportional to the increase in the read locks required. For example, when the number of cities is *8*, *6* write locks and *32,439* read locks take place respectively. When the number of cities is *12*, the number of write locks that take place is *34* while the number of read locks is *256,927,023*.

6.7. Discussion

The performance results presented in this chapter demonstrate the effectiveness of NIP lazy task creation technique in efficiently exploiting the identified parallelism in application at run-time. Furthermore, the benefits of the NIPDSM caching techniques were demonstrated. Other works that used the NIP run-time library also demonstrated the advantages of NIPLTC (Johnson 2000) and NIPDSM (Kelly 2000; Webber 1998) for certain types of parallel object-oriented applications, mainly not memory intensive in nature.

In the evaluation of NIPDSM presented in this chapter, it has not been possible to achieve promising performance results for two of the applications that were evaluated: matrix multiplication and Barnes-Hut. The overwhelming communication costs in combination with some implementation related drawbacks resulted in poor performance on the APP.

The slowdowns observed in the execution of applications, especially on the SMP workstation, suggest that the requirements of the NIP execution model memory semantics are too strict. The NIP execution model memory semantics defined that a lock be acquired on an object before a method can be called on it. For the implementation of these semantics in the NIPDSM system, a mutex synchronisation construct is associated with each object. As a result, a locking operation on a mutable object requires at least one pair of acquire and release operations on the mutex. This introduces a significant overhead when applications access objects in memory. Chapter 7 discusses further the effect that the design of the NIP execution model memory system has on the performance of NIP applications and the applicability of implicit object-based locking in the field of parallel object-oriented computing.

Next, NIPDSM implementation related issues that further affect performance are identified.

Load Balancing and Communication

The load balancing and communication services are not optimal. As described in Chapter 5, simple implementations of these two services were provided in order to have a fully functional run-time environment. The most important issues related to these two services are:

- When a cache-miss occurs, a new task is always created—from either a local or a remote tasklet—if there are no active tasks at that node. However, as the load

balancing service depends on the underlying operating system for task scheduling, when the object is finally cached, the suspended task will resume even if there other active tasks running. This results in context switching, which is one of the problems that NIPLTC tries to overcome through the use of tasklets. Although an upper limit on the number of tasks that can simultaneously exist on a node, either in suspended or active state, is imposed in order to reduce the potential overheads, a better solution is required.

- For every message exchange between two nodes, two memory-copy operations take place before the appropriate operating system call is made, which usually results in yet another memory-copy operation. Furthermore, the current implementation is based on TCP/IP, which further reduces communication efficiency.

Lack of Object Distribution

In the current implementation of the NIPDSM, the manager node for an object is always the node on which the task creating the object is running. This introduces a bottleneck when a great number of objects are created together by one task, as in the case of the matrix multiplication and Barnes-Hut applications, because that node is overwhelmed with NIPDSM related requests.

CHAPTER 7

CONCLUSIONS AND DISCUSSION

In this chapter the conclusions of the thesis are drawn. The motivation for this research work and the summary of the introduced run-time techniques in the areas of lazy task creation and distributed shared memory define the context for the discussion. Additionally, the findings from the performance evaluation of the benchmark applications are considered when suggesting possible optimisations in the current design and implementation of the NIP run-time. The thesis concludes with an outline of possible avenues for future research.

7.1. Object-Oriented Parallel Computing

Chapter 1 identifies the need for a new approach to parallel programming. Through the analysis of data from the top500 list of supercomputers (TOP500 List 2000), a trend is observed towards massively parallel architectures that are based partly or entirely on commodity hardware. Software developers should not be burdened with the task of managing the high degree of parallelism that is made available to them through such hardware architectures. Instead, it is suggested that software developers should only have to concentrate on the algorithmic issues of their applications by using a programming paradigm that, although it does not expose any hardware architecture characteristics, it allows for parallelism in applications to be implicitly exploited. Additionally, good software engineering practices should be supported.

To meet this challenge, the NIP programming model is proposed in Chapter 2. The key features of the introduced model are object-orientation and implicit parallelism. These two characteristics of the NIP programming model are considered by many experts as essential for the future of parallel computing (Almasi and Gottlieb 1994; Culler and Singh 1999; Skillicorn and Talia 1998; Sterling et al. 1995). Previous work on the UFO language has demonstrated that object-orientation, state, and implicit parallelism may be successfully combined at the language level (Sargeant 1993).

While the NIP programming model describes a software development methodology, the NIP execution model defines the semantics to which an execution environment should adhere in order to host applications implemented through the proposed methodology. The main characteristics of the NIP execution model are illustrated through an abstract machine (Figure 7-1): implicit exploitation of parallelism and automatic management of computational resources (parallelism manager), object-based memory.

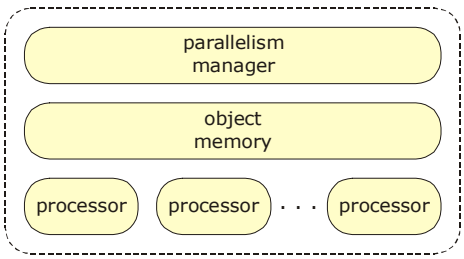


Figure 7-1: The major components of the abstract machine as suggested by the NIP execution model

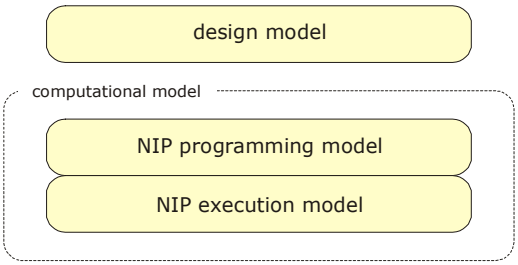


Figure 7-2: The parallel computing paradigm with the NIP programming and execution models

Figure 7-2 illustrates the parallel computing paradigm as proposed by this thesis. The NIP programming and execution models replace the corresponding layers of the parallel

computing paradigm presented in Chapter 2. This thesis focuses on issues related with the bottom layer of Figure 7-2, the design and implementation of a run-time environment that adheres to the semantics of the NIP execution model.

7.2. Run-time Support

The main research goal of this research work has been the investigation of run-time techniques to support the parallel execution of applications developed with an object-oriented, implicitly parallel programming language. The NIP lazy task creation technique (Chapter 3) and the NIP distributed shared memory (Chapter 4) were designed and implemented as part of the NIP run-time environment (Chapter 5), which is available as a C++ library.

NIP Lazy Task Creation

The tasklet is a new construct that was first introduced by Watson (Watson 1996) to represent the parallelism in an iterative computation (Watson and Parastatidis 1999a; Watson and Parastatidis 1999b; Watson and Parastatidis 1999c). In this thesis, an object-oriented approach to its design and implementation was adopted. The patterns of parallel execution that can be represented by a tasklet are extended to include function and recursion parallelism.

The tasklet construct is used by the NIP lazy task creation technique. During the execution of an application, parallel tasks are lazily created from existing tasklets as computational resources become available. The performance evaluation of a number of micro-benchmarks (Chapter 6) demonstrates the efficiency benefits from representing parallel computations using tasklets. It is also shown that applications with a very high degree of parallelism can almost optimally utilise the available processing resources even when the degree of parallelism offered by the hardware is orders of magnitude lower.

NIP Distributed Shared Memory

NIPDSM provides an implementation of the NIP execution model memory semantics. It is an all-in-software, object-based, distributed shared memory system. Central to the design of the NIPDSM is the NIP entry consistency model, a variation of the original entry consistency model (Bershad and Zekauskas 1991). The NIP entry consistency model defines the requirements for the management of the object locking and replication in a parallel system. It provides a view of the available memory as defined by the NIP execution model semantics.

An object in the NIPDSM is implicitly associated with a lock, which must be acquired before any method can be called. The methods are classified as read or write, depending on whether the state of the object, for which they are called, is altered. The object's private lock is acquired in non-exclusive or exclusive mode respectively. The fine granularity of locking avoids the problem of false-sharing, from which page-based systems suffer. In the current implementation of the NIPDSM, significant run-time overheads are introduced to the execution of applications, as the performance results of Chapter 6 demonstrate. Access to a NIPDSM object is orders of magnitude more expensive than it is to a C++ object in virtual memory.

An important characteristic of the NIPDSM is the overlapping of communication and computation through cooperation with the NIP load balancing service. When a task blocks on a lock operation, the load balancing service is informed and a new task is created through NIPLTC, if there is not one already available to use the freed computational resources.

A key feature of the NIPDSM is its caching mechanism. The implementation of the NIP entry consistency model, through proxy nodes, improves the memory access time for applications that repeatedly call methods on the same object within short time periods (i.e., temporal locality in memory access). In previous object-based DSM systems, it was not possible to take advantage of memory access patterns other than temporal locality in order to improve cache-hit rates and, hence, memory access times. Through NIPDSM, the access patterns in object-oriented applications are explored in order to improve the performance of applications:

- Applications that exhibit spatial locality in memory access can take advantage of the object grouping based on location caching technique. Spatially adjacent objects in NIPDSM are cached as groups. NIPDSM does not suffer from the problem of false-sharing, unlike page-based DSM systems. Objects that cannot be locked are not included in a group.
- A problem with DSM systems has been their inability to improve the access times of applications that use dynamically created data structures, such as trees, lists, etc. NIPDSM uses information about the relation between objects in the memory to improve cache-hit rates. The object grouping based on associations caching technique can also be used in applications to explicitly define memory access patterns (*e.g.*, the order in which the elements of a matrix are accessed).

- The object grouping based on access history technique improves performance in applications that exhibit a recurring memory access pattern. The order in which objects are locked is recorded at run-time and used when grouping objects together.

The performance evaluation of the micro-benchmarks in Chapter 6 demonstrated the performance gains and potential benefits from the object grouping techniques. However, despite the improved execution times due to object grouping, the parallel execution of applications suffers from the great overheads of the NIPDSM operations.

The NIP execution model memory semantics define that objects be implicitly locked before methods can be called on them (Chapter 2). This removes the burden of dealing with concurrency related issues in memory access from programming language compilers or application developers. Unless an implementation can be demonstrated, which satisfies the memory semantics of the NIP execution model without incurring the overheads of NIPDSM, the results presented in this thesis suggest that a memory system based on implicit locking of objects is not an effective design for a run-time system to support parallel object-oriented computing. In the absence of schemes which support coarse-grained locking of objects, the ratio of computation to locking is such that favourable performance cannot be achieved.

7.3. Potential NIPDSM Enhancements

A number of possible enhancements that may be considered in future implementations of the NIPDSM are presented here. The goal of the enhancements is performance improvements and/or introduction of new features.

Mask-based Locking (Multiple Writers)

The performance evaluation of the Barnes-Hut application (Chapter 6) illustrated a known problem with the NIPDSM and other object-based DSM systems. Although NIPDSM does not suffer from false-sharing when separate objects are accessed, concurrent access to different data members of the same object is not allowed, when at least one of those accesses requires an exclusive lock.

In order to deal with the problem of false-sharing, some page-based DSM systems allow multiple-writers to access different parts of the same page. Multiple copies of a page are maintained. A page-merging algorithm that combines the changes in the copies of a page is necessary. A similar approach could be adopted in the NIPDSM for objects.

However, a more elegant solution, which is achieved through the extension of the consistency protocol semantics rather than by additions to the implementation, could be investigated.

A lock operation on an object could be accompanied by a mask, which identifies the part(s) of the object involved. The combination of the lock type (i.e., read/write) and the mask determines whether access to the object may be granted. For example, in the Barnes-Hut application, a mask could have logically split a body into two parts, allowing one task to write to the first part of the body while others read from the second part. A programming language compiler could determine whether mask-based locking is required for a particular class of objects. The structure of each mask (i.e., the number of logical parts in which an object is divided) could also be specific to each class.

Re-arrangement of Locking Requests

An approach to dealing with the problem of object thrashing might be the re-arrangement of locking requests. On every node, an attempt could be made to serve pending read or write locking requests on an object as a group. This requires that some lock requests pending in the queue of an object be re-arranged and brought forward. For example, it could be beneficial if any queued read lock requests on an object were served ahead of any pending remote write requests, even if they were submitted later. In this manner, the number of object invalidations may be reduced. The effect that delayed locking requests may have on performance requires investigation.

7.4. Future Research Directions

This research work focused on the design and implementation requirements of an execution environment for object-oriented implicitly parallel computing. Based on the discussion presented in this thesis, new research avenues are identified in the fields of dynamic task management and object-based distributed shared memory.

Implicit Decomposition of Container Objects for Data Parallel Computations

Software developers may find it useful, and sometimes necessary, to express an algorithm in a data parallel manner by using, for example, a 'for each' programming language construct. This thesis demonstrated that such a computation could be represented by an appropriate specialisation of the tasklet construct. The application of a function to the elements of a vector was a representative example. However, it is left up to the programming language support tools (*e.g.*, compiler) to identify and express, in terms of

tasklets, the data parallelism in applications. The design of a tasklet that decomposes container objects at run-time and supports the 'for each' pattern of parallel computation could be investigated.

Heuristics-based Lazy Task Creation

It may be possible to use a heuristics approach in determining whether it is beneficial to pay the extra overhead of lazily creating a new parallel task. For example, when work is requested, the rate by which computation is inlined from a tasklet may be compared to the rate by which new parallel tasks are created from the same tasklet. If the comparison favours inlining, it may be the case that it would be more beneficial to reject the request for a new task.

Heuristics may also give an indication of the most advantageous granularity for new tasks (*e.g.*, the number of iterations that are grouped together from a grouping-capable tasklet). Furthermore, additional run-time or application-provided information may be used when deciding on lazily creating a task, like the location and size of the objects to be used by the new parallel task, the communication overhead between the two nodes involved (i.e., the node requesting work and the node with the available work), the processor(s) speed at a particular node, etc.

Type-assisted DSM

The object-oriented nature of applications may benefit the performance of an object-based DSM system, such as the NIPDSM. Type specific information can be used to improve the effectiveness of the object grouping caching techniques.

Access to the elements of an entire container data structure is often required in applications (*e.g.*, lists, sets, vectors, etc.). Using type information that is provided by the compiler or inferred at run-time through reflection, an object-based DSM system could reduce the overheads of locking operations by allowing entire data structures to be locked with only one operation. The degree of logical parallelism in the application should not be compromised due to a container-wide lock. For example, in a data parallel computation involving the elements of a container object, the entire computation should be regarded as the owner of the lock, hence allowing concurrent access to all the elements. Additionally, concurrent access to the individual elements should be supported when the container object is not locked, as in the current implementation of the NIPDSM.

In the discussion of the NIPDSM caching techniques, it was assumed that a programming language compiler could provide the associations between objects that are required by the object grouping based on associations caching technique. In addition to

the work on compiler techniques that can deduce the necessary object associations, an approach based on type information collected at run-time could also be investigated.

7.5. Concluding Remarks

This thesis described the design and implementation of the NIP lazy task creation technique and the NIP distributed shared memory system for supporting object-oriented implicitly parallel computing. The effectiveness of the tasklet construct for representing iterative and recursive computations was established and the benefits of object grouping techniques for an object-based distributed shared memory system were demonstrated. The performance evaluation of memory intensive parallel applications with low computation to locking ratio revealed the unsuitability of implicit object locking for parallel object-oriented computing.

REFERENCES

Adve, S. V., Cox, A. L., Dwarkadas, S., Rajamony, R. and Zwaenepoel, W. (1996). "A Comparison of Entry Consistency and Lazy Release Consistency Implementation." In *Proceedings of the 2nd IEEE Symposium on High-Performance Computer Architecture (HPCA-2)*: pp.26-37.

Adve, S. V. and Gharachorloo, K. (1996). "Shared Memory Consistency Models: A Tutorial." *IEEE Computer*, 29(12): pp.66-76.

Adve, S. V. and Hill, M. D. (1990). "Weak Ordering: A New Definition." In *Proceedings of the 17th Annual International Symposium on Computer Architecture*: pp.2-14.

Agarwal, A., Bianchini, R., Chaiken, D., Johnson, K., Kranz, D., Kubiatowicz, J., Lim, B. H., Mackenzie, K. and Yeung, D. (1995). "The MIT Alewife Machine: Architecture and Performance." In *Proceedings of the 22nd Annual International Symposium on Computer Architecture*: pp.2-13, Santa Margherita Ligure, Italy.

Agarwal, A., Chandra, A. K. and Snir, M. (1989). "On Communication Latency in PRAM Computations." In *Proceedings of the Symposium on Parallel Algorithms and Architectures*: pp.11-21, ACM.

Agha, G. and Hewitt, C. (1987). "Actors: An Conceptual Foundation for Concurrent Object-Oriented Programming." *Research Directions in Object-Oriented Programming*: pp.49-74.

Ahamad, M., Bazzi, R., John, R., Kohli, P. and Neiger, G. (1992). *The Power of Processor Consistency*. Technical Report, GIT-CC-92/34, Georgia Institute of Technology.

Ahuja, S., Carriero, N. and Gelernter, D. (1986). "Linda and Friends." *Computer*, 19(8): pp.26-34.

Almasi, G. S. and Gottlieb, A. (1994). *Highly Parallel Computing*, Benjamin/Cummings.

Anderson, T., Levy, H., Bershad, B. and Lazowska, E. (1991). "The Interaction of Architecture and Operating System Design." In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)*: pp.108-120.

Appel, A. W. and Li, K. (1991). "Virtual Memory Primitives for User Programs." *ACM SIGPLAN Notices - Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 26(4): pp.96-107.

Bacon, D. F., Graham, S. L. and Sharp, O. J. (1994). "Compiler Transformations for High-Performance Computing." *ACM Computing Surveys*, 26(4): pp.345-420.

Bal, H. E., Kaashoek, M. F. and Tanenbaum, A. S. (1992). "Orca: A Language For Parallel Programming of Distributed Systems." *IEEE Transactions on Software Engineering*, 18(3): pp.190-205.

Barnes, J. and Hut, P. (1986). "A Hierarchical O(N log N) Force Calculation Algorithm." *Nature*(324): pp.446-449.

Bennett, J. K., Carter, J. B. and Zwaenepoel, W. (1990a). "Adaptive Software Cache Management for Distributed Shared Memory Architectures." In *Proceedings of the Seventeenth International Symposium on Computer Architecture*: pp.125-134.

Bennett, J. K., Carter, J. B. and Zwaenepoel, W. (1990b). "Munin: Distributed shared memory based on type--specific memory coherence." In *Proceedings of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*: pp.168-176.

Bershad, B. N. and Zekauskas, M. J. (1991). *Midway: Shared Memory Parallel Programming with Entry Consistency for Distributed Memory Multiprocessors*. Technical Report, CMU-CS-91-170, School of Computer Science, Carnegie Mellon University.

Bershad, B. N., Zekauskas, M. J. and Sawdon, W. A. (1993). "The Midway Distributed Shared Memory System." In *Proceedings of the IEEE COMPCON Conference*: pp.528--537.

Billard, E. A. and Pasquale, J. C. (1997). "Load Balancing to Adjust for Proximity in Some Network Topologies." *Parallel Computing*, 22(14): pp.2007-2023.

Billas, A., Iftode, L. and Singh, J. P. (1998). "Evaluation of Hardware Support for Next Generation Shared Virtual Memory Clusters." In *Proceedings of the International Conference on Supercomputing*: pp.274-281.

Bird, R. and Wadler, P. (1988). *Introduction to Functional Programming*. Englewood Cliffs, N.J., Prentice Hall.

Blumofe, R. D., Joerg, C. F. and Kuszmaul, B. C. (1995). "Cilk: An Efficient Multithreaded Runtime System." *SIGPLAN Notices*, 30(8): pp.207-216.

Buck, B. and Keleher, P. (1998). "Locality and Performance of Page- and Object-based DSMs." In *Proceedings of the 12th International Parallel Processing Symposium*.

Burks, A. W., Goldstine, H. H. and von Neumann, J. (1962). "Preliminary Discussion of the Logical Design of an Electronic Computing Instrument (Part I)." *Datamation*, 8(September (Part I), October (Part II)).

Carter, J. B. (1995). "Design of the Munin Distributed Shared Memory." *Journal of Parallel and Distributed Computing*, 29: pp.219-227.

Carter, J. B., Bennett, J. K. and Swaenepoel, W. (1991). "Implementation and Performance of Munin." In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*: pp.152-164.

Cox, A. L., Dwarkadas, S., Keleher, P. J., Lu, H., Rajamony, R. and Zwaenepoel, W. (1994). "Software Versus Hardware Shared-Memory Implementation: A Case Study." In *Proceedings of the 21st Annual International Symposium of Computer Architecture*: pp.106-117, IEEE Computer Soc. Press.

Culler, D., Dusseau, A., Goldstein, S., Lumetta, S., von Eicken, T. and Yelick, K. (1993a). "Parallel Programming in Split-C." In *Proceedings of the Supercomputing 93*: pp.262-273.

Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K., Santos, E., Subramonian, R. and von Eicken, T. (1993b). "LogP: Towards a Realistic Model of Parallel Computation." In *Proceedings of the 4th Symposium on Principles and Practise of Parallel Programming*: pp.78-85, ACM SIGPLAN.

Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K., Subramonian, R. and von Eicken, T. (1996). "LogP: A Practical Parallel Model of Computation." *Communications of the ACM*, 39(11): pp.78-85.

Culler, D. E., Arpaci-Dusseau, A., Arpaci-Dusseau, R., Chun, B., Lumetta, S., Mainwaring, A., Martin, R., Yoshikawa, C. and Wong, F. (1997). "Parallel Computing on the Berkeley NOW." In *Proceedings of the 9th Joint Symposium on Parallel Processing*, Lobe, Japan.

Culler, D. E. and Singh, J. P. (1999). *Parallel Computer Architecture - A Hardware/Software Approach*, Morgan Kaufmann.

Darlington, J. and Reeve, M. J. (1981). "ALICE: A Multiple-Processor Reduction Machine for the Evaluation of Adaptive Languages." In *Proceedings of the FPCA*: pp.65-75.

Dongarra, J. J. (1994). *Performance of Various Computers Using Standard Linear Equations Software*, CS-89-85, Computer Science Department, University of Tennessee.

Eager, D. L., Lazowska, E. D. and Zahorjan, J. (1986). "Adaptive Load Sharing in Homogeneous Distributed Systems." *IEEE Transactions on Software Engineering*, 12(5): pp.662-675.

Engler, D. R., Lowenthal, D. K. and Andrews, G. R. (1993). *Filaments: Efficient Fine-Grain Parallelism on Shared-Memory Multiprocessors*. Technical Report, TR93-13a, Department of Computing Science, University of Arizona.

Field, A. J. and Harrison, P. G. (1988). *Functional Programming*. Workingham, England, Addison-Wesley.

Flynn, M. J. (1972). "Some Computer Organisations and Their Effectiveness." *IEEE Transactions on Computers*, C21(9): pp.948-960.

Forin, A., Barrera, J., Young, M. and Rashid, R. (1988). "Design, Implementation and Performance Evaluation of a Distributed Shared Memory Server for Mach." In *Proceedings of the 1988 Winter USENIX*.

Fortune, S. and Wyllie, J. (1978). "Parallelism in Random Access Machines." In *Proceedings of the 10th Annual Symposium on Theory of Computing*: pp.114-118, ACM.

Forum, M. P. I. (1994). "MPI: A Message-Passing Interface Standard." *International Journal of Supercomputer Applications and High Performance Computing*, 8(3/1).

Freeh, V. W., Lowenthal, D. K. and Andrews, G. R. (1994). "Distributed Filaments: Efficient Fine-Grain Parallelism on a Cluster of Workstations." In *Proceedings of the 1st Symposium on Operating Systems Design and Implementation*: pp.201-212, Monterey, CA.

Gajski, D. D. and Peir, J.-K. (1985). "Comparison of five multiprocessor systems." *Parallel Computing*, 2(3): pp.264-282.

Gharachorloo, K., Gupta, A. and Hennessy, J. (1993). *Revisions to "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors"*. Technical Report, CSL-TR-93-568, Computer Systems Laboratory, Stanford University.

Gharachorloo, K., Lenoski, D., Laudon, J., Gibbons, P., Gupta, A. and Hennessy, J. (1990). "Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors." In *Proceedings of the 17th International Symposium on Computer Architecture*: pp.15-26, ACM.

Goldstein, S. C. (1997). *Lazy Threads: Compiler and Runtime Structures for Fine-Grained Parallel Programming*. PhD Thesis, Department of Computer Science, University of California-Berkeley.

Goldstein, S. C., Schauser, K. E. and Culler, D. E. (1996). "Lazy Threads: Implementing a Fast Parallel Call." *Journal of Parallel and Distributed Computing*, 37(1): pp.5-20.

Gurd, J. R., Kirkham, C. C. and Watson, I. (1985). "The Manchester Prototype Dataflow Computer." *Communications of the ACM*, 28(1): pp.34-52.

Halstead, R. (1985). "Multilisp: A Language for Concurrent Symbolic Computation." *ACM Transactions on Programming Languages and Systems*, 7(4): pp.501-538.

Henson, M. C. (1987). *Elements of Functional Languages*, Blackwell Scientific Publications.

Hillis, D. W. and Tucker, L. W. (1993). "The CM-5 Connection Machine: A Scalable Supercomputer." *Communicatios of the ACM*, 36(11): pp.31-40.

Hudak, P. (1989). "Conception, Evolution, and Application of Functional Programming Languages." *ACM Computing Surveys*, 21(3): pp.359-411.

Hudak, P. and Fasel, J. (1992). "A Gentle Introduction to Haskell." *SIGPLAN Notices*, 27(5): pp.T1--T53.

Hyde, R. and Fleisch, B. D. (1998). "A Case for Virtual Distributed Objects." *Parallel and Distributed Computing Practices*, 1(3).

IEEE (1996). *9945-1:1996 (ISO/IEC) [IEEE/ANSI Std 1003.1 1996 Edition] Information Technology-Portable Operating System Interface (POSIX)-Part 1: System Application: Program Interface [C Language] (ANSI)*, IEEE Standards Press.

Iftode, L., Blumrich, M., Dubnicki, C., Oppenheimer, D. L., Singh, J. P. and Li, K. (1999). "Shared Virtual Memory with Automatic Update Support." In *Proceedings of the International Conference on Supercomputing*.

Iftode, L., Singh, J. P. and Li, K. (1998). "Scope Consistency: A Bridge between Release Consistency and Entry Consistency." *Theory of Computing Systems*, 31: pp.451-473.

James, D. V. (1994). "The Scalable Coherent Interface: Scaling to High-Performance Systems." In *Proceedings of the COMPCON'94*: pp.64-71, Los Alamitos, California, IEEE CS Press.

Johnson, G. (2000). *Numeric Parallel Programming in NIP*. 3rd Year Project, Dep. of Computing Science, University of Newcastle upon Tyne, UK.

Jones, P. S. L. (1987). *The Implementation of Functional Programming Languages*. Englewood Cliffs, Í. J., Prentice-Hall International.

Jul, E., Henry, L., Hutchinson, N. and Black, A. (1988). "Fine-Grained Mobility in the Emerald System." *ACM Transactions on Computer Systems*, 6(1): pp.109-133.

Keleher, P. (1995). *Lazy Release Consistency for Distributed Shared Memory*. PhD Thesis, Department of Computer Science, Rice University.

Keleher, P., Cox, A. L., Dwarkadas, S. and Zwaenepoel, W. (1994). "TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems." In *Proceedings of the 1994 Winter USENIX Conference*: pp.115-131, San Francisco, CA, USA.

Keleher, P., Cox, A. L., Dwarkadas, S. and Zwaenepoel, W. (1995). "An Evaluation of Software-Based Release Consistent Protocols." *Journal of Parallel and Distributed Computing*, 29: pp.126-141.

Keleher, P., Cox, A. L. and Zwaenepoel, W. (1992). "Lazy Release Consistency for Software Distributed Shared Memory." In *Proceedings of the 19th Annual Internation Symposium on Computer Architecture*: pp.13-21.

Kelly, T. C. (2000). *An Investigation of some Explicit and Implicit Parallel Programming Techniques*. MSc. Dissertation, Dep. of Computing Science, University of Newcastle upon Tyne, UK.

Kranz, D., Halstead, R. and Mohr, E. (1989). "Mul-T, A High-Performance Parallel Lisp." In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*: pp.81-90, Portland, OR.

Lamport, L. (1978). "Time, Clocks and the Ordering of Events in a Distributed System." *Communications of the ACM*, 21(7): pp.558-565.

Lamport, L. (1979). "How to Make a Multiprocessor Computer That Correctly Executes Multiprocessors Programs." *IEEE Transactions on Computers*, C-28(9): pp.690-691.

Lavender, R. G. and Schmidt, D. C. (1996). "Active Object: An Object Behavioral Pattern for Concurrent Programming." *Pattern Languages of Program Design*. Coplien, Vlissides and Kerth. Monticello IL.

Lee, J. and Ramachandran, U. (1990). "Synchronization with Multiprocessor Caches." In *Proceedings of the 17th Annual Symposium on Computer Architecture*: pp.27-37.

Levelt, W. G., Kaashoek, M. F., Bal, H. E. and Tanenbaum, A. S. (1992). "A Comparison of two Paradigms for Distributed Shared Memory." *Software-Practise and Experience*, 22: pp.985-1010.

Lewis, T. G. and El-Rewini, H. (1992). *Introduction to Parallel Computing*, Prentice-Hall.

Li, K. (1986). *Shared Virtual Memory on Loosely Coupled Multiprocessors.* PhD Thesis, Dep. of Computer Science, Yale University.

Li, K. and Hudak, P. (1989). "Memory Coherence in Shared Virtual Memory Systems." *ACM Transactions on Computer Systems*, 7(4): pp.321-358.

Loh, P. K. K., Hsu, W. J., Wentong, C. and Sriskanthan, N. (1996). "How Network Topology Affects Dynamic Load Balancing." *IEEE Parallel & Distributed Technology*, 4(3): pp.25--35.

Lu, H., Dwarkadas, S., Cox, A. L. and Zwaenepoel, W. (1997). "Quantifying the Performance Differences between PVM and TreadMarks." *Journal of Parallel and Distributed Computation*, 43(2): pp.65-78.

Lüling, R., Monien, B. and Ramme, F. (1992). *A Study on Dynamic Load Balancing Algorithms.* Technical Report PC^2, TR-001-92, PC^2- Paderborn Center for Parallel Computing, Universität-GH Paderborn.

Microsoft (2000). Microsoft .NET: Realizing the Next Generation Internet (white paper).

Mohr, E., Kranz, D. A. and Halstead, R. H. J. (1991). "Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs." *IEEE Transactions on Parallel and Distributed Systems*, 2(3): pp.264-280.

Moore, G. E. (1965). "Cramming More Components Onto Integrated Circuits." *Electronics Magazine*(38): pp.114-117.

Moore, G. E. (1997). "An Update on Moore's Law." In *Proceedings of the Intel Developer Forum*, San Francisco.

Moore, G. E. (1998). What is Moore's Law, Intel.

Mosberger, D. (1993). "Memory Consistency Models." *Operating Systems Review*, 27(1): pp.18-26.

Newman, A. (1996). *Special Edition Using Java*, Que.

Oracle (1999). Oracle Application Server - Statement of Direction (white paper).

Ousterhout, J. K. (1994). *TCL and the TK toolkit*, Addison Wesley Longman Publishing Co.

Parastatidis, S. (2000). *The NIP Runtime System Developer's Guide (work in progress)*, , Dep. of Computing Science, University of Newcastle upon Tyne.

Parastatidis, S. and Watson, P. (1999a). "An Object-based Software DSM for the NIP Parallel System." In *Proceedings of the 1st Workshop on Software Distributed Shared Memory in conjunction with the 1999 ACM International Conference on Supercomputing ICS'99*, Rhodes, Greece.

Parastatidis, S. and Watson, P. (1999b). *An Object-based Software DSM for the NIP Parallel System*. Technical Report, CS-TR-678, Dep. of Computing Science, University of Newcastle upon Tyne.

Perrott, R. H. and Zarea-Aliabadi, A. (1986). "Supercomputer Languages." *ACM Computing Surveys*, 18(1): pp.5-22.

Pfister, G. F. (1998). *In Search of Clusters*, Prentice-Hall.

Protic, J., Tomasevic, M. and Milutinovic, V., Eds. (1998). *Distributed Shared Memory - Concepts and Systems*, IEEE Computer Society.

Rovner, P. (1986). "Extending Modula-2 to Build Large, Integrated Systems." *IEEE Software*, 3(6): pp.46-57.

Sargeant, J. (1993). *United Functions and Objects: An Overview*. Technical Report, UMCS-93-1-4, Dept. of Computer Science, University of Manchester.

Sargeant, J. and Kirkham, C. (1994). *The Uflow Computational Model and Intermediate Format*. Technical Report, UMCS 94-5-1, Dept. of Computer Science, University of Manchester.

Scales, D. J., Gharachorloo, K. and Thekkath, C. A. (1996). "Shasta: A Low Overhead, Software-Only Approach for Supporting Fine-Grain Shared Memory." In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*: pp.174-185, Cambridge, MA.

Schmidt, D. C. (1995). *An OO Encapsulation of Lightweight OS Concurrency Mechanisms in the ACE Toolkit*. Technical Report, WUCS-95-31, Washington University.

Skillicorn, D. B. and Talia, D. (1998). "Models and Languages for Parallel Computation." *ACM Computing Surveys*, 30(2): pp.123-169.

Sterling, T., Messina, P. and Smith, P. H. (1995). *Enabling Technologies for Petaflops Computing*, Massachusetts Institute of Technology.

Stroustrup, B. (1997). *The C++ Programming Language*, Addison-Wesley.

Stumm, M. and Zhou, S. (1990). "Algorithms Implementing Distributed Shared Memory." *Computer*, 23(5): pp.54-64.

Sunderam, V. S. (1990). "PVM: A Framework for Parallel Distributed Computing." *Concurrency: Practice and Experience*, 2(4): pp.315-339.

Tanenbaum, A. (1995). *Distributed Operating Systems*, Prentice Hall International Editions.

Tanenbaum, A. S. (1999). *Structured Computer Organisation*, Prentice-Hall.

Tanenbaum, A. S., van Renesse, R., van Staveren, H., Sharp, G. J., Mullender, S. J., Jansen, A. J. and van Rossum, G. (1990). "Experiences with the Amoeba Distributed Operating System." *Communications of the ACM*, 33(2): pp.46-63.

TOP500 List (2000). The TOP500 List on-line, http://www.top500.org.

TOP500 List Authors (2000). *The TOP500 List Database.* Personal Communication.

Treleaven, P. C. (1985). "Control-Driven, and Demand-Driven Computer Architecture (Abstract)." *Parallel Computing*, 2.

Trinder, P. W., Hammond, K., Mattson Jr, J. S. and Partridge, A. S. (1996). "GUM: A Portable Parallel Implementation of Haskell." In *Proceedings of the Programming Language Design and Implementation*: pp.79--88, Philadelphia, USA.

Turner, D. A. (1985). "Miranda: A Non-strict Functional Language with Polymorphic Types." *Functional Programming Languages and Computer Architecture - Lecture Notes in Computer Science*, Springer-Verlag. 201: pp.1-16.

Valiant, L. G. (1990). "A Bridging Model for Parallel Computation." *Communications of the ACM*, 33(8): pp.103--111.

Vandervoorde, M. T. and Roberts, E. S. (1988). "WorkCrews: An Abstraction for Controlling Parallelism." *International Journal of Parallel Programming*, 17(4): pp.347-66.

Veen, A. H. (1986). "Dataflow Machine Architectures." *ACM Computing Surveys*, 18(4): pp.366-396.

von Eicken, T., Basu, A., Buch, V. and Vogels, W. (1995). "U-Net: a User-Level Network Interface for Parallel and Distributed Computing." In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*: pp.40-53, Copper Mountain, Colorando.

von Eicken, T., Culler, D. E., Goldstein, S. C. and Schauser, K. E. (1992). "Active Messages: a Mechanism for Integrated Communication and Computation." In *Proceedings of the 19th Annual Internation Symposium on Computer Architecture*: pp.256-266.

Wagner, D. B. and Calder, B. G. (1993). "Leapfrogging: A Portable Technique for Implementing Efficient Futures." In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*: pp.208-217.

Warren, M. S., Becker, D. J., Goda, M. P., Salmon, J. K. and Sterling, T. (1997). "Parallel Supercomputing with Commodity Components." In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97)*: pp.1372-1381.

Warren, M. S., Germann, T. C., Lomdahl, P. S., Beazley, D. M. and Salmon, J. K. (1998). "Avalon: An Alpha/Linux Cluster Achieves 10 Gflops for $150k." In *Proceedings of the Supercomputing '98*, Los Alamitos, IEEE Comp. Soc.

Watson, I., Woods, V., Watson, P., Banach, R., Greenberg, M. and Sargeant, J. (1988). "Flagship: A Parallel Architecture for Declarative Programming." In *Proceedings of the 15th Annual Symposium on Computer Architecture*: pp.124-130.

Watson, P. (1996). *The NIP Runtime System.* Personal Communication.

Watson, P. and Parastatidis, S. (1999a). "The NIP Parallel Object-Oriented Computational Model." *Network-based Parallel Computing: Communication, Architecture, and Applications - Third International Workshop, CANPC'99*. Lecture Notes in Computer Science, 1602. Sivasubramaniam, A. and Lauria, M. Orlando, Florida, USA, Springer-Verlag: pp.122-136.

Watson, P. and Parastatidis, S. (1999b). *NIP: A Parallel Object-Oriented Computational Model.* Technical Report, CS-TR-658, Dep. of Computing Science, University of Newcastle upon Tyne.

Watson, P. and Parastatidis, S. (1999c). "An Optimised Lazy Task Creation Technique for Iterative and Recursive Computations." In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*: pp.1971-1977, Las Vegas, Nevada, USA, CSREA Press.

Webber, J. (1998). "Vorlon: A Visual Object-Oriented Approach to Parallel Application Development." In *Proceedings of the Automated Software Engineering*, Honolulu, IEEE Press.

Webber, J. (1999). *The Vorlon Visual, Object-Oriented Programming Language.* Personal Communication.

Webber, J. (2000). *Visual, Object-Oriented Development of Parallel Applications.* PhD Thesis (to be submitted), Dep. of Computing Science, University of Newcastle upon Tyne.

Wirth, N. (1971). "The Programming Language Pascal." *Acta Inf.*, 1: pp.35-63.

Zhou, Y., Iftode, L., Singh, J. P., Li, K., Toonen, B. R., Schoinas, I., Hill, M. D. and Wood, D. A. (1997). "Relaxed Consistency and Coherence Granularity in DSM Systems: A Performance Evaluation." In *Proceedings of the 6th ACM Symposium on Principles and Practice of Parallel Programming*: pp.193--205.

Zucker, R. N. and Baer, J.-L. (1992). "A Performance Study of Memory Consistency Models." In *Proceedings of the 19th Annual International Symposium on Computer Architecture*: pp.2-12.