Affinity Scheduling of Unbalanced Workloads

A Thesis Submitted to the College of
Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the Degree of
Master of Science
in the
Department of Computational Science
University of Saskatchewan
Saskatoon, Saskatchewan

by

Srikant Subramaniam

August, 1993

PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computational Science
University of Saskatchewan
Saskatoon, Saskatchewan, Canada
S7N 0W0

Affinity Scheduling of Unbalanced Workloads

Candidate: Srikant Subramaniam

Supervisor: Derek L. Eager

Master of Science Thesis

Presented to the College of Graduate Studies and Research

University of Saskatchewan

August, 1993

ABSTRACT

Shared memory multiprocessor systems are becoming increasingly important and common. Multiprocessor environments are significantly different from uniprocessor environments, raising new scheduling issues that need to be considered. A fundamental scheduling issue arises in situations in which a unit of work may be processed more efficiently on one processor than on any other, due to factors such as the rate at which the required data can be accessed from the given processor. The unit of work is said to have an "affinity" for the given processor, in such a case. The scheduling issue that has to be considered is the tradeoff between the goals of respecting processor affinities (so as to obtain improved efficiencies in execution) and of dynamically assigning each unit of work to whichever processor happens to be, at the time, least loaded (so as to obtain better load balance and decreased processor idle times).

A specific context in which the above scheduling issue arises is that of shared memory multiprocessors with large, per-processor caches or cached main memories. The shared-memory programming paradigm of such machines permits the dynamic scheduling of work. The data required by a unit of work may, however, often reside mostly in the cache of one particular processor, to which that unit of work thus has affinity.

In this thesis, the design of "affinity scheduling" algorithms, in which both affinity and load balancing considerations play major roles in the scheduling policy, is explored. Two new affinity scheduling algorithms are proposed for a context in which the units of work have widely varying execution times. An experimental study of these algorithms finds them to be superior to the previously proposed algorithms in this context.

ACKNOWLEDGEMENTS

I would like to thank my supervisor Derek Eager for his guidance and encouragement throughout my research work. Derek was always available to discuss the ideas and the results presented in this thesis and often steered me out of the dark holes that research was getting me into. Working with him was a great learning experience and I greatly appreciate the time he took to read my thesis. I believe that this thesis is better for it. I would also like to thank the other members of my committee, Carey Williamson, Mark Keil, and Joseph Irudayaraj, for their helpful suggestions.

Financial assistance in the form of a Graduate Scholarship from the College of Graduate Studies and Research is appreciated.

I want to thank Anant "Jacobi" Jalnapurkar for his help in deciphering the Jacobi puzzle. I doubt, other than the two of us, if anybody else would have pursued the Jacobi algorithm with greater passion and fervour. Thanks to the members of the DISCUS group for listening to my ideas providing suggestions. Thanks especially go to Sundaram for helping me out during the initial stages of my research as I took my first faltering steps into the world of parallel programming. I would like to thank fellow graduate students who have made my stay at Saskatoon an enjoyable one. I would also like to express my gratitude to Gail and Maureen for their cooperation.

Finally, I have received a lot of support, friendship and encouragement from a lot of people in matters unrelated to my research. I am extremely grateful to all of them for their kindness. It is not possible for me to mention all the acknowledgements in this space, so I end with the following dedication.

To my parents

Contents

List of Figures

Chapter 1

Introduction

The term *parallel processing* refers to the execution of multiple program instructions concurrently, rather than one after the other as in *sequential* or *serial* processing, as happens in conventional systems. Parallel processing is not a new phenomenon. Hardware designers have dealt with the problems and rewards of parallelism at least since the days of von Neumann. In fact, early designs for what we know today as the von Neumann machine included considerations for a variety of parallel features. These parallel designs were rejected mainly because of poor reliability of the components available for building the machines. The designers' lack of experience in building any kind of computing engine also argued for adoption of the simplest possible design.

What *is* new is the emerging popularity of parallel processing. Manufacturers, unable to provide sufficient speed solely through faster hardware components, now propose multiprocessors, in which programmers control and exploit multiple CPUs directly to cooperate in solving a problem, as the new direction that high-performance computers must take [50].

It is traditional to divide parallel processing architectures into two broad classes: SIMD and MIMD [23]. SIMD machines are especially designed to perform parallel computations on vector or matrix types of data. In the most common form of SIMD machines, multiple arithmetic logic units, called processing elements (PEs), operate in parallel in a lockstep fashion. These PEs are under the supervision of a single control unit and are synchronized to perform the same function at the same time. SIMD machines are often the most cost-effective way of achieving high performance in some particular application domains, being very well suited for speeding up the matrix calculations used in finite-element analysis, seismic modeling and image processing, for example. It is quite clear, however, that SIMD machines are not "general purpose", since they require very regular program and data structures.

MIMD architectures employ multiple processors that can execute independent instruction streams. Software processes executing on MIMD architectures are synchronized by message passing through an interconnection network or through shared data in shared memory units, under software control, in contrast to the lockstep hardware imposed synchronization in SIMD systems. The impetus for developing MIMD architectures can be ascribed to several interrelated factors. Most importantly, MIMD computers support higher level parallelism (subprogram and task levels) rather than just the "data parallelism" exploited by SIMD machines, and thus are truly "general purpose" systems.

The performance of a parallel program on a MIMD architecture is critically dependent upon the amounts of overhead that arise from the following four sources: *synchronization, process management, communication,* and *load imbalance.*

Synchronization occurs when a processor must wait for some action by another processor, such as relinquishing a critical region, or completing some unit of computation. Such waiting may increase program execution time, not only owing to the wasted time itself, but also owing to non-productive usage of perhaps scarce, shared resources (such as bus cycles), while waiting (for example, while checking whether or not the required action has yet occurred). While synchronization can be a dominant source of overhead, at least that portion of the overhead owing to non-productive usage of shared resources can often be eliminated to a large extent through the use of appropriate scalable synchronization primitives and algorithms [6].

Process management overhead refers to the time required to create, destroy and schedule multiple "processes" (units of sequential execution). In the message passing model, an address space must be allocated for each process (such processes are termed *heavyweight processes*), and the operating system must be involved in process management. Processes are therefore expensive (as operating systems are slow) and are used sparingly (typically one process per processor).

In the shared-memory model, on the other hand, a separate address space is not needed for each unit of sequential execution. In the lowest cost implementations, units of sequential execution that share address spaces (often termed "lightweight processes" or *threads*) are implemented outside the operating system, in a user-level thread library (*user-level threads)* [9] [20] [21] [33], although they may also be implemented in the operating system (*kernel threads*). Ideally, the availability of cheap threads would allow the programmer to structure a program in a very natural way, representing each logically separate unit of sequential computation by a separate thread. In practice, however, with even the most efficient thread implementation, the programmer may be forced to consider carefully the thread "granularity" (amount of work each thread represents) in light of the overhead of thread management in the system in use. Thus the "one process per processor" program structure is common in the shared memory model as well (particularly in a form in which the processes are "workers" retrieving units of work from a queue maintained in shared memory) [21], ensuring negligible process management overhead.

Communication overhead is introduced by interaction between processes. Communication manifests itself as cache misses in multiprocessors with caches, as non-local memory accesses (and also possibly cache misses) in machines that support a shared memory programming model, yet in which the shared memory is physically distributed, and as the sending or receiving of messages in a distributed-memory machine. Note that large-scale shared memory machines invariably have physically distributed memory modules, and thus typically fall into the second of these categories.

Load imbalance occurs when some processors are idle and yet there is work ready to be performed that no processor has yet started. There are two distinct sources of load imbalance in a parallel program: (1) an uneven assignment of computation to units of work, and (2) an uneven assignment of units of work to processors. A "fine grain" decomposition (one in which the units of work are very small) minimizes the effects of variance owing to the uneven assignment of computation to units of work. To avoid an uneven assignment of units of work to processors, most shared-memory programming systems use a central work queue from which either idle processors remove light-weight threads (the central queue is a "ready queue" in this case), or idle worker processes remove units of work [59] [61]. A central queue facilitates a dynamic, even distribution of load among processors, and ensures that no processor remains idle while there is work to be done. This dynamic scheduling scheme is effective because, in a shared-memory program on a shared-memory multiprocessor, each unit of work can execute on any processor and still access the shared data it requires.

A disadvantage of dynamic load scheduling through the use of a central queue, however, is that it may introduce a significant amount of communication overhead. A central work queue as may be used in the shared-memory model can significantly increase communication costs, as each unit of work is executed on the next available (idle) processor, even though the data needed by the unit of work may reside in the cache or local memory of some other processor. A sequence of units of work may require the same data, but if not processed on the same processor, will cause the data to move back and forth between processor caches (if the data is cached), or will result in many remote references (if the data is not cached). The resulting communication overhead can be substantial, and is incurred whether or not load imbalance (the motivation for dynamic scheduling in the first place) is a substantive concern.

In addition to overheads arising from synchronization, process management, communication, and load imbalance within a single application, overheads may also arise from the scheduling actions required in order to effectively run multiple parallel applications concurrently. Although such "multiprogramming" may be desirable to improve system efficiency and thus ultimately improve the performance offered to the end-users, it raises a number of problems whose solutions require new scheduling mechanisms and policies. In particular, in this context as well, there is a tradeoff between the minimization of load imbalance, through dynamic scheduling, and the minimization of communication overhead, which must be addressed in the design of the scheduler.

The scheduling of shared memory MIMD multiprocessors is the domain of this research. Of particular interest is the scheduling of units of work within a single application, when these units of work have "affinities" for particular processors (owing to the locations of the data they access). A particular approach to scheduling termed *affinity scheduling* [43], in which both affinity and load balancing considerations play major roles, is considered. Affinity scheduling is studied in the context of a "one process per processor" program structure in which the units of work being scheduled are the iterations of a program loop that has been parallelized, and in which this loop is repetitively executed. This is a common shared-memory program structure in practice.

The primary goal of this research is to develop and experimentally study improved affinity scheduling algorithms for the context of "unbalanced" workloads in which the parallel loop being executed is severely unbalanced with respect to the amounts of computation represented by each iteration, a situation in which simple forms of affinity scheduling may become inappropriate. Two new algorithms *dynamic* and *wrapped* are proposed. An experimental study finds that these new algorithms achieve both the objectives of maintaining affinity, and load balance, even for severely unbalanced workloads. Both algorithms achieve a significant improvement in performance when compared to the principal existing algorithm.

Chapter 2 discusses shared memory multiprocessor systems, focusing on the key issues pertaining to scheduling in such systems. Chapter 3 discusses affinity scheduling in particular, and describes the existing as well as the proposed affinity scheduling algorithms. Chapter 4 outlines the experimental study of affinity scheduling algorithms that was performed in this thesis research, and presents and analyzes the results obtained. Finally, the major thesis conclusions and contributions, and possible directions for future research, are summarized in Chapter 5.

Chapter 2

Shared Memory Multiprocessors

Multiprocessing is a popular way to increase system computing power beyond the limits of current uniprocessor technology. In a multiprocessor, multiple instruction streams execute in parallel, exchange data and synchronize by passing messages or sharing memory. The shared memory programming model is widely believed to be easier to use than the message passing model. The conceptual simplicity of the shared memory model arises from similarities with sequential programming. Evidence in favor of the shared memory model is the current considerable effort in software development designed to provide the illusion of shared memory on multiprocessors with physically distributed memory modules.

This chapter covers some of the important aspects of multiprocessor systems that support a shared-memory programming model (memory may or may not be, in fact, physically distributed). The chapter begins with an architectural overview where various architectural approaches are overviewed. This is followed by a section on scheduling in shared memory systems where the concepts involved in job and thread scheduling, and also loop scheduling, are discussed.

2.1 Architectural Overview

2.1.1 UMA and NUMA Machines

Shared memory multiprocessor architectures are usually classified on the basis of the type of physical memory subsystem provided. The memory modules of a multiprocessor can be distributed among processors: these architectures are commonly referred to as *non-uniform memory access* (NUMA) architectures because the access time to the memory module that is local to a processor is much lower than the access time to remote modules. NUMA machines are generally difficult to program (in some cases, as difficult as if a message-passing programming model was used) because their performance is sensitive to the allocation of shared data structures to physical memory modules. In the *uniform memory access* (UMA) machines, all global memory is accessed through a common interconnection (such as a single bus), so the access time to any shared memory location is uniform across processors.

The design of an efficient UMA or NUMA multiprocessor system is constrained by two major problems. The first, and possibly the most important, is access latency – that is, the delay between the issuance of a memory access by a processor and its completion. The second problem is contention among accesses from different processors. When there are too many fast CPUs all vying for access to the same memory, contention on the network interconnecting processors and memory modules can greatly degrade system performance.

Caches have been employed as a common approach to solving both these problems. The locality of memory references over time and space enables the cache to service a majority of all the memory requests, allowing main memory to handle only a small fraction of the requests. In this manner, the average access latency is reduced, as is contention on the network connecting processors and memory modules. The use of caches on NUMA machines may also allow a simpler programming style than on NUMA architectures without caches, as even remotely stored data can be accessed quickly once cached locally.

The simplicity of sharing code and data structures among the processes comprising the parallel application is a major advantage in shared memory multiprocessors. If shared data is cacheable, however, this sharing can result in several copies of a shared block of data in one or more caches at the same time. To maintain a coherent view of the memory, these copies must be consistent. This is the *cache coherence* problem. There exist both hardware [8] and software [13] solutions to the cache coherence problem.

The two main categories of hardware based solutions to the cache coherence problem are the snoopy cache protocols and the directory protocols. Snoopy cache protocols [25] are mainly suited for bus based multiprocessors. A snoopy cache controller listens to transactions between main memory and the other caches and updates its state based on what it hears. On learning that some cache has modified the value of a data block, the other caches could either invalidate or update their copy (write-invalidate or write-update). Every cache must process every transaction to find out whether it refers to data stored in the cache. Because all caches in the system must observe memory transactions, a shared bus is typically the medium of communication, as general interconnection networks do not support efficient broadcasting.

The directory based schemes [2] are mainly suited for large scale multiprocessors with more complex interconnection networks. It is desirable in such environments to send memory consistency commands to only those caches that have a stale copy of the data, rather than to broadcast all transactions as in the snoopy cache coherence protocol approach. Such selective communication requires storing exact information about which caches have copies of all cached blocks. Hence, some sort of bookkeeping has to be done by means of a *directory* that tracks all copies of blocks. This directory can be either centralized or distributed. A directory entry for each block of data contains a number of pointers to specify the current location of each copy, as well as state bits. Memory operations query the directory to determine whether cache coherence actions are necessary, and consistency commands are sent to only those caches that have a copy of the block.

2.1.2 CC-NUMA and COMA Machines

Two interesting variations of large scale shared memory systems that have recently emerged are *cache coherent non-uniform memory access* machines (CC-NUMA) and *cache only memory architectures* (COMA). They both have physically distributed main memory and use directory based cache coherence – a hardware technique for maintaining the consistency of cached shared data. A major difference is that COMA machines automatically migrate and replicate main memory blocks among the system's distributed memory modules (using directory based cache coherence), rather than tying down particular physical memory address ranges to particular memory modules as in conventional architectures.

Large scale multiprocessors with a single address space and coherent caches offer a flexible and powerful computing environment. The single address space and coherent caches together ease the problem of data partitioning and dynamic load balancing. They also provide better support for parallelizing compilers, standard operating systems and multiprogramming, thus enabling more flexible and effective use of the machine. Currently, many research groups are pursuing the design of such multiprocessors [3] [27] [36] [51]. Examples of the CC-NUMA machines are the MIT Alewife machine [3] and the Stanford DASH multiprocessor [36], while examples of COMA machines are the Swedish Institute of Computer Science's Data Diffusion Machine (DDM) [27] and the Kendall Square Research KSR1 machine [51].

A CC-NUMA machine consists of a number of processing nodes connected through a high bandwidth low latency interconnection network. Each processing node consists of one or more high performance processors, associated caches, and a portion of the global shared memory. Cache coherence is maintained by a directory based, write-invalidate cache coherence protocol. The Stanford DASH multiprocessor is an example of a CC-NUMA machine. The prototype [37] consists of 16 processing nodes, each with 4 processors, for a total of 64 processors. Each processor has a first level cache of 64 Kbytes and a 256 Kbytes second level cache. The CC-NUMA software model allows processes to be attached to specific processors and for data to be allocated from any specific node's main memory. Most systems do not support the migration or replication of data among the main memories. Those that do manage such data movements at the page level, using the virtual memory system facilities [16]. This is in contrast to the COMA machines where such migration and replication happens at a much finer granularity, and is supported in hardware.

COMA machines are characterized by the lack of any permanent binding of physical memory address ranges to particular memory modules. Instead, processors host a large set-associative memory. The task of such a memory is twofold. It acts as a large (second or third level) cache for the processor, since data accessed by a processor is migrated or replicated to its local memory as necessary, but in addition, serves as a portion of the total physical memory storage space and may store program code or data that the local processor may never even use. This intermediate form of memory is referred to as *Attraction Memory* (AM) in DDM [27].

The cache coherence protocol for a COMA machine can adopt techniques used in other cache coherence protocols and extend them with the functionality for finding a datum on a cache read miss and for handling replacement. A directory based protocol can be employed with the directory information statically distributed in a NUMA fashion, and the data itself allowed to move freely. Retrieving the data on a read miss would then require one extra indirect access to the directory home to find where the item currently resides. The access time, including this extra indirection, would be similar to that required on a CC-NUMA for a read miss on a non-locally stored address. The directory home can also make sure that the last copy of an item is not lost. (Recall that there is no other backing memory as in the case of a conventional cache system.)

The directory based coherence protocol in the DDM relies on a hierarchical snooping bus architecture and uses a hierarchical search algorithm for finding an item. The directory information in DDM is dynamically distributed in the hierarchy. The coherence protocol attracts the data used by a processor to its AM. The coherence unit, comparable to a cache-line, which is moved around by the protocol is called an *item*. On a memory reference, a virtual address is translated into an item identifier.

The item identifier space is logically the same as the physical address space of typical machines, but there is no permanent mapping between an item identifier and a particular location in a memory module. Instead, an item identifier corresponds to a location in an attraction memory, whose tag matches the item identifier. A coherent shared memory view of the system is provided, as all processors share the same item identifier space. Shared data, which may be replicated in many AMs, is kept consistent by way of the directory based protocol.

A COMA is reminiscent of a NUMA architecture, in that the shared memory is physically divided among the processors. In a NUMA, however, different portions of the physical address space (and thus, typically, program's data segments) are statically allocated to each memory, while in a COMA, a datum has no home and might be moved by the coherence protocol to reside in any or many AMs according to its usage. Figure 2.1 compares a COMA to other shared memory architectures.

COMA architectures are currently of great research interest, and in addition are architectures in which forms of affinity scheduling are expected to be quite useful. For this reason, in the following two subsections a detailed look is taken at two particular COMA machines, the KSR1 and DDM.

Figure 2.1: Comparing COMA to more conventional architectures

2.1.3 The Kendall Square Research Machine (KSR1)

The Kendall Square KSR1 machine [51] is a family of high performance general purpose COMA multiprocessors that combine parallel processing, scalability and a conventional shared memory programming environment. The KSR1 is offered in multiple configurations, from 8 to 1088 CMOS based superscalar 64 bit processors, and correspondingly a performance range from 160 MIPS and 320 MFLOPS to 21,760 MIPS and 43,520 MFLOPS.

The architecture is a hierarchy of slotted rings with (in the current product) up to 32 processors in each ring at level 0 and up to 34 rings at level 1, yielding a maximum of 1088 processors. The basic design allows higher-level rings, however, permitting tens of thousands of processors. The rings together with their associated memory system components are termed "ALLCACHE Engines (AEs)", and are given level numbers: for example, the "ALLCACHE Engine 0 (AE-0)" includes the memory subsystem distributed among all nodes on a local slotted ring, while the "ALLCACHE Engine 1 (AE-1)" includes the memory subsystem associated with the slotted ring connecting a set of AE-0's into a single system. The KSR1 provides an efficient environment for a wide variety of workloads and programming environments through a large, hardware-supported coherent shared address space with a large number of processors, and dynamic management of memory through hardware migration and replication of data throughout the distributed memory nodes (each of maximum size 32 MBytes), using its ALLCACHE mechanism.

With ALLCACHE, an address becomes a name and this name automatically migrates throughout the system and is associated with a processor in a cache-like fashion as needed. Data movement occurs in sub-pages of 128 bytes (16 words) of its 16K pages. Copies of a given sub-page are made by the hardware and sent to other nodes to reduce access time. A processor can pre-fetch data into its local cache memory and post-store data for other processors. The hardware is designed to exploit spatial and temporal locality. When a processor writes to an address, all copies of the data are updated and memory coherence is maintained.

A memory read will, if not satisfied either within the local processor cache or memory, generate a request which is placed into an open slot on the communications ring; that slot is matched against the cached elements associated with every processor on the local AE-0, and the first such processor able to fulfill the request does so by placing the required address and data into the next open slot.

In the case of a multi-AE-0 system, one of the AE-0 locations will contain a special purpose node, an ALLCACHE Routing Directory Cell (ARDC), containing knowledge of every address resident within every processor memory on its local AE-0. As the request passes the ARDC, the ARDC is able to determine if the request can be fulfilled within the local AE-0; if so, the request then passes on to the next node on AE-0, but if the request cannot be satisfied within the local AE-0, it passes on to a matching ARDC on the AE-1, and then around the AE-1 to all other ARDC's associated with all other AE-0's in the entire system. At least one of the AE-1 ARDC's will have a reference to the required address and the request is routed down to that AE-0 and thence to the node containing that address. The address and data are then placed back on the AE-0, from where it travels around to the local ARDC, up to the AE-1, around to the requesting ARDC to the requesting AE-0 and finally to the requesting processor.

2.1.4 The Data Diffusion Machine (DDM)

The Data Diffusion Machine [27] is a COMA architecture with a hierarchical network topology and set-associative AMs at its tips storing data value, address tag, and state for all data (see Figure 2.2). Between each level in the hierarchy are set-associative state memories and directories, storing state information for all data in their subsystems, but not their values.

The attraction memories of each cluster within the DDM are connected by a single bus. The distribution and coherence of data among the attraction memories is controlled by the snooping protocol *memory above*, and the interface between the processor and the attraction memory is defined by the protocol *memory below*. These

Figure 2.2: The Data Diffusion Machine with its hierarchical topology

protocols manage data in units termed "items"; each AM stores one small state field per item.

The DDM uses a hierarchical write-invalidate cache coherence protocol that uses the directories to make the coherence traffic as local as possible. The state of a datum in the directory indicates if the datum resides in the subsystem of the directory, and if so, whether other copies might exist outside the subsystem. A directory can therefore judge if a coherence transaction needs to be propagated upwards or downwards, and serves as a filter keeping the traffic as local as possible. Ensuring such locality not only limits the search time, but also minimizes the traffic in the network.

The DDM protocol has a hierarchical search algorithm that uses the state information in the directories to find a datum. A read request moves up the hierarchy as far as necessary to find a directory marked as having a copy of the datum in its subsystem. On its way down in the subsystem, the request is guided by the directories to one copy of the datum (a selection mechanism ensures that only one AM receives the request). The search takes at most $2L - 1$ bus transactions for L levels in the hierarchy. The read request marks its search path with transient states, used by the reply to find its way to the requesting node. The reply changes the transient states to appropriate stable states. The reply is returned in at most $2L - 1$ bus transactions.

Write-invalidate is implemented in a general network by the writing node sending out an erase request and waiting for acknowledgments to be received from each individual AM with a copy of the datum. In a hierarchical network, the topmost node of the subsystem in which all the copies of the item reside may send the acknowledgment. The acknowledgment might be received by the writing AM even before all other AMs have received the erase request.

Although most memory accesses tend to be localized in the machine, it is quite possible that the higher levels in the hierarchy may demand a higher bandwidth than the lower levels, which may cause a network bottleneck. However, the top part of the hierarchy can be "widened" (its network capacity increased) to become a "fat tree" or be replaced by a general network, to form a heterogeneous network.

In the DDM prototype implementation, the hardware implementation of the processor/attraction memory is based on the system TP881V by Tadpole Technology, U.K. Each such node has up to four Motorola 88100 20 MHz processors (referred to as "88k" in Figure 2.3), each with separate 16 Kbyte instruction and data caches, 8 or 32 Mbytes DRAM, and interfaces for SCSI bus, Ethernet, and terminals.

Figure 2.3: The implementation of a DDM node consisting of up to four processors sharing one attraction memory [27]

A DDM Node Controller (DNC) board is used to interface the TP881 node and the first level DDM bus as shown in Figure 2.3. The DNC handles memory accesses between the processor and the main memory of the node, the behavior of which is changed into that of a set-associative memory. The copy-back protocol of the processor caches has been adapted to the DDM protocol. The processor caches have a cache-line size of 16 bytes (which is also the coherence unit of the attraction memories). The DDM bus is an asynchronous split-transaction bus, pipelined in four phases: transaction code, snoop, selection and data. It operates at 20MHz, with a 32 bit data bus and a 32 bit address bus. and provides a bandwidth of about 80 Mbytes per second.

2.2 Scheduling in Shared Memory Systems

Shared memory multiprocessors are frequently used as compute servers with multiple parallel applications potentially executing at the same time. In such multiprogrammed environments, there are three different levels of scheduling. At the lowest level is the scheduling of the units of work within a single parallel construct (such as the iterations of a loop that has been parallelized). The next level of scheduling concerns the scheduling of an application's threads of control among the processors assigned to that application. The highest level of scheduling is concerned with the allocation of processors among competing applications or "jobs".

The most basic goal of a multiprocessor scheduling policy is the assignment of system resources to user "tasks"(units of work, threads or jobs) in a manner that ensures efficient operation of the entire system. In achieving this goal, there are numerous common scheduling properties that are desirable, including fair and predictable service, maximal system throughput and processor utilization, minimizing the total execution time required to execute the tasks comprising a single job, minimizing the mean job response time, minimizing processor idle time, reducing the communication and scheduling overheads, and avoidance of indefinite postponement [18] [48].

Previous research efforts in scheduling have emphasized varying objectives. Some of the scheduling policy issues that have been actively researched in the literature include the order in which tasks are assigned to processors. This issue of task ordering can result in two types of policies, based on the desired objective. Policies of the first type attempt to minimize the total execution time required to execute the tasks of a single job. A considerable amount of research has been devoted to this scheduling problem [1] [14] [15] [29] [32]. Policies of the second type attempt to minimize the mean task or job response time. This problem is specifically addressed in [15] [29] [38] [40] [52].

The survey in this section does not consider the issues of task ordering explicitly, focusing instead on those scheduling issues arising from load balance, overhead, and affinity considerations. Section 2.2.1 considers job and thread scheduling, while Section 2.2.2 considers loop scheduling.

2.2.1 Job and Thread Scheduling

The basic operating system scheduling unit in shared-memory multiprocessors is the kernel thread. Operating system scheduling policies may be classified based on whether or not the operating system takes into account the job identities of the kernel threads that are being scheduled (i.e., whether or not there is any effective policy component to the top level of scheduling described above). At one extreme are policies in which kernel threads are scheduled obliviously to the job identities, using a traditional scheduler for conventional time-shared systems such as that in UNIX. Many current shared-memory multiprocessors directly provide only this form of scheduling. The class of single user policies represents the other extreme where the entire machine is dedicated to the threads of a single job. In between these extremes is the class of policies that are similar to single user policies, but multiple jobs are executed simultaneously and the system processors are partitioned (in time or in space) among these active jobs. This class of policies is commonly referred to as partitioning policies. The allocation of processors to jobs can be either static or dynamic [38] [54] [61] [65].

Scheduling policies belonging to partitioning policies are commonly referred to as two-level scheduling policies as they actively employ both of the top two levels of scheduling described above. Higher level scheduling makes long-term decisions among jobs, such as deciding how the processors are to be partitioned, while lower level scheduling is concerned with the shorter term decisions about the threads of control comprising a single job. In the case of applications employing user-level threads, this lower level scheduling is mostly handled at the user-level.

Two-level scheduling strategies for multiprogrammed, shared memory multiprocessors can be classified along three dimensions [44]. The first, and the most basic, is in the manner in which concurrency among jobs is provided. Under *time sharing* policies, the processors are quickly rotated from one job to another. There are thus alternating periods when a job holds many (possibly all) processors, followed by few (possibly no) processors. Under *space sharing* policies, the processors of the machine are partitioned among the jobs. Space sharing tends to provide each job a more constant allocation of fewer processors than does time sharing. The frequent context switching (in time sharing) can affect process cache behavior. After a context switch a thread may be rescheduled on another processor, without the cache data it had loaded into the cache of the previous processor. Even if the thread is rescheduled onto the same processor, intervening threads of execution may have written some or all of the cache data. Since high cache-hit rates are essential in achieving good processor utilization in modern multiprocessors [36], scheduling strategies that ignore cache effects may severely degrade performance.

The second dimension concerns the frequency with which allocation decisions are made. In a *static* policy, an allocation decision is made for a job at the start of its execution. Whenever the job has any processors, it has the number of processors decided upon initially. Under a *dynamic* policy, the number of processors allocated to each job may vary considerably during the job's lifetime. Although dynamic policies have greater potential to efficiently execute applications with fluctuating levels of parallelism, these policies incur more system overhead, which could lead to degradation in performance.

The final dimension is concerned with the issue of processor reallocation. In policies employing *coordinated* processor reallocation, processor reallocation is performed in concert with the application, while in the *uncoordinated* reallocation, the operating system may remove a processor without interaction with the affected application. Previous work using modeling [66] [67] has shown that uncoordinated preemption can lead to very poor performance. First, many parallel applications use synchronization primitives that require "spin waiting" on a variable (remaining in a tight loop in which the variable's value is read and tested) until it is "set". If the thread that is to set the variable is preempted and its processor taken away, the threads that are waiting for the variable to be set will waste processor cycles. This is an example of a more general problem of preemption of threads critical to the progress of the application, which can seriously affect both the application and the overall system performance.

To address the above issues, a number of different policies have been proposed. For example, to address the processor reallocation issue, gang scheduling [47] ensures that all runnable processes from the same application execute at the same time. When an application is preempted, all of its processes are preempted, so processes will not be wasting processing resources spin-waiting. While gang scheduling handles some problems associated with processor preemption, it may suffer the overhead due to context switching and due to processor cache corruption.

To improve cache behavior, scheduling strategies that use information about the amount of data a process has on each processor's cache have been proposed. A detailed description of these policies appears in the next chapter. Alternatively, strict space-partitioned scheduling [10] may be used, where processes from an application are always scheduled onto the same subset of processors, ensuring that the data used by the processes is always found in the caches (subject to capacity constraints), and more generally that the application has a very stable and predictable environment in which to operate.

The process control strategy proposed by Tucker and Gupta [61] addresses both processor reallocation and cache problems, by partitioning processors into groups and by dynamically ensuring that the number of runnable processes of an application matches the number of physical processors allocated to it. In their paper, the authors show that if the number of runnable processes belonging to a parallel application significantly exceeds the effective number of physical processors executing it, its performance can be significantly degraded. There are a number of reasons for this deterioration in performance. One possible reason is the preemption of processes for which other processes are spin waiting. Other possible reasons for degradation include the overhead of unnecessary context switching, and the problem of processor cache corruption when several processes are being multiplexed on a given processor. A method to control the number of runnable processes associated with an application is proposed. In this approach, the optimal number of runnable processes for each application is determined by a centralized server, and applications dynamically suspend or resume processes in order to match that number.

The Dynamic processor allocation policy [44] is a dynamic, space-sharing policy where processors are reallocated among jobs in response to changes in the jobs' parallelism. Each job requests processors whenever it could use more than it is currently allocated and offers to release processors whenever it cannot make active use of them. This constitutes the basic mechanism of the Dynamic policy: moving processors from jobs that currently have too many processors to jobs that have too few processors. This mechanism would suffice in a workload where the total demand for processors is always equal to the number of physical processors. In practice, there are many workloads where the total instantaneous demands for processors will exceed the number of processors physically present in the system. These cases are handled by using the "least valuable" processors currently available. In general, demand for processors is satisfied as follows:

- First, any unallocated processors are assigned.
- Next, processors that are allocated to some job but that the job is offering to release ("willing to pause" processors) are assigned.
- Finally, if necessary, processors are preempted from the job(s) with the largest current allocation. (Note that if the multiprogramming mix consists of a number

of identical jobs of constant parallelism, this will result in an equipartitioning of the system.)

At the thread scheduling level (and often at other levels as well), a common approach is to employ a single priority queue of runnable tasks. Since accesses to the global queue and obtaining the highest priority task can only be done sequentially, the single queue may become a bottleneck. The biggest advantage of the single queue policies is simplicity: when a task becomes runnable it is placed on the global queue and when a processor becomes idle it executes the highest priority task on the queue. Another key advantage is that a single queue achieves perfect load balancing, as tasks are never waiting at a busy processor while other processors are idle. A third advantage is the ease of making centralized scheduling decisions.

There are two general ways to reduce the overhead associated with using single queue scheduling. The first consists of keeping the global queue but modifying how tasks are removed from the queue, thus preserving the attractive properties of single queue scheduling. The second general approach consists of eliminating the use of a global queue. In the method proposed by Ni and Wu [46], processors are partitioned into k groups where each group has a dedicated queue. When there is a new runnable task, it is randomly routed to one of the k queues, and when a processor is idle, it selects a single task from its dedicated queue. Ni and Wu discuss a method to find the optimal value of k. The inability to fix an optimum value for all types of system workload is a drawback with this approach. By fixing the number of queues to an optimum value for a particular load, the system may perform poorly under different workloads. Also, the method ignores the affinity tasks may have for particular processors. Anderson et al. [5] investigate a number of more practical alternatives for thread scheduling, involving per-processor queues.

There are additional, somewhat subtle problems that arise when the threads being scheduled are user-level threads. Recall from Chapter 1 that threads separate the notion of a sequential execution stream from other aspects of traditional UNIX-like processes such as address spaces and I/O descriptors. While threads can be supported either by the operating system kernel or by user-level library code in the application address space, neither approach has been particularly satisfactory. Although the performance of kernel-level threads is an order of magnitude better than that of traditional processes, it is an order of magnitude worse than that of user-level threads [7]. The relative efficiency of user-level threads is obtained by performing operations at the user, rather than kernel level. This avoids the overhead of entering and exiting the kernel, which can be substantial relative to the amount of work required. Also, since the operations are executed within the context of a particular job, much of the "bullet-proofing" required in kernel code can be avoided in user-level implementations. User-level threads, on the other hand, have suffered from poor performance and even incorrect behavior in the presence of "real-world" factors such as multiprogramming, I/O and page faults [7], that may result in inappropriate scheduling actions owing to a lack of cooperation between operating system and application scheduling. Basically, problems arise because operating system scheduling is oblivious to the fact that what it views as a single thread of control (a kernel thread) is actually supporting many user threads multiplexed on top of it.

Anderson et al. [7] propose *scheduler activations* as a vehicle to provide the required cooperation. Scheduler activations provide for communication and cooperation between operating system and application scheduling. When an underlying kernel thread (termed a "scheduler activation") blocks (on a page fault, for example), another kernel thread is provided to the application to handle the other user threads that were being multiplexed on the now blocked kernel thread, avoiding a loss of effective parallelism. Mechanisms are also provided to facilitate movements of processors to and from the application.

2.2.2 Loop Scheduling

Loops are a rich source of parallelism in scientific code. Parallelizing compilers for sequential programs have been particularly successful in determining when loop iterations can be executed in parallel. Thus, loop scheduling has received considerable attention [22] [30] [34] [39] [43] [49] [56] [62].

The fundamental trade-off in scheduling loop iterations on multiple processors is that of maintaining balanced processor workloads, without excessive scheduling overhead. Loop iterations can be scheduled either statically (at compile time) or dynamically (at run-time) onto the processors of the parallel machine. The major advantage of static scheduling is the reduction in the run-time synchronization and scheduling overhead. Since dynamic scheduling defers the assignment of iterations to processors until run-time, better load balancing in the presence of factors unpredictable at compile time is achieved. The major difficulty in designing dynamic loop scheduling algorithms is in keeping the run-time synchronization and scheduling overhead small, without losing the attractive load balancing properties. In this section, we discuss the different loop scheduling algorithms that have been proposed for loops in which the iterations have no data dependencies among them, and thus can be executed in parallel without constraints (termed "do-all" loops).

Static scheduling adopts the simple strategy of scheduling the iterations on P processors in chunks of (assuming all iterations are of equal expected size) N/P (where N is the total number of iterations in the loop). This algorithm minimizes run-time synchronization overhead, but does not balance the load dynamically. If the iterations have unpredictable and variable running times, or if the processor allocation to the application varies during the course of execution of the loop (as it might in a multiprogramming environment), different processors may finish at widely different times, causing load imbalance. Since the loop finishing time is equal to the latest finishing time of any of the processors executing the loop, the overall finishing time may be greatly inflated.

Alternatively, if we schedule the iterations one at a time dynamically from a single global queue, a strategy called *self scheduling* [56], then there will be N scheduling operations. With self scheduling, a processor obtains a new iteration whenever it is idle, so the processors finish at nearly the same time and the workload is balanced. However, this algorithm may incur a large synchronization overhead if each iteration represents only a small amount of work.

The two scheduling schemes discussed above are extremes. Between these two lie dynamic schemes that attempt to minimize the cumulative contribution of uneven processor finishing times and scheduling overhead. Such schemes schedule iterations (from a single global queue) in chunks of iterations of size greater than one, but less than N/P. Both fixed size and variable size chunking schemes have been proposed in the literature.

Uniform sized chunking [34] reduces the synchronization overhead by having each processor take K iterations, instead of one, whenever it becomes idle. This algorithm amortizes the cost of each synchronization operation over the execution time of K iterations, reducing the synchronization overhead. However, there exists a greater potential for imbalance, as processor completion times may vary by up to K iteration execution times. Choosing an appropriate value of K is quite difficult, and an optimal choice would require detailed information about the loop and the machine environment that may be difficult to obtain in practice.

Guided Self-Scheduling (GSS) proposed by Polychoronopolous and Kuck [49] is intended to achieve simultaneously both low overhead and good load balance, by giving out large chunks (so as to ensure low synchronization overhead) at the beginning, and giving out small chunks (so as to ensure good load balance) at the end. Each time a processor needs to retrieve more work, it obtains a chunk of $\lceil \frac{n}{G} \rceil$ iterations, where n is the current number of unallocated iterations, and the scheduling parameter G is usually set to the number of processors. Assuming all loop iterations take the same amount of time to complete, this algorithm ensures that all processors finish within one iteration of each other. Further, when processors may start executing loop iterations at unpredictable times, the minimal number of synchronization operations are used in this case. GSS may still, however, suffer from scheduling overhead effects for small loops; note that the last few iterations are dispensed singly, for example. GSS may also suffer from load imbalance effects if iteration execution times vary widely, particularly if the execution times of successive iterations are correlated.

Eager and Zahorjan propose *Adaptive Guided Self-Scheduling* (AGSS) [22] to address the potential load imbalance and scheduling overhead deficiencies of GSS. To address the latter, the scheduling granularity is adaptively increased, if the loop iterations are determined (at run time) to be so short that excessive overhead may result. The load imbalance problem is addressed by providing for the *wrapped* assignment of loop iterations, where processors are assigned groups of iterations that are sampled uniformly throughout the iteration space, rather than blocks of consecutive iterations. In this manner, the potential for load imbalance that arises when iteration execution times vary widely, and the execution times of consecutive iterations are correlated, is significantly reduced.

Factoring, proposed by Flynn et al. [30], was specifically designed to handle iterations with execution time variance. In factoring, iterations are scheduled in batches each containing P (the number of processors) equal size chunks. The total number of iterations per batch is a fixed ratio of those remaining and hence the name *factoring*. Because initial chunk sizes are generally smaller in this method (in comparison to GSS when the parameter G in this method is chosen equal to P), it has better load balancing behavior than GSS (when G=P), when the computation times of loop iterations vary substantially.

Like the factoring algorithm, the *tapering* algorithm [39] is designed for loops where the execution time of iterations varies in such a way as to cause load imbalance under GSS. In the tapering algorithm, execution time profile information is used to estimate the average iteration time and the variance in iteration times. These estimates are then used to select chunk sizes that limit the amount of load imbalance that can occur to be within a given bound with a high probability.

Trapezoid self scheduling (TSS) [62] is another scheme that was developed for loops with variance in the execution times of the iterations. The idea is to reduce the synchronization overhead, while still maintaining a reasonable load balance. The algorithm allocates large chunks of iterations to the first few processors, and successively smaller chunks to the last few processors. The first chunk is of size $\frac{N}{2P}$, and successive chunks differ in size by $\frac{N}{8P^2}$ iterations. The difference in the size of successive chunks is always a constant in trapezoid self-scheduling (as opposed to the difference being a decreasing function both in GSS and in factoring). This avoids the many small chunks that occur towards the end in GSS.

Markatos and LeBlanc [43] propose a new loop scheduling algorithm that exploits the *affinity* that loop iterations may tend to have for a particular processor – the one whose local memory or cache contains the required data. The authors claim that by exploiting this affinity, the communication costs can be significantly reduced. This algorithm is explained in greater detail in the next chapter.

2.3 Summary

There are a wide variety of available computer architectures for parallel processing. To take advantage of the capability of these parallel machines, application programs must contain sufficient parallelism, and this parallelism must be effectively scheduled on multiple processors. In this chapter, shared memory computer architectures suitable for parallel processing have been described, as have several approaches to the scheduling problems in shared memory systems.

Chapter 3

Affinity Scheduling

The sustained performance of fast processors is critically dependent on cache performance. Cache performance in turn depends on locality of reference; when the sequence of memory words referenced by a program cannot be entirely stored in the cache, cache misses result. In modern computers, the penalty for a single cache miss might be tens or hundreds of processor cycle times [31]. It is not possible to build a cache that is large enough to hold the working sets of all possible software, nor is it possible to code all software to avoid all cache misses. However, it is possible to design scheduling mechanisms that use the cache (or caches) efficiently.

When an operating system deschedules one process, and starts another running, the assumption of locality, on which good cache performance depends, may be violated because the instructions and data of the newly scheduled process may no longer be in the cache or caches. Mogul and Borg [45] used address traces from a variety of real workloads to study the impact of having to periodically reload the "working set" of a process into the processor cache, owing to the process being descheduled and then later re-scheduled. They found cache reload overheads of up to 8% of the execution time, depending on the workload. This study was confined to process switching on a single processor under a general multiprogramming workload, and did not include operating system activity. On shared memory multiprocessors running computation intensive workloads, cache reload costs can become more significant.

Context switching (and, more generally, the reallocation of units of work among processors) thus has a cost above that of the associated scheduling operations performed by the kernel. In a shared memory multiprocessor, such cache effects may be an important consideration when scheduling, and in particular, it may be important to schedule a unit of work on a processor whose cache is more likely to contain relevant data, rather than on some other processor. Using such processor affinity information in shared memory multiprocessor scheduling has the potential to improve performance, particularly if this information is inexpensive to obtain and exploit.

There are several factors that determine the effectiveness of affinity scheduling. The first factor is the size of the *footprint* [58], that is, the data set that needs to be almost immediately loaded into the cache when a unit of work is scheduled to run on a processor. The second factor is the duration for which a unit of work runs on a processor once it is scheduled. This is a function of the time slice length (if some form of round-robin scheduling is employed) and the expected execution time before blocking or completion. The third factor is the number of intervening units of work (and their footprint sizes) that are scheduled on a processor between two successive times that a unit of work (or a unit of work using the same data) is scheduled on a processor. The cache sizes also play an important role in determining the effectiveness of affinity scheduling. Smaller caches tend to fill up faster, and it is quite possible that a large data set may sweep through the contents of the cache, removing any data that could have been reused later, thus nullifying any potential gains due to affinity scheduling.

The affinity of a specific unit of work for a particular processor may arise from many sources, not just from the cache effects described above, and a number of these types of affinity have been studied in the literature. For example, the affinity may be based on how fast the unit of work can run on a particular processor in an environment of heterogeneous processors. Scheduling problems of this type have received considerable attention in the literature [15] [29] [35]. These studies have produced exact and approximate algorithms to optimize a variety of performance metrics under different conditions. They have also provided insight and understanding of basic principles that underlie the performance of scheduling policies in these environments.

Another form of processor affinity concerns the resources associated with the processors. For example, each processor may have a set of resources available to it and each unit of work may have to execute on a processor that is associated with the set of resources it requires. In this environment, the goal of the scheduler is to optimize some performance metric while satisfying the constraints imposed by the system. This class of scheduling problems has also received considerable attention in the literature [15] [17] [64].

The form of processor affinity focused on here, however, is, as discussed above, based on the contents of processor caches. Specifically, it may be more efficient in a shared memory multiprocessor system to schedule a unit of work on a particular processor than on any other processor, if relevant data already resides in the processor's cache. This type of processor affinity is particularly interesting, because it may diminish with time, and because it is dependent on fairly complex characteristics of the workload. There are a significant number of references in the literature [19] [24] [26] [41] [43] [53] [54] [57] [60] [63] that deal with this type of processor affinity; this work is the subject of the first two sections of this chapter. In the remaining section, two new affinity scheduling algorithms are proposed.

3.1 Principles and Algorithms of Affinity Scheduling

A parallel application executes most efficiently when its workload is evenly distributed among the available processors, and required data is found in local caches or memories. If the workload is not distributed evenly among the processors, processor cycles are wasted while some processors wait for others to complete their work. If most of the data accesses are not to the local cache or memory, then the communication overhead in accessing this data may greatly inflate running times. Policies for load balancing, and for locality management, respectively, address these goals.

Load balancing policies attempt to distribute the workload among the processors as evenly as possible. Simple policies focus only on the initial assignment of work to processors, ensuring that each processor is initially assigned the same amount of work. More complicated policies use run-time assignment (or re-assignments, as in migration policies) of work to dynamically adjust the load throughout execution. As described in the previous chapter, the central work queue model [59] [61] is a commonly used technique in shared memory multiprocessors. In this model, units of work are placed on a central ready queue accessible to every processor. When a processor finishes some work (its unit of work completes, blocks, or is preempted), it removes the next entry from the ready queue. Under this model, no processor is idle unless the ready queue is empty. In a recent work, however, Anderson et al. [5] have shown in the context of user-level threads that a central ready queue can become a bottleneck in these systems if many very small threads are being individually scheduled, and that local per-processor queues can offer significant performance improvements in this case. (Similar considerations motivate techniques such as guided self scheduling in the loop scheduling context; here, however, the size of the schedulable unit is increased rather than distributing the queues, to solve the problem.)

Locality management policies attempt to co-locate processes and the data they access to minimize the communication overhead. In a UMA multiprocessor, where each processor has a private local cache and there is a shared memory space, frequently referenced data is loaded into the cache. In a NUMA multiprocessor, where each processor has a local memory, but can also access non-local memory, data is usually transferred in bulk to the local memory as needed. In either case, the best performance always results when most memory references are satisfied locally. The main goal of locality management strategies is to reduce unnecessary communication caused by a poor placement of work or data.

Although both load balancing and locality management policies attempt to improve the performance of the system, conflicts can arise between them. Load balancing attempts to distribute the load evenly, thus keeping all the processors busy. Locality management attempts to allocate work close to its data. Once a program begins execution, a load balancing strategy might be eager to shift the work from one processor to another so as to alleviate any imbalance as it occurs. A locality management strategy on the other hand might not favor this migration of work, since the work will have already built up state and thus an affinity (by loading the cache or local memory with the relevant data) for the processors on which it is executing. As a result, attempts to evenly distribute the workload can adversely affect locality, while attempts to improve locality of reference can create an imbalance in the workload.

Many current systems, in particular UMA multiprocessors using the central work queue model, resolve this conflict in favor of load balancing. The central work queue ensures that no processor is idle while there is work to be done. A unit of work in the work queue is always assigned to the next available, idle processor, even if there is previously established state on another processor. Resolving the conflicts between load balancing and locality management policies requires an understanding of the potential costs and benefits of the two policies, and the factors that influence their effectiveness.

The locality of reference exhibited by a program and the degree of sharing that takes place determine the extent to which locality management policies can be effective. The load imbalance inherent in the algorithm, or the imbalance introduced by synchronization, determines the extent to which load balancing is necessary. The cost of implementing these policies depends upon the techniques used, which may include an intelligent initial assignment of work to processors, work migration, and bulk data transfer.

Squillante and Nelson [55] consider a fundamental question concerning the task migration tradeoff in general: how expensive must task migration be (in terms of inflated execution times owing to violated affinities) before it is not beneficial to have an idle processor migrate a task waiting at another processor? Clearly, it would be beneficial to migrate such a task if the cost to do so is negligible. This study attempts to identify the points where it becomes detrimental to migrate a task, with respect to the costs of not adhering to processor affinities. The paper presents a queueing model for a particular class of policies that the authors refer to as *threshold scheduling* *policies.* In these threshold policies, each processor has its own local queue and when a processor becomes idle, it randomly chooses another processor and searches that processor's queue for waiting tasks. If this queue contains more than a threshold number of tasks, including the task in service, the processor migrates one of these tasks to its local queue and proceeds to execute the task. Otherwise, the idle processor randomly probes a different processor in the same fashion. The authors show that even when migration is expensive, it often pays to migrate tasks to balance the load. Extensive migration was shown to cause processor thrashing, where processors spend most of their time executing migrated tasks, rather than tasks of their own. The conclusion from this paper was that unconditionally fixing tasks onto the processors to which they have affinity may cause significant load imbalance. This load imbalance then results in fairness problems and idle time.

In another paper, Squillante and Lazowska [53] modeled the phenomenon by which a process develops an affinity for a particular processor during execution based on the contents of the local cache. Using queueing theory techniques they modeled a multiprocessor system running multiple single process applications with different affinity based scheduling policies. These policies covered a wide spectrum in terms of their relative attention to affinity and to load balancing. The policies studied included:

- *First Come First Served (FCFS)* scheduling from a global queue, which completely ignores a task's affinity for a particular processor;
- *fixed scheduling*, where tasks are initially assigned to processors and never migrate to other processors;
- *last processor scheduling*, where a task is scheduled on the processor where it last executed. In this algorithm, when a processor becomes idle, it searches the ready queue for the first *matching* task; i.e., one that last executed there. If a matching task is found, the processor executes it; otherwise, the first task on the ready queue is executed;

- *minimum intervening scheduling*, which attempts to schedule the task with the greatest affinity for a processor whenever a scheduling decision has to be made. When a processor becomes idle, it computes, for each task x on the ready queue, the number of tasks executed since x last executed there. The processor then executes the task with the minimum value;

- *limited minimum intervening scheduling*, similar to minimum intervening scheduling, but limits the number of processors for which a task maintains affinity information.

The results showed that minimum intervening scheduling performed best under light to moderate loads, and that fixed scheduling performed best with heavy loads, where load imbalance was not a problem. They argued that if a process suspends execution for any reason, it should be resumed on the same processor, avoiding migration whenever possible. The analytical model was used to demonstrate that ignoring cache affinity in multiprocessor scheduling can result in significant degradation of program performance.

Thakkar and Sweiger [57] studied the performance of an Online Transaction Processing (OLTP) application on the Sequent Multiprocessor system using the TP1 [4] benchmark. Written in SQL, the benchmark simulates a banking system workload. The TP1 testing revealed a hierarchy of performance bottlenecks that show the limitations of the current hardware and software. Process migration was shown to cause significant performance degradation. The authors propose a time-based scheduling algorithm to solve this problem. In this approach, a process would be scheduled to stay on a processor and would be reinvoked on the same processor if the time duration from the last context switch was less than some (fixed) value. This method of simply attaching processes to processors was found to yield a significant improvement in the performance.

In contrast, Gupta et al. [26] found that cache affinity scheduling of operating system processes has only a limited impact (no more than a 3-4% improvement in application performance) in the multiprogrammed UMA multiprocessor environment studied, when the scheduling quantum of the system is sufficiently large, and applications perform I/O at normal rates. These results were obtained using a relatively short time quantum of 10 ms. The authors conclude that the benefits due to affinity scheduling (at least within the operating system kernel) are likely to be small if the time to load the footprint is small relative to the interval for which the process runs on the processor. The benefits will also be small if the intervening process (or processes) wipe out most of the previous contents of the cache.

Vaswani and Zahorjan [63] reached a similar conclusion. The results were based, however, solely on scientific workloads and on a space-sharing scheduling system that dynamically partitioned processors among applications. As this scheduling strategy produces very long (over 300 ms) effective time slices for scientific workloads, while the time to reload an entire cache was much smaller than this, the small gains for affinity scheduling are understandable. The focus of both the latter papers was on the use of affinity scheduling in the presence of multiprogramming, within the operating system kernel.

Torellas et al. [60] also focus on the issue of kernel process scheduling, in a bus-based multiprocessor (Silicon Graphics POWER Station 4D/340) executing a variety of workloads, including mixes of scientific, software development, and database applications. The standard scheduling algorithm used in the UNIX operating system was modified to incorporate the notion of affinity. The authors found that affinity scheduling reduced the number of cache misses by 7-36%, in the environment studied, resulting in execution time improvements of up to 10%. Although the overall improvements were small, at least affinity scheduling appeared to have no negative impact on the workloads. The paper classifies the workloads that are most likely to benefit from techniques that exploit affinity. These workloads are those whose processes: (1) are costly to reload because of the large amount of reuse data, (2) execute for short effective time slices during which blocking occurs infrequently or not at all, and (3) are in a multiprogramming mix with other processes that replace a large part of the cache when executed.

The paper also compares their affinity scheduling policy to two other means that have been used to increase the reuse of cache state, namely statically attaching processes to processors, and extending the time quantum. Attached scheduling lowers the number of cache misses over standard scheduling, but overall it degrades performance due to the increased load imbalance in the machine, while the drawback with increasing the time quantum is its inability to benefit processes that run only for a short time before blocking and that could benefit from a reduction in process migration. Finally, the authors also propose a more complex implementation of affinity scheduling. In this approach, the scheduling priority of a process is adjusted by an amount proportional to the time since the process was last executed on the processor that is searching the run queue. The gains from this implementation (when compared to the performance of the basic affinity scheduling algorithm) varied from 0-7% (depending upon the load and workload characteristics) and must be traded off with the disadvantages of a more complex implementation.

A study of cache affinity scheduling with the help of synthetic workloads by Devarakonda and Mukherjee [19] emphasized the point that previous studies showing negative results for affinity scheduling were with respect to kernel level affinity scheduling (coarse grained kernel threads) and that the kernel may not be the appropriate place to implement affinity scheduling. This study also pointed out several problems that arise from inappropriate implementation, such as the development of *pseudo-affinity* at the synchronization points if threads temporarily switch processors in synchronization operations. Since the system has no way of recognizing the switch as temporary, it will reschedule the thread on the processor on which it last ran, which is not the one on which it has the body of the cache footprint. They also point out that the short lifespan of threads in lightweight computation environments does not allow for cache reuse, and conclude that such threads are not good candidates for affinity scheduling.

The research efforts surveyed so far have considered affinity as built up through a process' own execution on some processor prior to being context-switched out. A distinguishing characteristic of the study by Markatos and LeBlanc [41] is that it concerns the scheduling of a thread initially on the "right" processor and affinity is considered to be built up by other threads, or through an initial data allocation. Markatos and LeBlanc observe that such locality conscious scheduling has great potential in shared memory multiprocessors and that scheduling threads close to their data can effect a large improvement in performance. This work considers the role of cache affinity for threads *within* an application, rather than in the context of multiprogramming. An optimistic approach to both load balancing and locality management is proposed. In the proposed *memory conscious* scheduling algorithm, a thread is assigned to a processor depending on the location of its data. Although it may be unlikely that any single memory (in an environment with physically distributed memories) will contain all the data needed by a thread, some large portion of its data should be local, resulting in fewer block transfers during execution. If load imbalance results, then a thread is reassigned to the idle processor. The results from this paper demonstrate that both load balancing and locality management can significantly improve performance. Load balancing policies that ignore locality in the initial assignment of threads to processors were shown (on the BBN Butterfly multiprocessor) to introduce an overhead of 12% when the remote reference penalty is very small, and up to 45% when the remote reference penalty is large.

Fowler and Konthothanassis [24] adopt a similar perspective on scheduling. They observe that since lightweight computations do not accumulate much history during their short lifetimes, lightweight thread scheduling policies that try to improve locality have generally met with limited success, as they usually base their decisions on the past history of the schedulable entity. The authors propose *object-affinity scheduling* (OAS) as a means to effectively address the locality issue in fine grain parallel programs. The scheduling strategy is integrated into Mercury [33], an object oriented parallel programming environment derived from Presto [9]. Due to the object oriented nature of Mercury, each thread has an object that is associated with it. This object usually contains data that the thread accesses and in many cases has already been used by another thread. Under such circumstances, it is beneficial to schedule a new thread on the same processor as that for previous threads associated with the same object, since there is likely to be a significant cache affinity between the thread and that processor's cache. This is the key idea in OAS. The main objective of OAS is to provide the conceptual advantages of abstract programming models that support relatively fine-grain parallel decomposition, while at the same time striving for high performance by automatically scheduling each thread for locality based on dynamic object location information.

The COOL programming language [12], like Presto and Mercury, extends C++. In COOL, calling a function prefixed with the keyword **parallel** forks a thread to execute the function asynchronously. Join synchronization is done explicitly using condition variables. Intended for use on distributed memory machines such as DASH [36], COOL uses a form of object affinity scheduling that, by default, runs the thread on the processor whose memory holds the invoked method's object. The programmer can also explicitly specify an affinity to some other object as well as create task affinity sets that cause the members of the set that are on the same node to be scheduled back-to-back to exploit temporal locality.

Markatos and LeBlanc [43] studied affinity scheduling in the context of loop scheduling on shared-memory multiprocessors, and proposed a new loop scheduling algorithm that simultaneously attempts to balance the workload, minimize synchronization, and co-locate loop iterations with the necessary data. This work is relevant for the specific scenario of a parallel loop embedded within an outer sequential loop; a context that is also assumed for the research in this thesis. The main motivation for affinity in loop scheduling is to reduce the time spent in bringing data into the local memory or cache. This can be a significant source of overhead, and it has been shown that it may consume 30-60% of the total execution time in this context [11] [26] [41]. By scheduling loop iterations on processors whose local memories or caches already contain the necessary data, the communication time can be significantly reduced. In order for the loop iterations to have an affinity for a particular processor,

1. the same data has to be repeatedly used by successive executions of an iteration (as in the case of an inner parallel loop within a outer sequential loop), and,

2. the data is not removed from the local memory (or cache) before it can be

reused.

While questioning the validity of the central work queue as a scheduling mechanism for loop iterations, the main conclusion from this paper is that loop scheduling algorithms for shared memory multiprocessors cannot afford to ignore the location of data. The location of data assumes added significance in light of the increasing disparity between processor and memory speeds [42]. However, the work does suffer from two significant limitations. As previously mentioned, the proposed algorithm is specific to the case of a single parallel loop nested within a serial loop. It does not address more general program structures, such as multiple parallel loops with different data access patterns nested within a single enclosing sequential loop. The potential of the affinity scheduling policy studied by the authors may be limited in such instances. Further, this work does not address the case where the workload exhibits both great load imbalance and affinity effects; this later limitation is addressed in the research in this thesis.

3.2 Impact of Architectural Trends on Affinity Scheduling

The impact of architectural trends on affinity scheduling has been investigated by several researchers in the literature. In this section, the important results from these investigations are briefly reviewed.

Markatos and LeBlanc [42] attempt to quantify the effect of the emerging trends (hardware and software) in shared memory multiprocessors and the implications for parallel program performance. The authors opine that during the time period in which processor speeds have improved by two orders of magnitude, communication bandwidth improved by at most one order of magnitude. An important ramification of this growing disparity between processor speed and communication bandwidth in shared memory systems is the potential inability of parallel programs to exploit further increases in processor speed, as such increases in processing speed are not accompanied by corresponding increases in communication bandwidth. Advances on the software front have progressively reduced the cost of creating threads of control. Very cheap threads enable fine grain parallel programming, which in turn exposes additional opportunities for synchronization, load imbalance and communication overhead. With the development of efficient synchronization primitives and scheduling algorithms based on the central work queue, the first two problems can be mostly solved. Communication overhead will thus be left as the principal source of overhead, and a major limitation to parallel program performance.

Vaswani and Zahorjan [63] use an analytical model to evaluate the effect of processor affinity on future machines. These future machines are assumed to be built using faster processors and having larger caches. The results from this study lead to the conclusion that it is advisable to include affinity information in current scheduling policies because this extra consideration does not currently degrade performance, and since affinity scheduling will become increasingly important.

Current research has concentrated on the development of large scale multiprocessor systems with a single address space and coherent caches. Section 2.1.2 described these architectures in detail. Recall from that section that in COMA architectures, the location of a data item in the machine is fully decoupled from its physical address, and the data item is automatically migrated or replicated in main memory depending on the memory reference pattern. Due to the large sizes of the cached main memories (32 MB in the KSR1), typically the working set of a unit of work may be held in the memory of the particular processor on which it is run. Thus, affinity scheduling considerations are expected to be particularly important on these machines.

3.3 Proposed Affinity Scheduling Algorithms

This research is concerned with the use of affinity information in loop scheduling. Specifically, a context is assumed in which a parallel loop is nested inside an outer sequential loop and is thus repeatedly executed, as in [43]. Of primary interest is the case in which the parallel loop being executed is unbalanced with respect to the amounts of computation represented by each iteration. The design issues involved in the affinity scheduling of severely unbalanced workloads have not been investigated in the literature and it is this anomaly that the thesis research addresses. The major drawback of the existing loop scheduling algorithms that exploit affinity in this context [43] is their inability to handle workloads that exhibit both great load imbalance and affinity.

For example, consider the common case of iterations over a triangular space (for example, in the backward substitution method employed in Gaussian Elimination/LU Decomposition), where the iteration execution times decrease (or increase) monotonically in iteration number as shown in Figure 3.1; suppose such a loop is nested inside an outer sequential loop and is thus repeatedly executed. The algorithms proposed in the literature always allocate, to each processor, a fixed partition of consecutive iterations of size $\frac{N}{P}$, where N is the number of iterations in the loop, and P is the number of available processors. As different processors have different amounts of computation

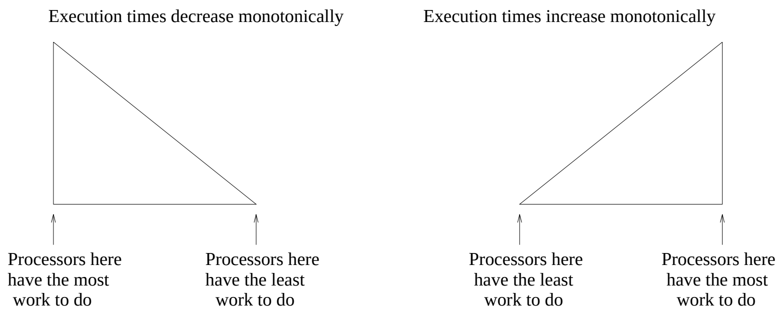

Figure 3.1: Iterations over a triangular space

to be performed in this case, a uniform initial allocation of iterations to all processors results in a situation where there are lightly loaded processors that finish well ahead of the other processors. In the algorithm proposed by Markatos and LeBlanc [43], idle processors probe the work queues of other processors and remove $\frac{1}{P}$ of the iterations from that of the most loaded processor. This migration of work from the most loaded processors to the idle processors results in a cache-reload overhead that is experienced every time the loop is executed, since there is no mechanism to ensure that the same iterations are migrated to the same processors. It is quite possible that this overhead might actually nullify any potential gains that result from exploiting affinity.

This thesis proposes two algorithms that combine affinity information and load balancing techniques in the context of loop scheduling. The key concept behind the first of the proposed algorithms is the readjustment of the sizes of the allocated partitions on subsequent executions of a loop. The key concept behind the second algorithm is to assign to each processor iterations that are dispersed throughout the iteration space, rather than consecutive iterations.

Both the proposed algorithms are based on the affinity scheduling algorithm proposed by Markatos and LeBlanc, which is termed here "static partitioned affinity scheduling" (or often simply "static".) Figure 3.2 gives a pseudo-code description of this algorithm. The algorithm divides the iterations of a loop into partitions of size $\frac{N}{P}$ iterations, where N is the number of iterations in the loop, and P is the number of available processors. The assignment of partitions to processors is done in the *assign_iterations* procedure, and results in the initial values for per-processor data values *lower* and *upper* giving the loop index of the next iteration that should be handed out, and the upper limit of the loop partition, respectively. In the loop execution, each processor removes $\frac{1}{P}$ of the iterations from its local work queue (as implemented by *lower* and *upper*) and executes them. If a processor's work queue is empty, it finds the most loaded processor, removes $\frac{1}{P}$ of the iterations from that processor's work queue, and executes them.

```
loop_initialization(N,P)
// executed by one processor at the beginning of the loop
// N is the number of loop iterations, P is the number of processors
{
       for (i = 0; i < P; i++) {
        // assign iterations ceil(i*N/P) to min(N, ceil((i+1)*N/P) - 1)
        // to processor i
        assign_iterations(i)
       }
}
loop { // executed by each processor
       // get 1/P of the local iterations to execute
       range = get_iterations(myproc, 1/P);
       if (range == empty) break;
       execute (range);
} forever
loop { // executed by each processor
       // find the most loaded processor
       max_load = find_most_loaded_processor();
       // get 1/P of the iterations from the most loaded processor
       range = get_iterations(max_load, 1/P);
       if (range == empty) break;
       execute (range);
} forever
```

Figure 3.2: Pseudocode for Static Partitioned Affinity Scheduling

The two algorithms proposed in this research are described below:

1. **Dynamic Partitioned Affinity Scheduling:** The basic idea in this approach is to keep track of the number of iterations that were actually executed by each processor in the previous execution of the loop. This serves as an indication of the actual load distribution in the loop and is used in the reallocation of the iterations for subsequent executions.

   There are three distinct phases in the proposed algorithm (the first of which is done only once, the remaining two of which are repeated for each loop execution):

   (a) Loop initialization phase: This phase is identical to the initialization phase in Figure 3.2 where the iterations of a loop are divided into partitions of size $\frac{N}{P}$ iterations; in this policy, however, this is only done for the first execution of the loop.

   (b) Loop execution phase: A processor removes $\frac{1}{P}$ of the iterations from its local work queue and executes them. If a processor's work queue is empty, it finds the most loaded processor, removes $\frac{1}{P}$ of the iterations from this processor and executes them. Every processor i keeps track of the actual number of iterations that it executed in the ith element of an array *executed* (of size P).

   (c) Re-initialization phase: At the beginning of all but the first execution of the loop, a dynamic readjustment of the initial partition size is achieved by calculating the new partition for processor i (assuming processors are indexed 0 through P - 1, and iterations are indexed from 0) as :

   <u>when i = 0</u>

   partition_start[i] = 0;

   partition_end[i] = executed[i] - 1;

<u>when i > 0</u>

partition_start[i] = partition_end[i-1] + 1;

partition_end[i] = partition_start[i] + executed[i] - 1;

By dynamically changing the sizes of the initial processor partitions, the proposed algorithm is more capable of handling workloads that are unbalanced with respect to the amounts of computation represented by each iteration. This claim assumes that the algorithm "converges" (this must be verified experimentally) in the sense that eventually the initial partitions stabilize to roughly constant ranges, thus yielding good affinity behavior.

2. **Wrapped Partitioned Affinity Scheduling:** Recall from Chapter 2 that Eager and Zahorjan [22] propose the wrapped assignment of iterations to rectify a major shortcoming of the Guided Self-Scheduling (GSS) [49] algorithm; specifically, the load imbalance that arises when consecutive iteration execution times vary widely, but in a correlated manner. The underlying idea behind this wrapped allocation of iterations is to allocate chunks of iterations that are essentially at a distance P (the number of processors in the system) from each other as opposed to consecutive iterations that are at a unit distance as in blocked allocation. We propose to incorporate this concept of wrapped allocation of iterations into the previously described static partitioned affinity scheduling approach. The basic idea would be to create a wrapped iteration space and apply partitioning within this wrapped space. The wrapped assignment of iterations has been shown to be capable of efficiently handling loops where the iteration sizes vary widely, and the lengths of consecutive iterations are correlated [22].

   Most of the implementation details of this algorithm are identical to the static partitioned affinity scheduling approach. The key difference, however, is in the loop execution phase. The wrapped assignment is implemented by mapping the consecutive iteration numbers provided to each processor by the initialization phase into the iteration numbers that correspond to the desired wrapped allocation.

3.4 Summary

This chapter considered the issue of affinity scheduling and the different approaches to affinity scheduling were discussed. In particular, the issue of affinity resulting from processor-cache contents was considered. Ignoring processor cache affinity in scheduling may have significant performance implications. With continuing increases in the relative cost of a cache miss, disregarding the cache reload time in scheduling decisions may cause significant increases in the execution times of individual units of work. Performance degradation in the system as a whole is also likely because of the increased bus traffic due to the cache misses. With the increasing popularity of the cached main memory architectures, the implications of a larger cache reload transient time will be more pronounced in the future. It is therefore essential to ensure that there is greater reuse of the cached state. As one of the means to achieve this greater reuse, affinity scheduling may hold great promise on these machines.

The research in this thesis considers affinity scheduling of parallel loops, and addresses the specific case in which the loops being executed are unbalanced with respect to the amounts of computation represented by each iteration, a situation in which simple forms of affinity scheduling may become inappropriate. Two new affinity scheduling algorithms are proposed to address this issue. The next chapter describes the experimental study of the proposed algorithms.

Chapter 4

Experimental Study

This chapter describes an experimental study that was carried out to assess the relative performance of the two new affinity scheduling algorithms proposed in Chapter 3. Section 4.1 outlines in more detail the goals of the study. The research methodology is described in Section 4.2. Sections 4.3 and 4.4 present the results and the conclusions from these results, respectively. Section 4.5 summarizes the chapter.

4.1 Goals

The primary goal of this research is to assess the performance of the two affinity scheduling algorithms proposed in Chapter 3, relative to that of the *static* partitioned affinity scheduling algorithm. The two proposed affinity scheduling algorithms, together with the *static* and the *Guided Self Scheduling* (GSS) algorithms were implemented on a Silicon Graphics eight-way multiprocessor (in the Department of Computational Science at the University of Saskatchewan) and their performance studied. The most important metric used to compare the relative performance is the *application execution time* using the proposed algorithms. The second metric concerns the *scalability* (application run time for bigger problems on bigger systems) of the proposed algorithms. This metric is particularly relevant, given the rising popularity of large scale shared memory machines like the KSR1 [51] and DDM [27]. The scalability is heavily dependent upon the frequency of remote operations. Scalability is thus measured by counting the *number of iterations migrated* as a result of the proposed algorithms. Recall that one of the major aims of the proposed algorithms is to reduce the number of migrations and thereby reduce the overhead of the resulting cache-reload transient that accompanies every such migration.

The GSS algorithm was included in the experimental study so as to provide an additional reference point by which to interpret the results for the other algorithms. As previously explained in Chapter 2, GSS is a dynamic algorithm in which iterations are allocated from a single global queue. Large chunks of iterations are allocated to processors at the beginning of a loop so as to reduce synchronization overhead, while small chunks are allocated towards the end of the loop to balance the workload. Under GSS, when each processor retrieves work from the queue it takes $\lceil \frac{n}{G} \rceil$ of the n remaining iterations, where G is typically chosen to be equal to P, the number of processors. While this ensures that all processors finish within one iteration of each other, GSS may still, however, suffer from scheduling overhead effects for small loops; note that the last few iterations are dispensed singly, for example. Further, since the first chunk in GSS contains $\lceil \frac{N}{G} \rceil$ of the iterations, where N is the number of iterations in the loop, the remaining iterations may not have enough work to balance the load, if the loop is unbalanced and $G = P$. In the particular implementation of the GSS algorithm used in this research, each processor takes $\lceil \frac{n}{2P} \rceil$ of the remaining iterations so as to address this latter problem in part. With this change, GSS now starts with smaller chunks, increasing its ability to appropriately schedule unbalanced loops, without introducing significant additional synchronization overhead. As the GSS algorithm does not exploit any affinity, its performance relative to the other affinity scheduling algorithms provides a yardstick by which the performance of the affinity scheduling algorithms can be measured.

In addition to the general goal of assessing the relative quality of the two proposed algorithms, other goals of the study were to observe:

1. The effect of the structure and extent of load imbalance in the parallel loop being scheduled, on the application execution time and the scalability of the

proposed algorithms.

2. The effect of the working set sizes on the application execution time and the scalability of the algorithms.

3. The effect of varying the number of processors in the system on the application execution time.

4. The "convergence" rate of the dynamic algorithm; i.e., the number of parallel loop executions (within an outer sequential loop) required before the initial partition of the dynamic algorithm becomes roughly constant.

4.2 Research Methodology

All four scheduling algorithms (*static, dynamic, wrapped* and *GSS*) were implemented on a SGI 4D/380S - an eight-way multiprocessor running IRIX version 4.0.5. The IRIX operating system is based on AT&T's System V.3 UNIX with enhancements to support multiprocessing. The 4D/380S is a bus-based, cache-coherent, eight processor machine. Each processor in the system consists of a 33 MHz MIPS R3000 CPU and a R3010 floating point coprocessor with private instruction (64 KB) and data (first level: 64 KB, second level: 256 KB) caches. The algorithms were developed using the 'C' programming language and made extensive use of the IRIX multiprocessing library. All experiments were performed with the system in "single user" mode. In this experimental state, all unnecessary daemons are turned off and no further logins are permitted. This ensures that approximately the same state is maintained on every run of the experiments by providing a constant operating environment.

4.2.1 Workload

The proposed algorithms are evaluated using both a synthetic application where the parameters may be varied to allow exploration of algorithm performance under a wide variety of conditions, and also a sample real numerical application to which they are applicable. Each of these is described in turn:

1. A synthetic application. Since one of the primary goals of this thesis is to investigate the performance of affinity scheduling algorithms in the context of unbalanced workloads, the synthetic application incorporates a parallel loop in which the structure and extent of load imbalance may be varied. The application is also parameterized to allow variation of the number of loop iterations, as well as the number of executions of the loop. The synthetic application considered in this research has the following structure:

```
for k = 1 to K { // outer sequential loop
    for i = 1 to N { // parallel loop
        // f is a function
        seq_lim = f(i);
        for j = 1 to seq_lim { // inner seq. loop
            // perform a write operation
            A[i, j] = 1;
        }
    }
}
```

Figure 4.1: Outline of the synthetic application

The upper bound K of the outer sequential loop on k controls the number of executions of the parallel loop. Note that for small values of K, the initial partition sizes used by the *dynamic* algorithm may not have yet converged to reasonable values. In such a situation, it is possible that the readjustment of partition sizes may not bring about the desired benefits. The working set size and the degree of parallelism can be varied by changing the upper bound N of the parallel loop. The upper bound of the inner sequential loop on j in the i^{th} iteration is computed using a function f that takes as input the current value of the parallel loop index i. Forms of f are chosen so as to result in a roughly constant total working set size and computational requirement, across all runs with a given value of N. The form of f chosen (and thus the specific upper bounds of the inner sequential loop on j) controls the structure and extent of load imbalance.

The function f is varied in the experiments (as are the other simple parameters K and N). These parameters are used to generate different types of workloads to evaluate the performance of the proposed algorithms under a wide variety of conditions.

2. Jacobi iterative algorithm: Given a non-singular matrix A and an n-dimensional vector b, we want to solve the system of linear equations $Ax = b$ for the unknown n-dimensional vector x. The Jacobi algorithm starts with an initial approximation of the solution vector, $x^{(0)}$, and repeatedly computes a new estimate $x^{(t)}$ from the previous estimate $x^{(t-1)}$ using the formula:

$$x_i^{(t)} = \frac{\sum_{(j=1, j \neq i)}^{n} (-a_{ij} x_j^{(t-1)}) + b_i}{a_{ii}}$$

The algorithm terminates when the estimates converge or after some pre-determined number of iterations have been performed. The basic algorithm follows:

INPUT the number of equations and unknowns n; the entries a_{ij}, $1 \leq i, j \leq n$ of the matrix A; the entries b_i, $1 \leq i \leq n$ of the n-dimensional vector b; the entries XO_i, $1 \leq i \leq n$ of $XO = x^{(0)}$; tolerance TOL; the maximum number of iterations N.

OUTPUT the approximate solution $x_1, \ldots, x_n$ or a message that the number of iterations was exceeded.

```
converged = FALSE;

For k = 1, ... ,N            // sequential loop

      For i = 1, ... ,n     // parallel loop
```

$$x_i = \frac{-\sum_{(j=1, j \neq i)}^{n} (a_{ij} XO_j) + b_i}{a_{ii}}$$

```
      If ||x - XO|| < TOL
      then
            printf (x1, ... ,xn);
            converged = TRUE;
            break;
            (Procedure completed successfully)
      else
            for (i = 1, ... ,n)
            XO_i = x_i;
if (!converged)
then
      printf(``Maximum number of iterations exceeded'');
      (Procedure completed unsuccessfully)
```

Note that since the i^{th} iteration of the parallel loop always accesses the i^{th} row of the matrix, there is a significant amount of data locality that can be exploited. As we are concerned with applications that also have significant amounts of load imbalance, we consider the use of the Jacobi algorithm on problems in which most of the a_{ij} are zero; i.e., the matrix A is sparse. In this case, a good implementation will avoid storing the zero elements explicitly, and avoid multiplication with them. The parallel loop part of an example implementation is shown in Figure 4.2. Only the non-zero coefficients are stored in the array *nzCoeffs*, while the array *nzColumns* records which columns they belong to. Element *firstNz[i]* indicates where the information for row i starts in *nzCoeffs* and *nzColumns*.

```
// N is the number of rows in the input matrix
// MAX_NONZEROS is the total number of non zero elements
// in the input matrix
int firstNz[N];
int nzColumns[MAX_NONZEROS];
double constant[N];
double nzCoeffs[MAX_NONZEROS];
for (i = 0; i < n; i++) { // parallel loop
    sum[i] = 0;
    for (nz = firstNz[i]; nz < firstNz[i+1]; nz++) {
        if (i != nzColumns[nz])
           sum[i]+= nzCoeffs[nz] * X0[nzColumns[nz]];
    }
    x[i] = (constant[i] - sum[i])/a[i][i];
}
```

Figure 4.2: Jacobi algorithm for sparse matrices

The parallel i loop is enclosed within an outer sequential loop (not shown in the figure) that iterates until convergence is reached or until some pre-determined number of iterations have been executed, as indicated previously.

4.3 Results

The primary goal of the experimental study is to assess the performance of the two proposed algorithms relative to the static algorithm. As explained in the previous chapter, the major drawback of the existing loop scheduling algorithms that exploit affinity [43] is their inability to handle workloads that exhibit both great load imbalance and affinity. Specifically, these algorithms do not provide for the readjustment of partition sizes in the event of load imbalance. The resulting migration of work from the most loaded processors to the idle processors results in a cache-reload transient that is experienced every time the loop is executed since there is no mechanism to ensure that the same iterations are migrated to the same processors. It is quite possible that this overhead might actually nullify any potential gains that arise from exploiting affinity. The two loop scheduling algorithms proposed earlier attempt to respect affinity and achieve good load balance, even for severely unbalanced workloads.

Results are reported here for experiments using both the synthetic application and the Jacobi iterative algorithm, described in the previous section. Results for the synthetic application include cases of varying the numbers of iterations and iteration sizes. With respect to iteration size variability, the following three cases were considered:

1. Loops in which the iteration size decreases linearly (henceforth referred to as the "triangular workload"). The upper bound of the inner sequential loop j in the i^{th} iteration is set to (N - i + 1), where N is the size of the parallel loop. (Results are reported in Section 4.3.1.)

2. Loops in which a fraction of the iterations are of a (constant) large size, while the other (remaining) fraction has (constant) smaller size (henceforth referred to as the "rectangular workload"). The generic pseudocode of the parallel loop body is given in Figure 4.3. Workloads that vary significantly in the amount of load imbalance are generated by varying the value of k. In the experimental results reported in Section 4.3.2, values of k of 2, 4 and 8 were used.

```
if (i < par_loop_size/k) // i is the parallel loop index
   seq_lim = par_loop_size * (k/2);
else
   seq_lim = 1;

for j = 1 to seq_lim
   A[i, j] = 1;
```

Figure 4.3: Generic pseudocode (for rectangular and balanced workloads)

3. Loops that have a constant iteration size. This workload is achieved using the pseudocode shown in Figure 4.3 by setting the value of $k = 1$. (Results are reported in Section 4.3.3.)

Experiments were performed for outer sequential loop bounds of 100, 500, 1000 and 5000, although only the results obtained using an outer sequential loop bound of 500 are presented in Sections 4.3.1, 4.3.2 and 4.3.3, as the results for the other cases are very similar in nature. (In Section 4.3.5, results for various smaller outer sequential loop bounds are presented so as to assess the convergence rate of the dynamic algorithm.) The parallel loop bound sizes considered are 128 and 1024.

Results for the Jacobi algorithm, as described in the previous chapter, are also reported here, for varying input matrix sizes. In this particular algorithm, since successive executions of an iteration of the parallel loop access exactly the same matrix elements, there is a significant amount of data locality that can be exploited. Since the implementation of the Jacobi algorithm used is essentially for sparse matrices, the application may also exhibit significant amounts of load imbalance. Experiments were performed using two different matrix sizes (128×128 and 1024×1024, each with approximately 15% sparsity). The elements of the matrix were generated randomly using a function that maintained a correlated variation in the number of non-zero elements between any two successive rows of the matrix. Figure 4.4 shows the pseudocode outline of this function. All elements not assigned to by this procedure remain with a value of zero, except the diagonal elements which are set to non-zero values. (Results are reported in Section 4.3.4.)

```
number = par_loop_size/4; // #non-zero elements in a row
// ‘i’ loop over all the rows of the matrix
for i = 1, NUM_ROWS, i+=8 {
    for k = i to i + 8 {
        for j = 1 to number {
            // assign random values to the coefficients
            a[k][j] = rand();
        }
    }
    number-= 2;
}
```

Figure 4.4: Function to generate correlated number of non-zero elements

4.3.1 Triangular Workload

In the triangular workload, the iteration size decreases linearly. Figures 4.5 and 4.6 present the completion time (in seconds) of this form of the synthetic application, for parallel loop values of 128 and 1024, respectively. As the figures indicate, *GSS* performs the worst among all the four scheduling algorithms. This is because the *GSS* algorithm uses a single work queue and hence does not exploit any affinity information. The performance of the *GSS* algorithm has been included in these results so as to serve as a baseline by which the results for the affinity scheduling algorithms can be interpreted. As the results indicate, *static* performs the worst among the three affinity algorithms under comparison. In the case of iterations that decrease linearly in size, a uniform initial allocation of iterations to all processors (as is done in the *static* approach) results in a situation where the lightly loaded processors finish ahead of the other processors. These idle processors now probe the work queues of other processors and remove $\frac{1}{P}$ of the iterations from that of the most loaded processor. The cache-reload transient that arises as a result of the migration of work from the most loaded processors to the idle processors partially nullifies the resulting gains from exploiting affinity. The *dynamic* and *wrapped* algorithms result in a much improved initial load balance, minimize migrations and hence reduce the cache-reload transients. This hypothesis is confirmed by the results in Figures 4.7

and 4.8 which present the number of iterations that were migrated. (Note that the number of iterations migrated in these figures is the sum over all 500 iterations of the outer sequential loop.)

Figures 4.9 and 4.10 present the performance of the *dynamic* and *wrapped* algorithms in terms of the completion rate improvement (%) over that of the *static* algorithm. The completion rate improvement is calculated as:

$$\left(\frac{\frac{1}{completion\_time} - \frac{1}{static\_completion\_time}}{\frac{1}{static\_completion\_time}}\right) \times 100$$

($Completion\_time$ represents the completion times of either the *dynamic* or the *wrapped* algorithm, as the case may be.) The above expression translates into:

$$\left(\frac{static\_completion\_time - completion\_time}{completion\_time}\right) \times 100$$

From Figures 4.9 and 4.10, the *wrapped* algorithm achieves a maximum speedup of approximately 50%, while the *dynamic* algorithm achieves a maximum speedup of approximately 25% when compared to that of the *static* algorithm. As Figures 4.9 and 4.10 indicate, there is hardly any difference in the execution times of the three affinity scheduling algorithms when the number of processors is less than three and the number of parallel loop iterations is 1024, in contrast to the case when the parallel loop bound is 128. This is because the loop working set for large values of the parallel loop size does not fit in two caches, but does fit in three. Thus, there are significantly fewer cache evictions on three or more processors in this context; once the caches can contain the entire working set, affinity scheduling reduces the need for any main memory accesses.

Note that increasing the number of processors beyond three has little effect on the relative performance of the two proposed algorithms for large parallel loop sizes. For smaller parallel loop sizes, the relative performance of the *wrapped* algorithm decreases as the number of processors is increased as shown in Figure 4.9. This is due to the per-chunk overhead in the *wrapped* algorithm. For larger loop sizes (and hence larger chunks of iterations), this overhead has less of an effect as is evident in Figure 4.10.

4.3.2 Rectangular Workload

With the rectangular workload, more skewed loops can be studied. Different degrees of imbalance are achieved by assigning $k = 2$, 4 and 8 in Figure 4.3. Figures 4.11–4.28 present the performance of the scheduling algorithms for this workload. Again, the performance of the *dynamic* and *wrapped* algorithms is significantly better than that of the *static* algorithm. As the results from Figures 4.13, 4.14, 4.19, 4.20, 4.25 and 4.26 indicate, both of the proposed algorithms significantly reduce the number of iterations that need to be migrated, thus greatly enhancing the potential for processor cache affinity.

Figures 4.11 and 4.12, along with Figures 4.15 and 4.16, present the results for the rectangular ($k = 2$) workload. These figures are very similar to those for the triangular workload (Figures 4.5 and 4.6, along with 4.9 and 4.10), which is explained by the fact that for $k = 2$ the workload distributions are crudely similar (for example, the maximum and the minimum iteration sizes are the same). As for the triangular workload, the *wrapped* algorithm achieves a maximum speedup of approximately 50%, while the *dynamic* algorithm achieves a maximum speedup of approximately 25% when compared to that of the *static* algorithm. For smaller parallel loop sizes (as shown in Figure 4.15), there is a decrease in the performance of the *wrapped* algorithm as the number of processors is increased. This again is similar to the situation in the triangular workload described in the preceding section.

Figures 4.17 and 4.18 show that there is a significant degradation in the performance of both the GSS and the static algorithms for larger values of k. Consider the generic pseudo-code shown in Figure 4.3. When $k = 4$, the first $\frac{1}{4}^{th}$ of the iterations contain nearly all the work. Although (in the implementation used here) GSS assigns only $\frac{1}{2P}$ of the iterations to the first chunk of iterations that it dispenses, those iterations in this case contain nearly all of the total work; the remaining iterations do not have enough work to balance the load. This is the reason why there is no significant difference in the performance of both the GSS and the *static* algorithms when the number of processors is increased from 1 to 2. Although the *static* algorithm uses per-processor task queues, the first chunk taken by the first processor still contains nearly all the work (as in GSS). In the case of the *dynamic* algorithm, this condition is experienced only on the first few iterations of the outer sequential loop. The dynamic readjustment of partition sizes ensures a better load balance in the system on successive iterations of the outer sequential loop. As the *wrapped* algorithm allocates iterations that are at a distance P (the number of processors in the system) apart, this avoids assigning all the time-consuming iterations to a single processor or chunk, and thus minimizes the chances of load imbalance.

For reasons similar to those elaborated above, there is a marked degradation in the performance of both the GSS and *static* algorithms in Figures 4.23–4.24 when $k = 8$. In this situation, there is a greater load imbalance as the first $\frac{1}{8}^{th}$ of the iterations contain nearly all the work and hence there is virtually no improvement in the performance of both the GSS and the *static* algorithms for smaller number of processors.

These results highlight both the shortcomings of the *static* algorithm and the advantages of the *dynamic* and *wrapped* algorithms when used for severely unbalanced workloads. As the results indicate, for a small number of processors, the performance of the *static* algorithm is comparable to that of the GSS algorithm (which clearly has the worst performance figures).

4.3.3 Balanced Workload

In order to estimate the amount of overhead (if any) as a result of either the wrapped or the dynamic algorithms, experiments were conducted using perfectly balanced loops (loops with a constant iteration size, $k = 1$ in Figure 4.3). Figures 4.29 and 4.30 present the completion times for each of the algorithms. There is basically no difference among the execution times of the three affinity scheduling algorithms. This indicates that the additional overhead due to the *wrapped* and *dynamic* algorithms is negligible. Figures 4.31 and 4.32 present the number of iterations migrated. Recall from Section 4.3.1 that the number of iterations migrated in these figures is a sum over all 500 iterations of the outer sequential loop. So as can be seen, the number of migrations is very small. The *wrapped* algorithm has the fewest migrations, perhaps because there is a little computational overhead added to the loop (due to the wrapped allocation of iterations) that may tend to decrease the effect of variability owing to cache and/or bus queueing delays. On the other hand, the *dynamic* algorithm has a comparatively larger number of task migrations when compared to the other two algorithms. This is because the *dynamic* algorithm follows a rather aggressive policy in varying the initial partition sizes resulting in a situation wherein the initial partition sizes may often be slightly unequal, unjustifiably so. However, this does not appear to have any significant detrimental effects as there is no appreciable difference in the execution times of the three affinity scheduling algorithms under comparison.

4.3.4 Jacobi Iterative Algorithm

As previously described, experiments using the Jacobi algorithm were performed using two different matrix sizes (128×128 and 1024×1024, each with approximately 15% sparsity). The elements of the matrix were generated randomly using a function that maintained a correlated variation in the number of non-zero elements between any two successive rows of the matrix.

Figures 4.33–4.38 present the results obtained for each of the two matrix sizes. As the loop working set size used in our experiments for large values of parallel loop size (Figures 4.34 and 4.38) does not fit in two caches, there is hardly any difference in the execution times of the three affinity scheduling algorithms when the number of processors is less than three. As is evident from these figures, once the caches contain the relevant data, affinity scheduling reduces the need for any main memory access. Similar to the results obtained for both the triangular and rectangular ($k = 2$) workloads, further increasing the number of processors (for large parallel loop sizes) has little effect on the performance of the affinity scheduling algorithms.

From Figures 4.37 and 4.38, the *wrapped* algorithm achieves a maximum speedup of approximately 50% over the *static* algorithm. Corresponding figures for the *dy-*

namic algorithm are approximately 25%. As seen earlier in Sections 4.3.1 and 4.3.2, for small parallel loop sizes (as seen in Figures 4.33 and 4.37), the contention overhead manifests itself in the decrease in performance as the number of processors is increased. For larger values of the parallel loop (Figures 4.34 and 4.38), there is a marked improvement in the performance of the *wrapped* algorithm as the per-chunk overhead gets amortized over larger chunks.

Although the Jacobi algorithm differs significantly in form from the previously considered workloads (see Figures 4.1, 4.2 and 4.3), the performance of the two proposed algorithms is remarkably similar for all these workloads indicating the relative insensitivity of these algorithms to the particular form of the workload.

4.3.5 Convergence Rate of the Dynamic Algorithm

Experiments were also performed for different values of the outer sequential loop to determine the "convergence" rate of the dynamic algorithm (i.e., the minimum value of the outer sequential loop bound for which the initial partitions stabilize to yield roughly constant ranges). Figures 4.39–4.42 and 4.43–4.46, present the results obtained for the triangular workload and also for the rectangular ($k = 8$) workload, respectively. (Note that the results are presented only in terms of the number of migrations as the execution times for small sequential loop iteration counts are heavily influenced by the startup overhead.) As is evident from these figures, for small outer sequential loop bounds (< 4), there is no perceptible difference between the *dynamic* and *static* algorithms. This is because at these small outer loop bounds, the initial partition sizes used by *dynamic* have still not converged to reasonable values, and thus the readjustment of partitioned sizes have not yet brought substantive benefits. Larger values of the outer sequential loop afford significantly greater potential for these initial partition sizes to converge. This is evident from Figures 4.42 and 4.43, where the number of iterations migrated in the case of the *dynamic* algorithm is fairly constant.

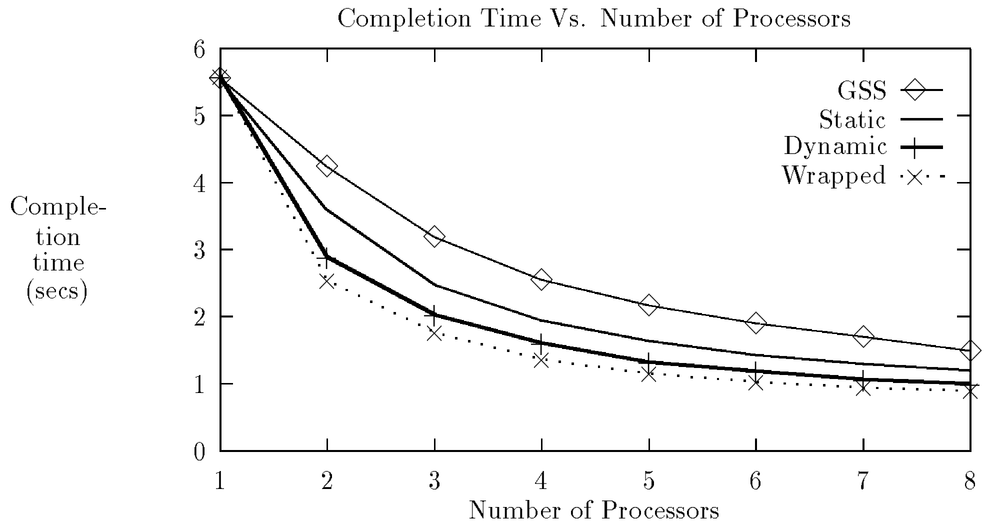

Figure 4.5: Triangular Workload ($N = 128$)

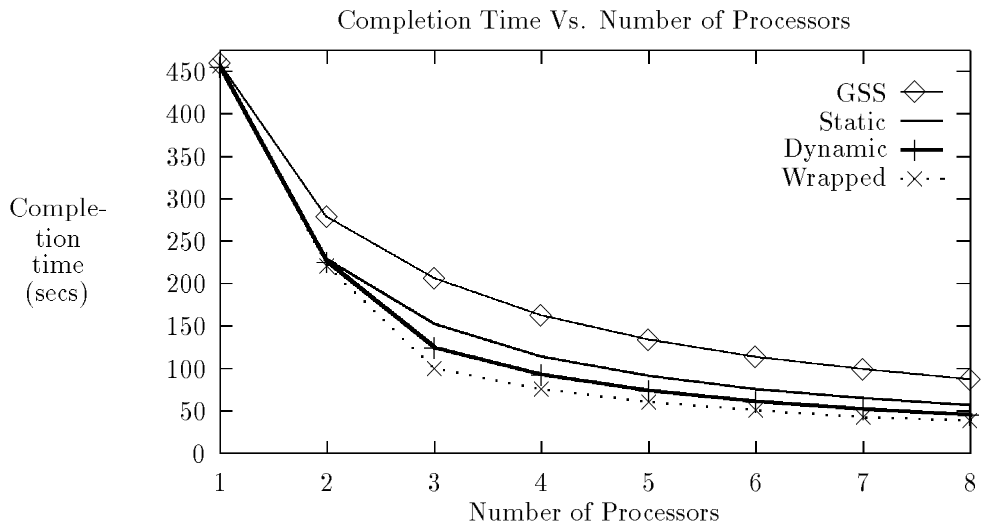

Figure 4.6: Triangular Workload ($N = 1024$)

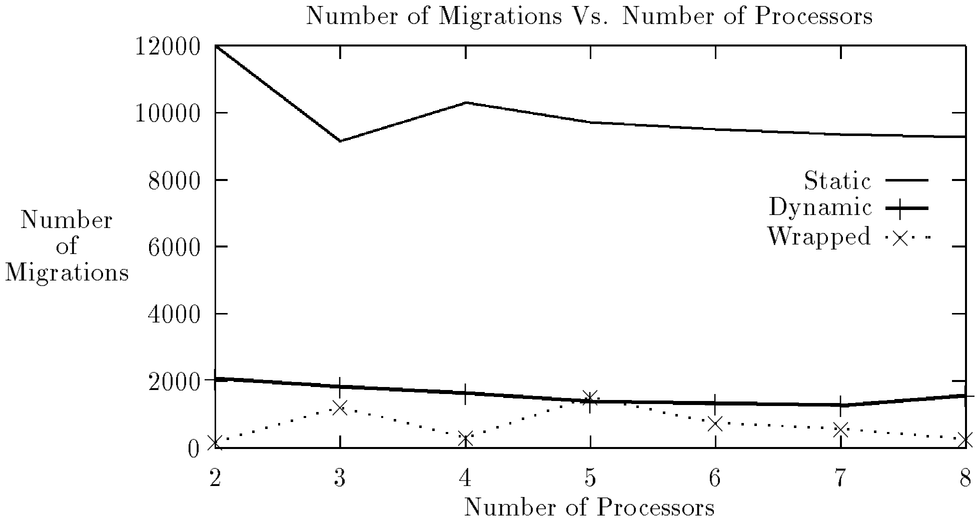

Figure 4.7: Triangular Workload ($N = 128$)

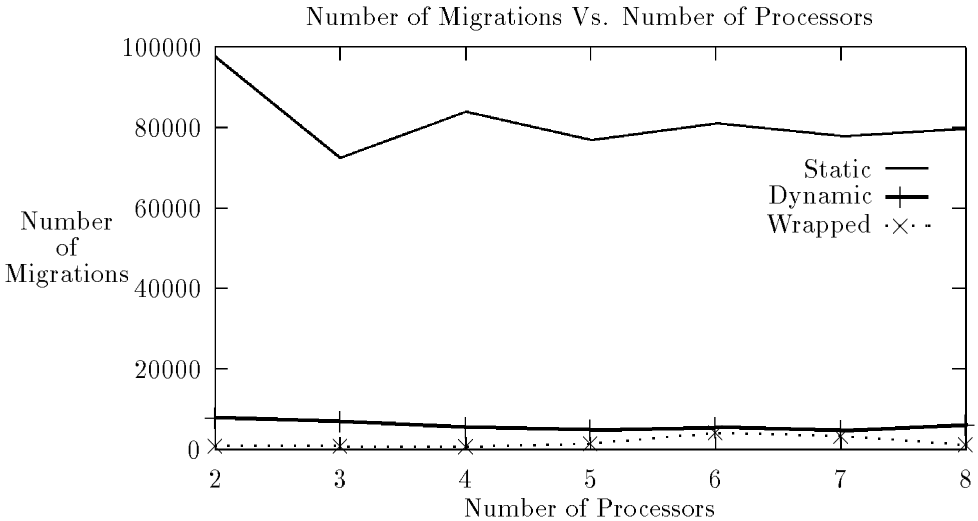

Figure 4.8: Triangular Workload ($N = 1024$)

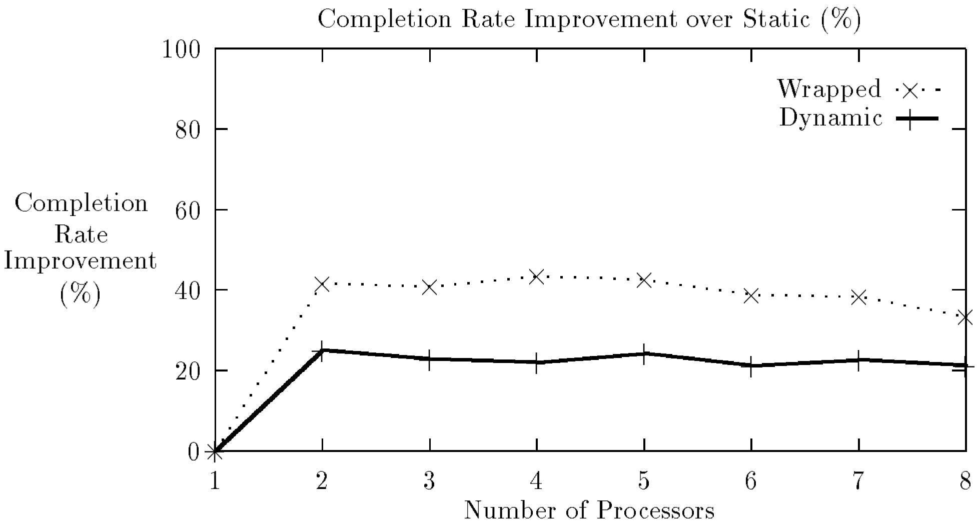

Figure 4.9: Triangular Workload ($N = 128$)

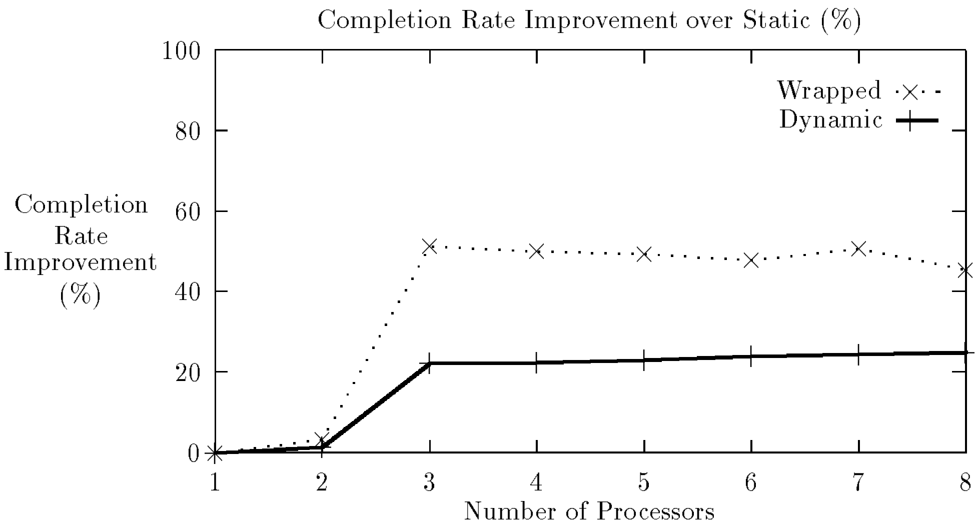

Figure 4.10: Triangular Workload ($N = 1024$)

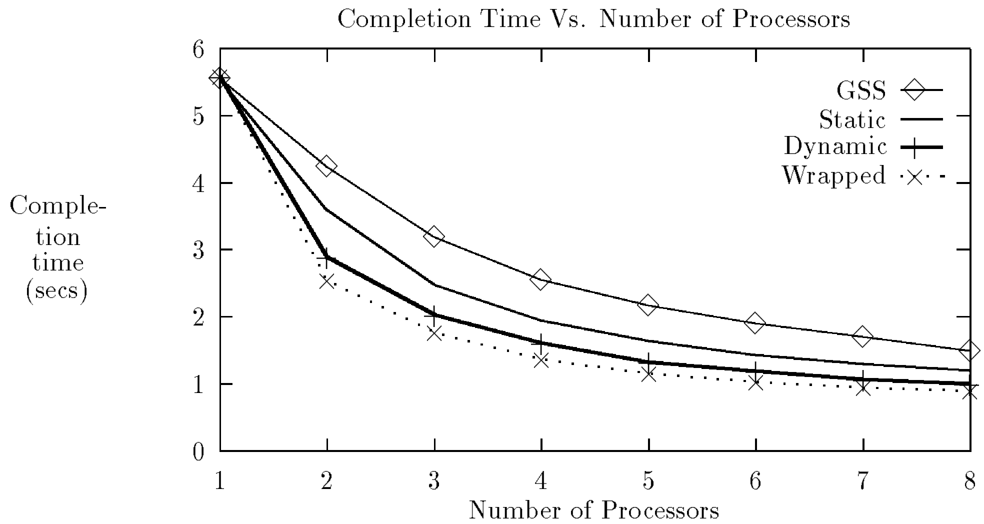

Figure 4.11: Rectangular Workload ($k = 2$, $N = 128$)

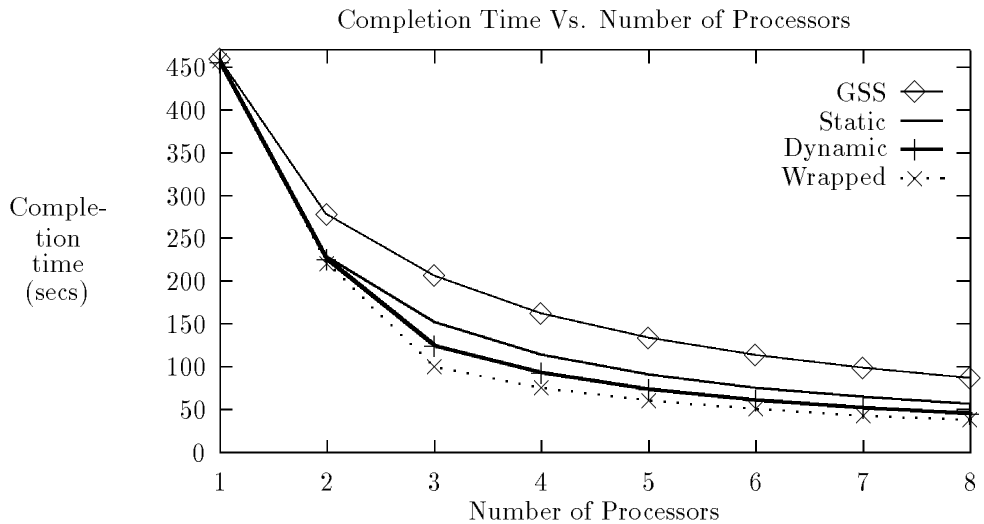

Figure 4.12: Rectangular Workload ($k = 2$, $N = 1024$)

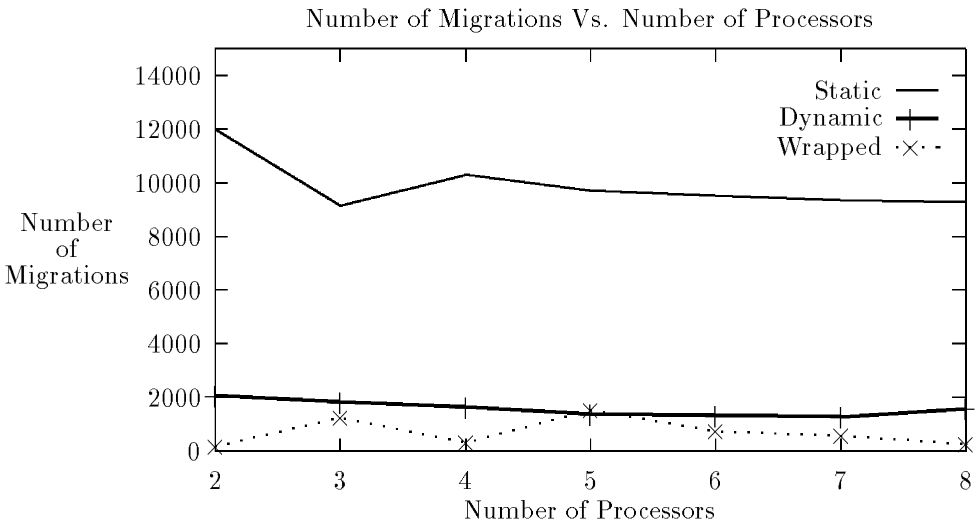

Figure 4.13: Rectangular Workload ($k = 2$, $N = 128$)

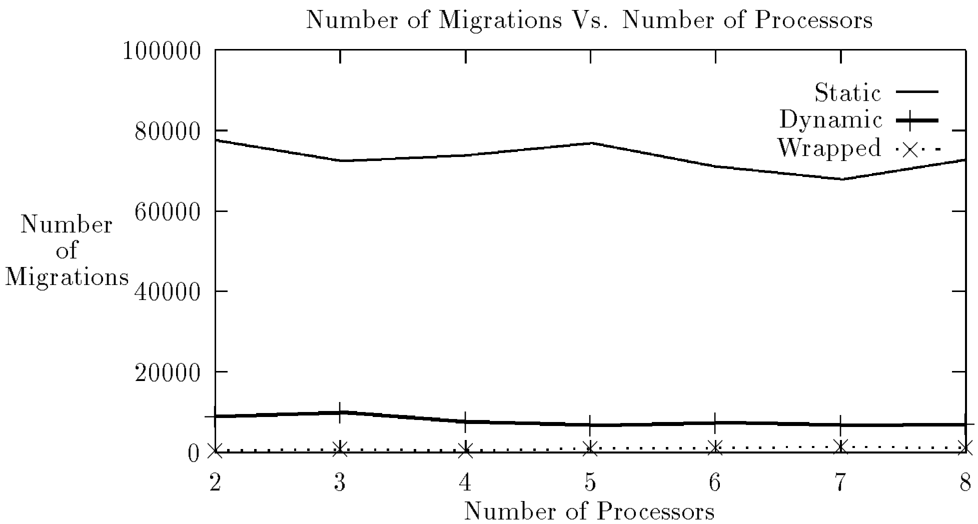

Figure 4.14: Rectangular Workload ($k = 2$, $N = 1024$)

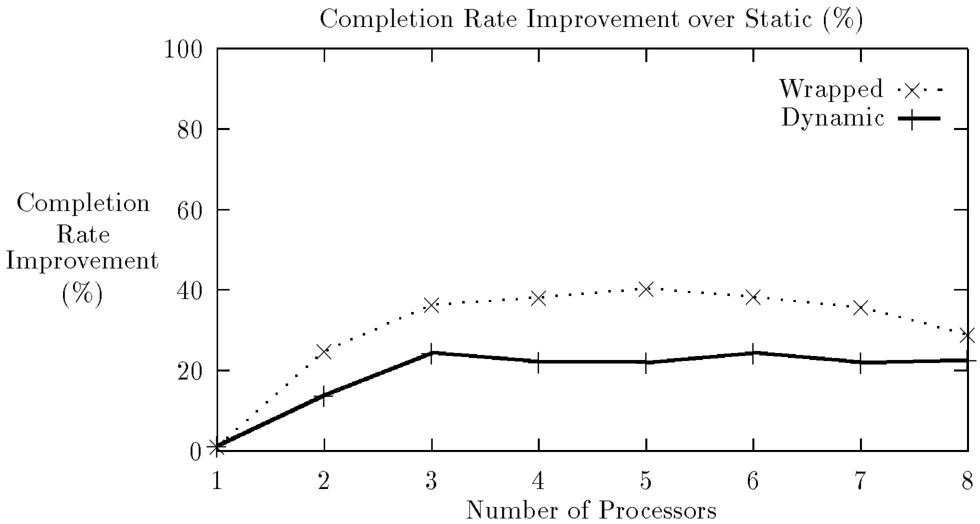

Figure 4.15: Rectangular Workload ($k = 2$, $N = 128$)

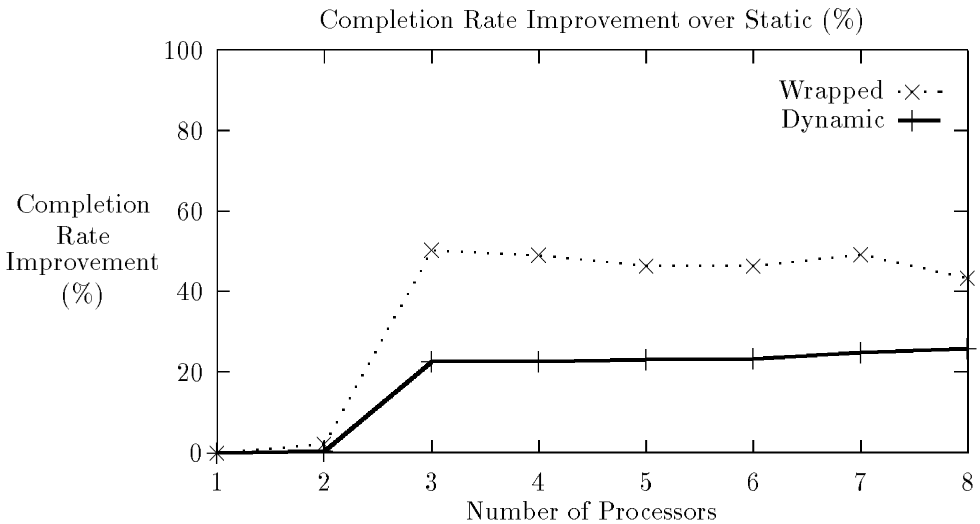

Figure 4.16: Rectangular Workload ($k = 2$, $N = 1024$)

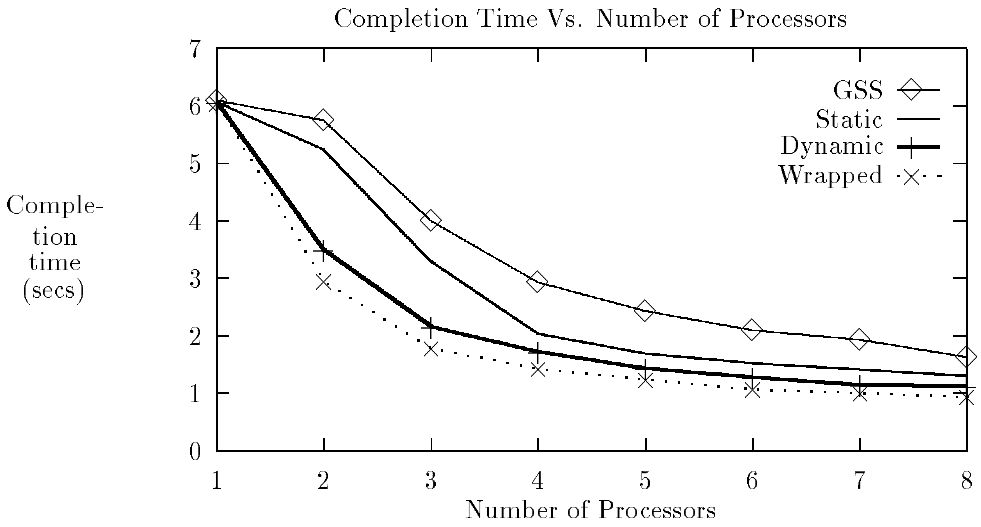

Figure 4.17: Rectangular Workload ($k = 4$, $N = 128$)

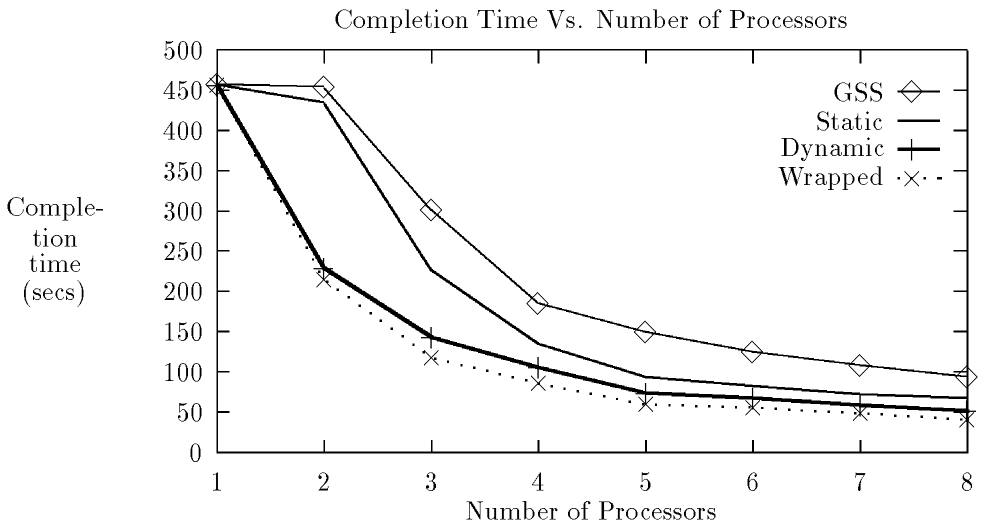

Figure 4.18: Rectangular Workload ($k = 4$, $N = 1024$)

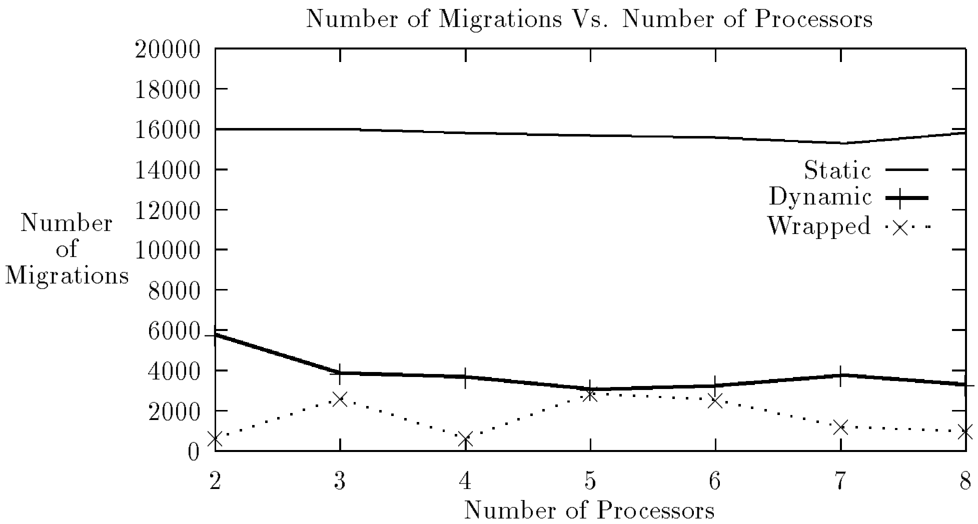

Figure 4.19: Rectangular Workload ($k = 4$, $N = 128$)

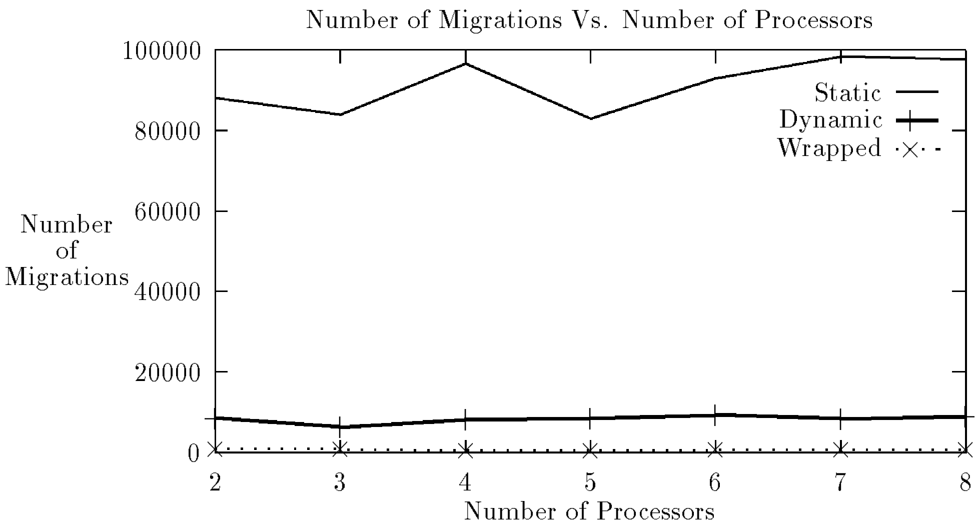

Figure 4.20: Rectangular Workload ($k = 4$, $N = 1024$)

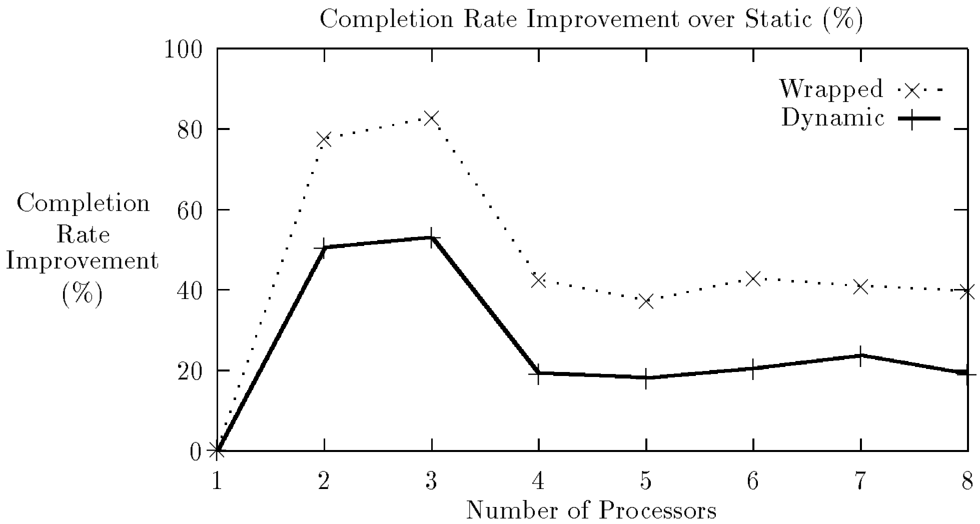

Figure 4.21: Rectangular Workload ($k = 4$, $N = 128$)

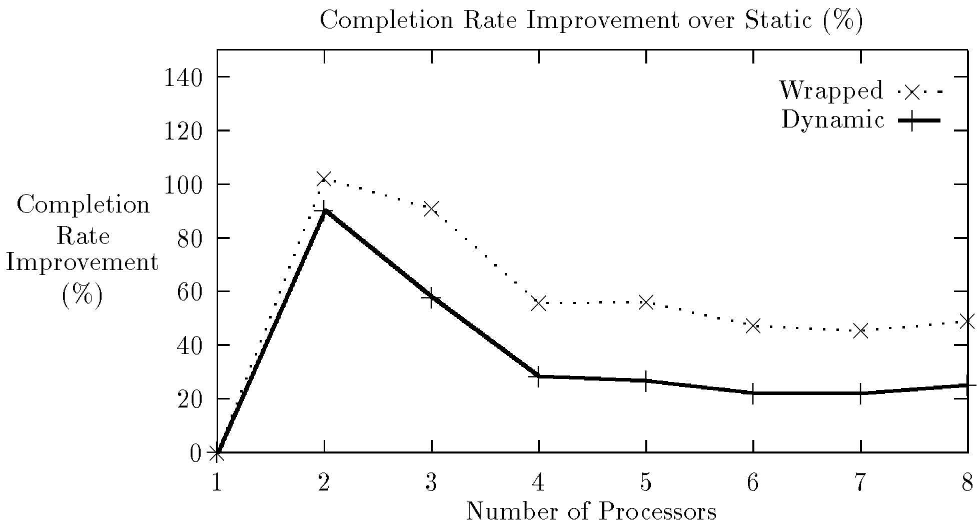

Figure 4.22: Rectangular Workload ($k = 4$, $N = 1024$)

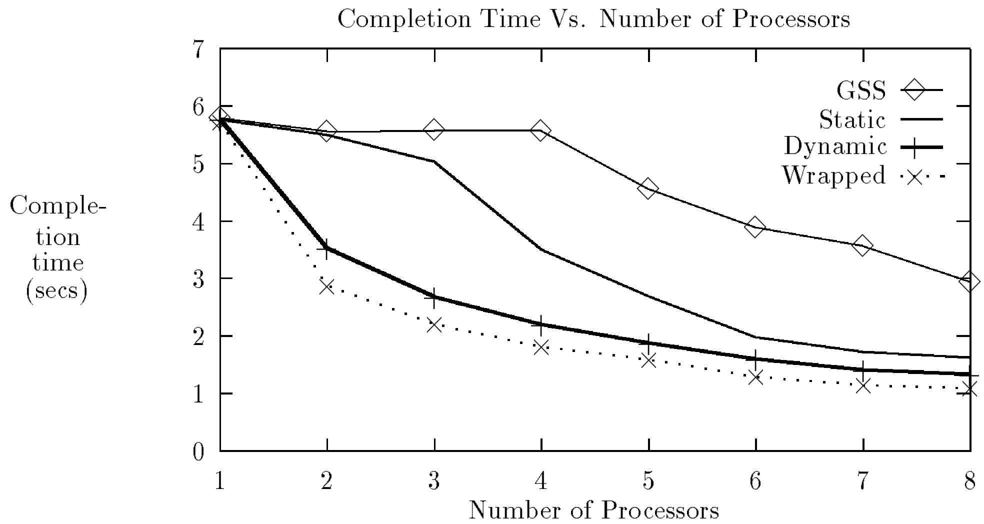

Figure 4.23: Rectangular Workload ($k = 8$, $N = 128$)

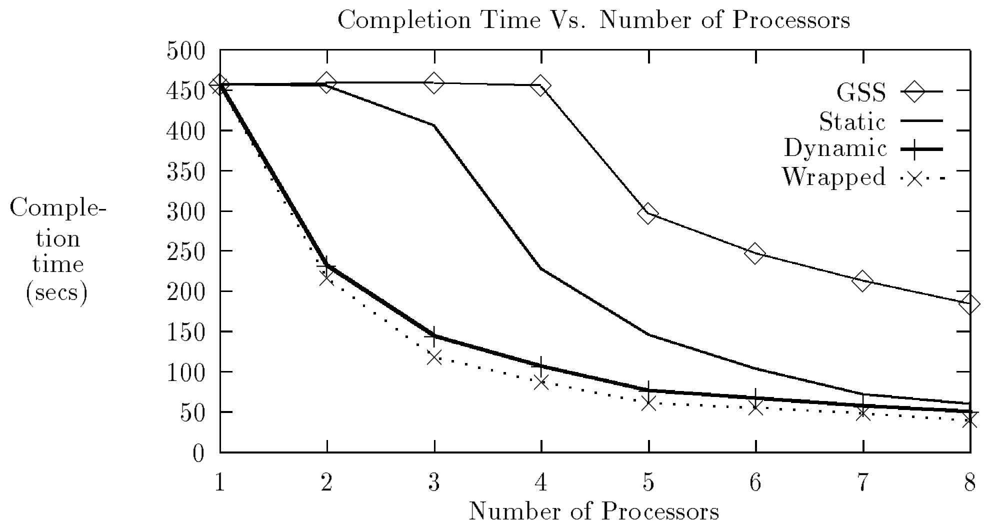

Figure 4.24: Rectangular Workload ($k = 8$, $N = 1024$)

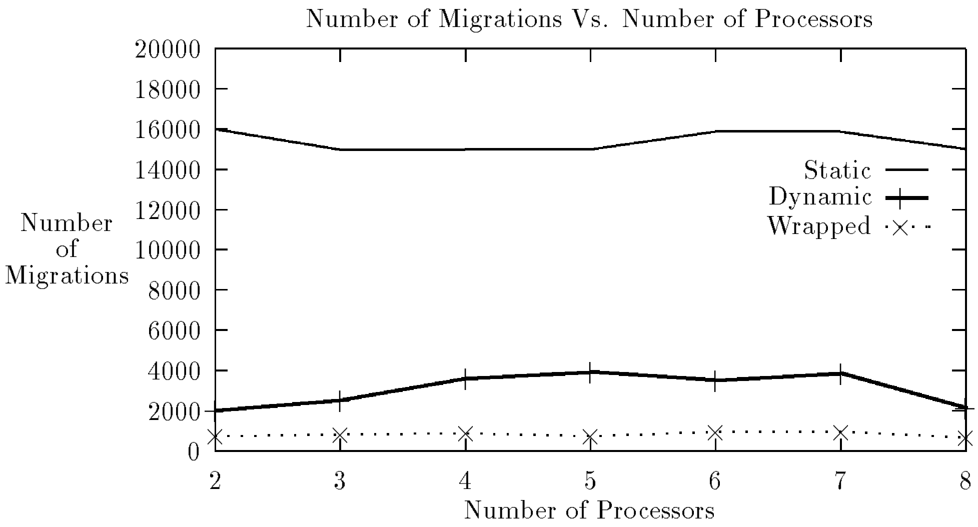

Figure 4.25: Rectangular Workload ($k = 8$, $N = 128$)

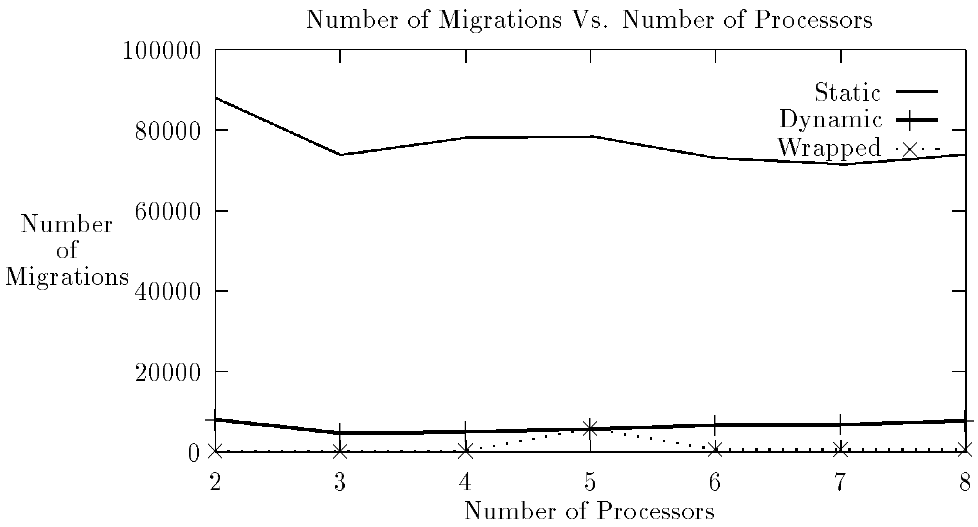

Figure 4.26: Rectangular Workload ($k = 8$, $N = 1024$)

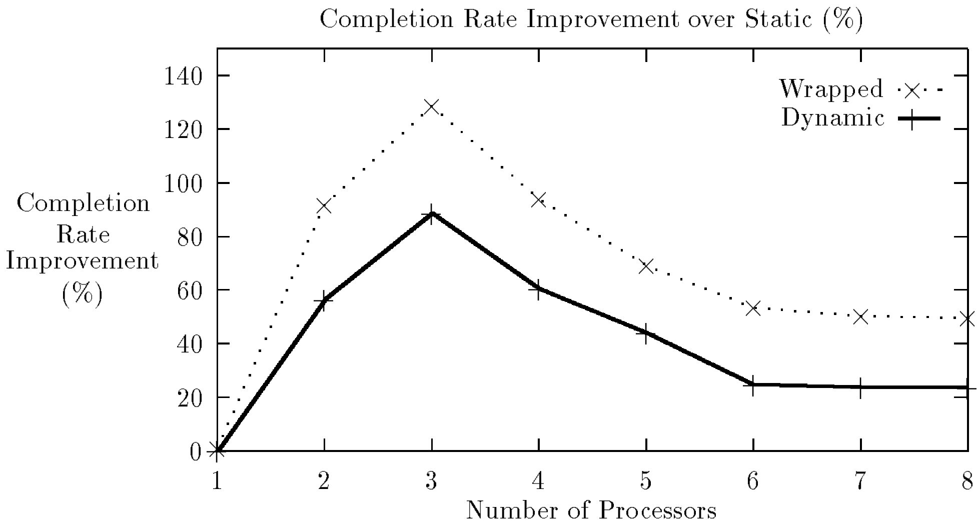

Figure 4.27: Rectangular Workload ($k = 8$, $N = 128$)

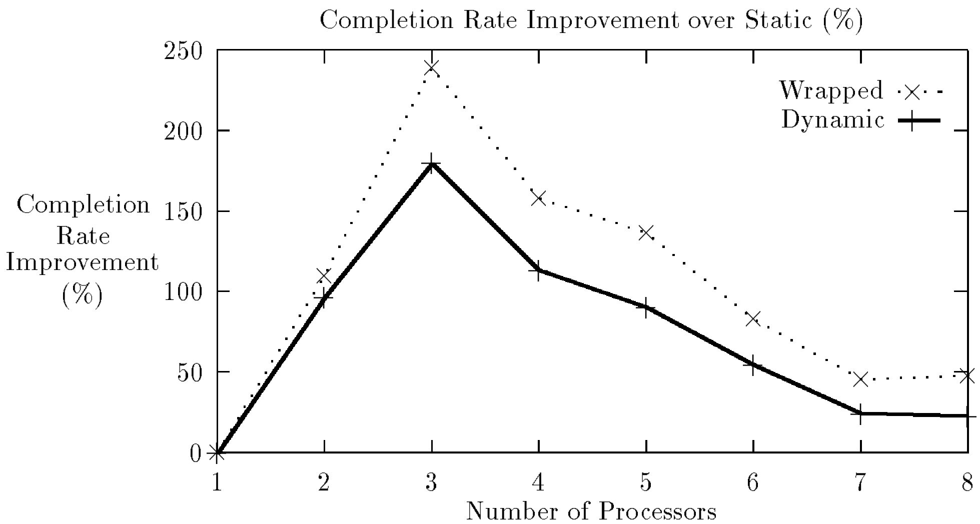

Figure 4.28: Rectangular Workload ($k = 8$, $N = 1024$)

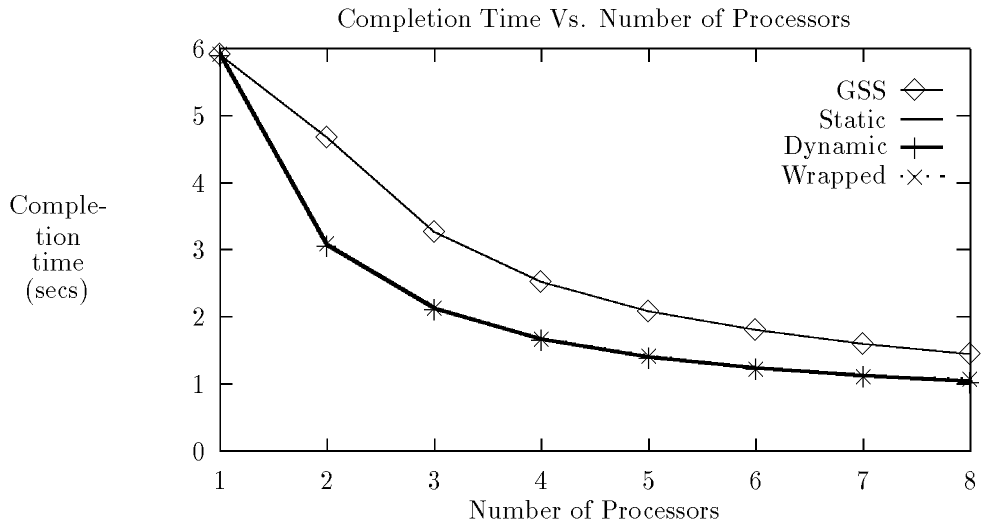

Figure 4.29: Balanced Workload ($N = 128$)

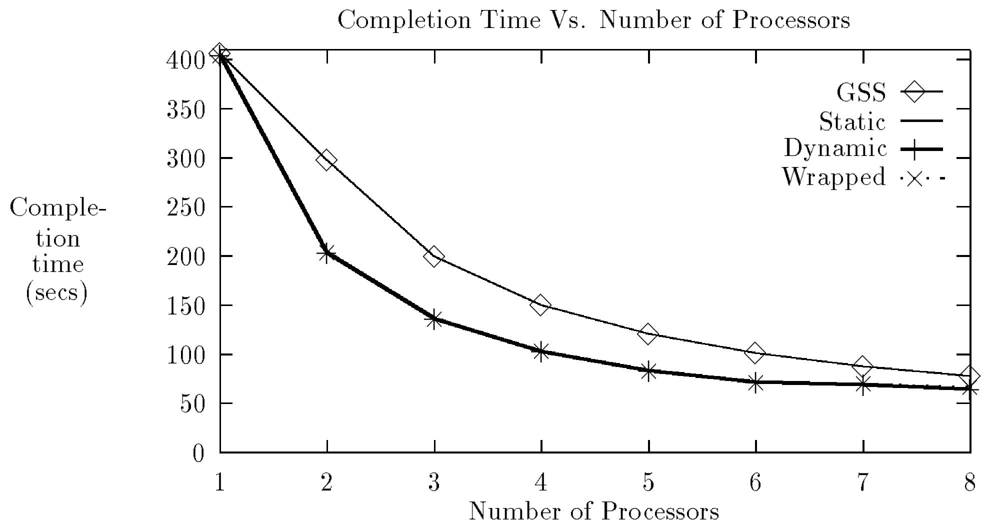

Figure 4.30: Balanced Workload ($N = 1024$)

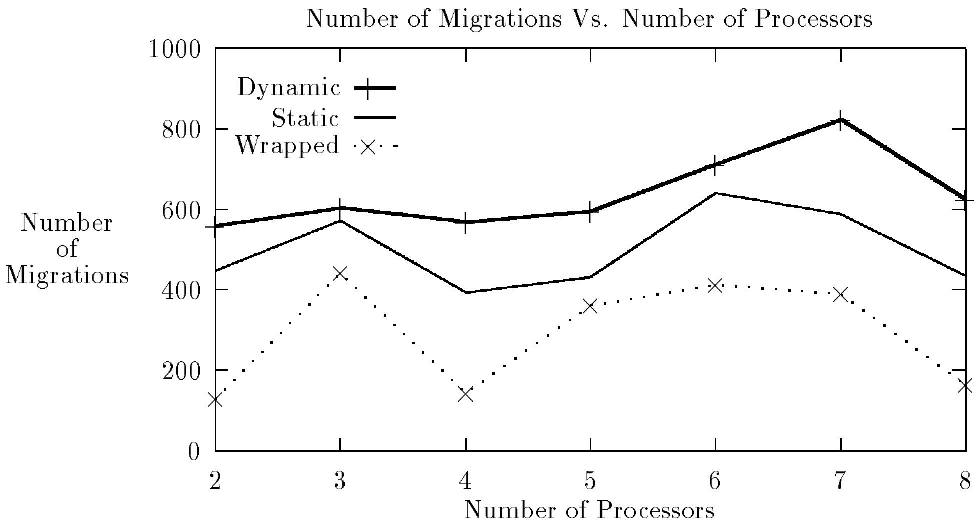

Figure 4.31: Balanced Workload ($N = 128$)

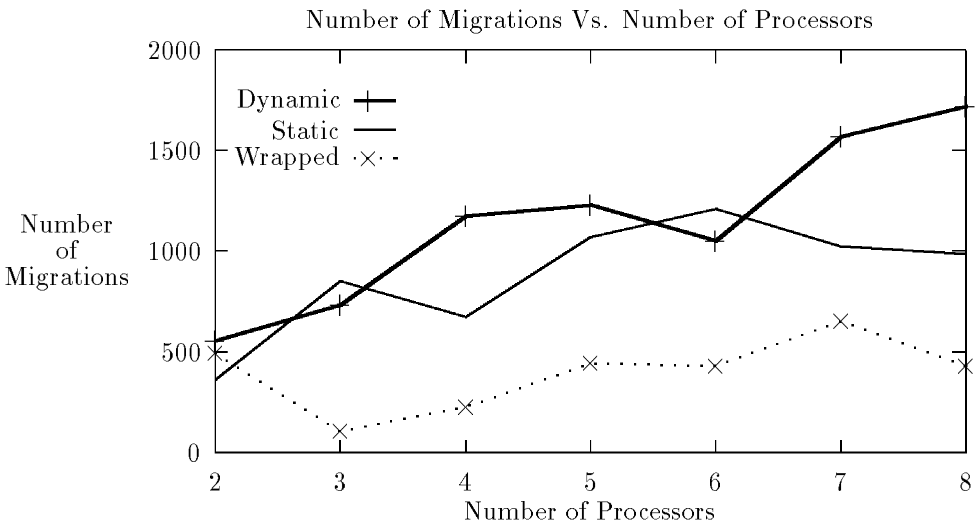

Figure 4.32: Balanced Workload ($N = 1024$)

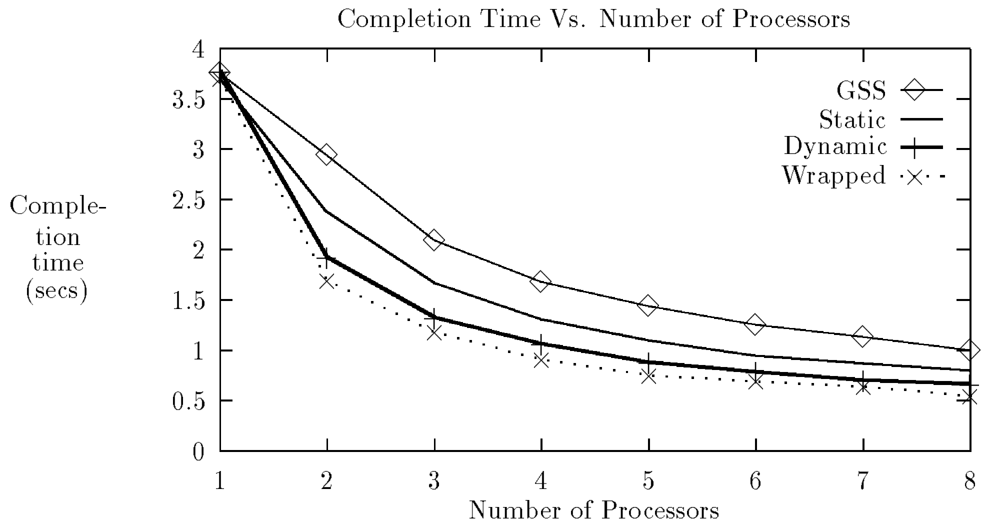

Figure 4.33: Jacobi Algorithm (matrix size = 128×128)

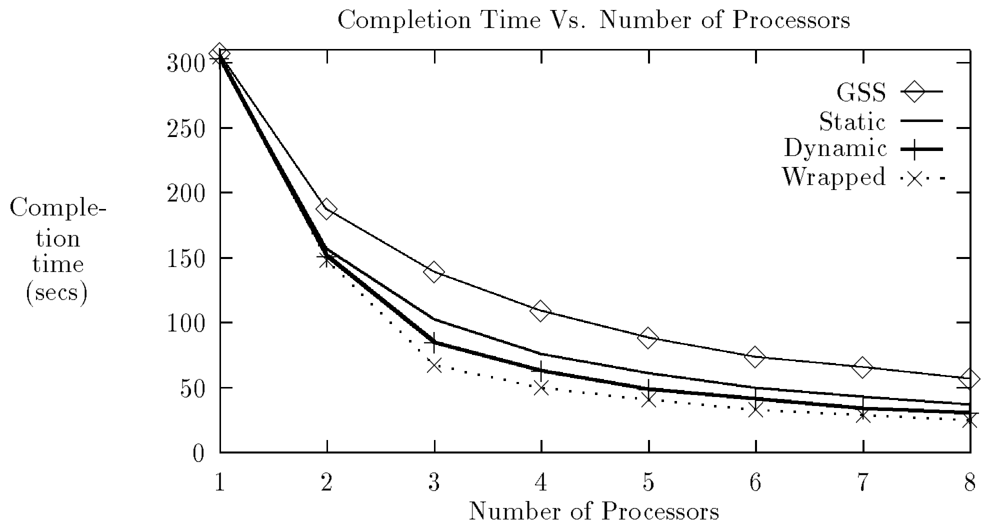

Figure 4.34: Jacobi Algorithm (matrix size = 1024×1024)

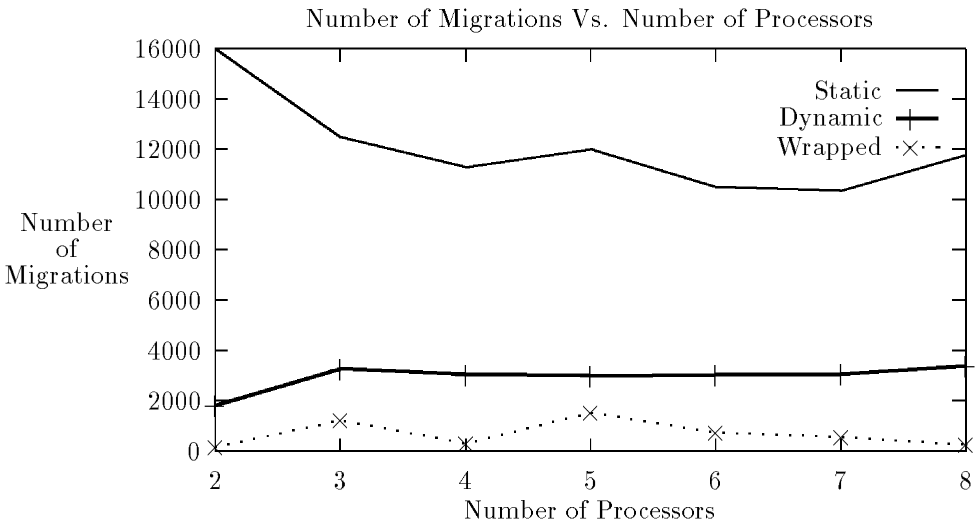

Figure 4.35: Jacobi Algorithm (matrix size = 128 $\times$ 128)

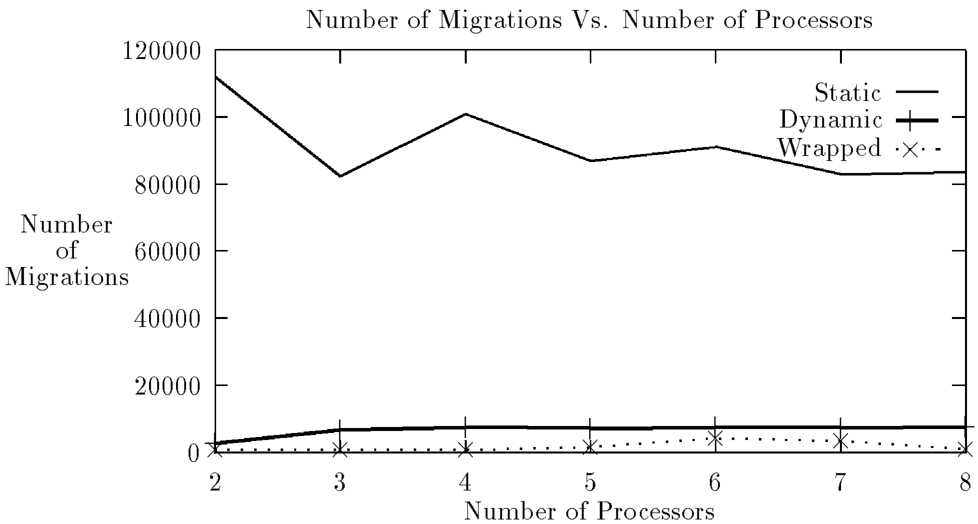

Figure 4.36: Jacobi Algorithm (matrix size = 1024 $\times$ 1024)

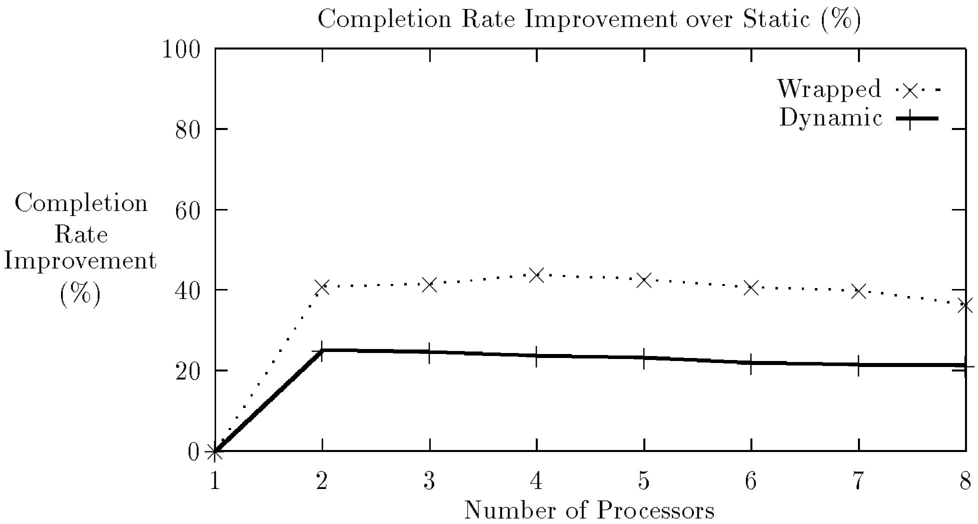

Figure 4.37: Jacobi Algorithm (matrix size = 128 $\times$ 128)

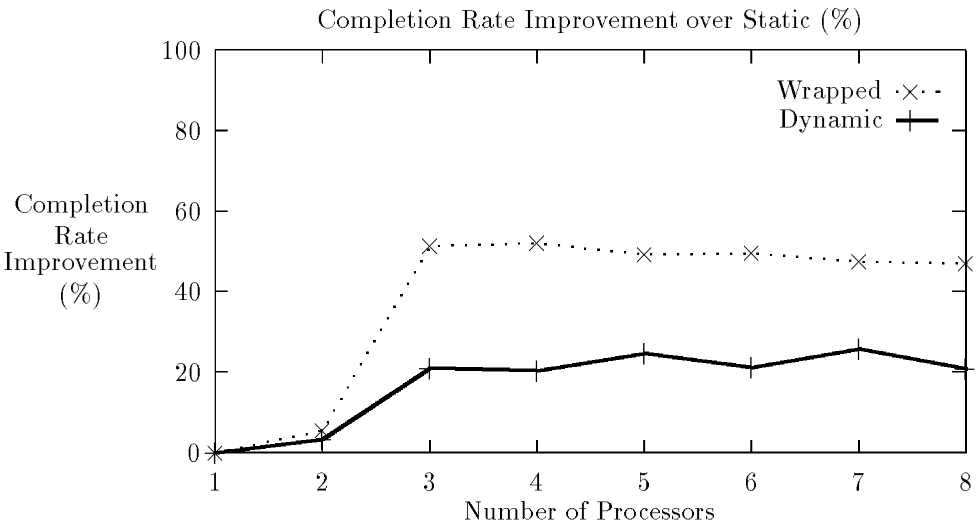

Figure 4.38: Jacobi Algorithm (matrix size = 1024 $\times$ 1024)

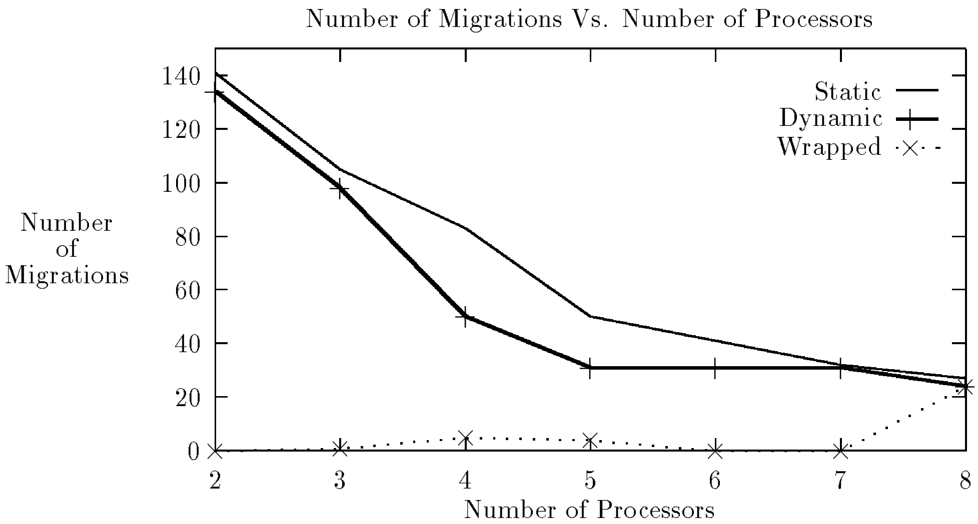

Figure 4.39: Triangular Workload ($N = 1024,\ K = 1$)

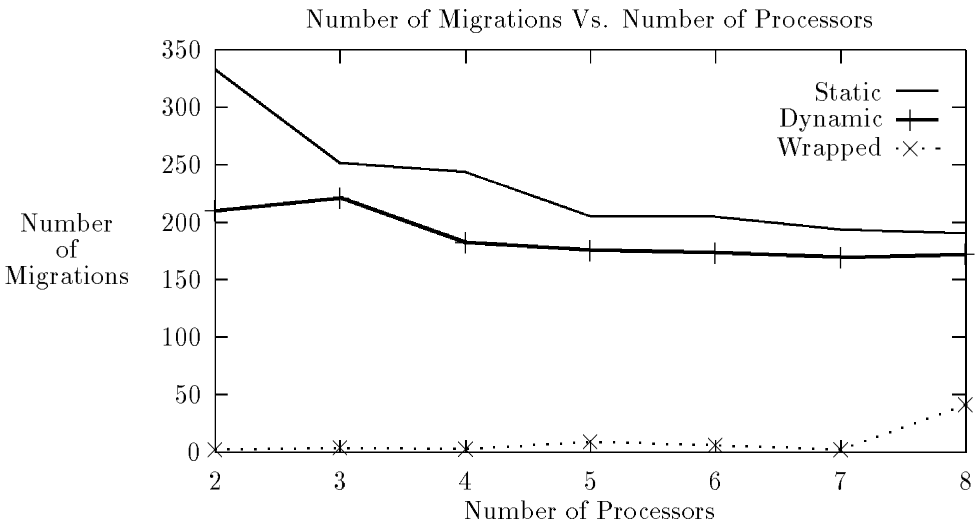

Figure 4.40: Triangular Workload ($N = 1024,\ K = 2$)

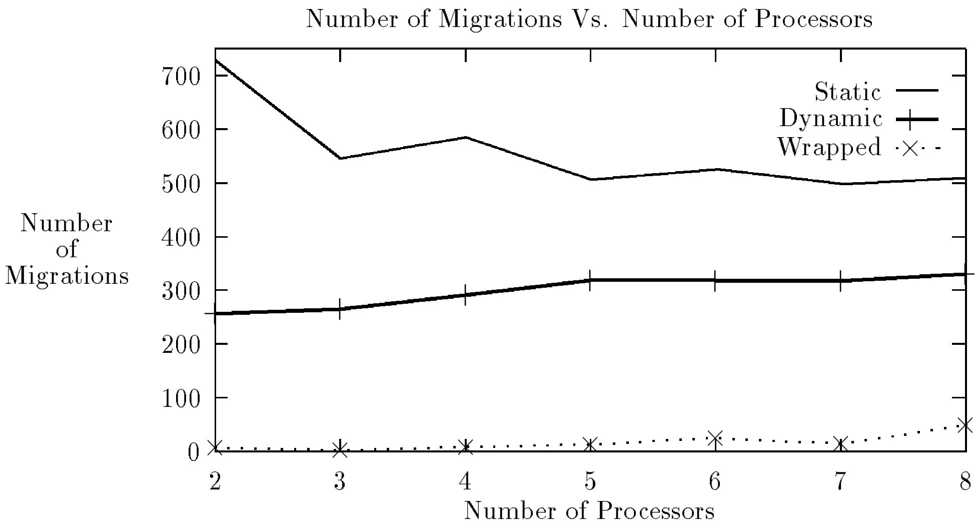

Figure 4.41: Triangular Workload ($N = 1024$, $K = 4$)

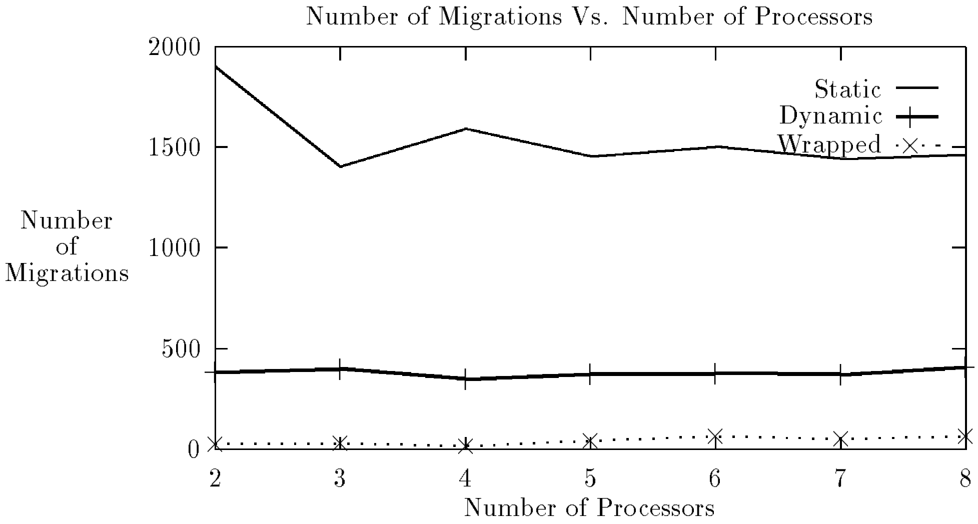

Figure 4.42: Triangular Workload ($N = 1024$, $K = 10$)

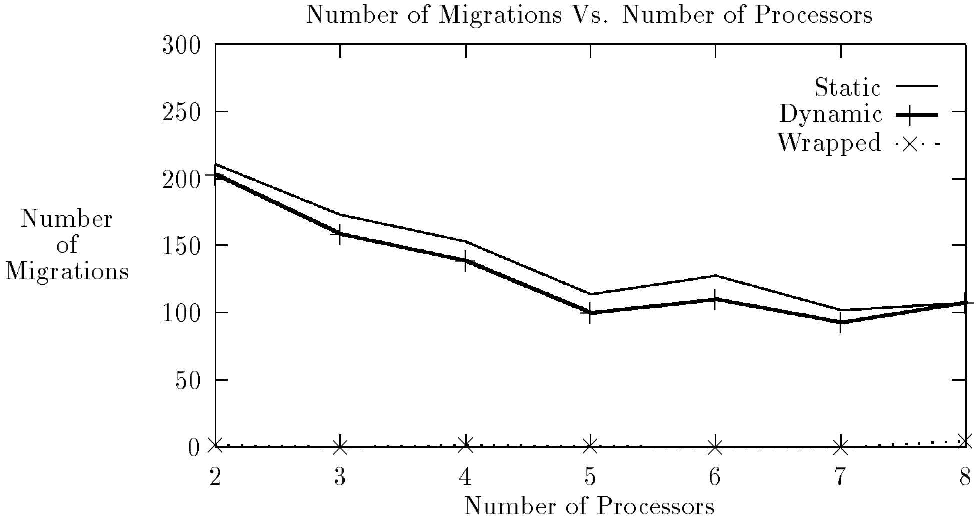

Figure 4.43: Rectangular Workload ($k = 8$, $N = 1024$, $K = 1$)

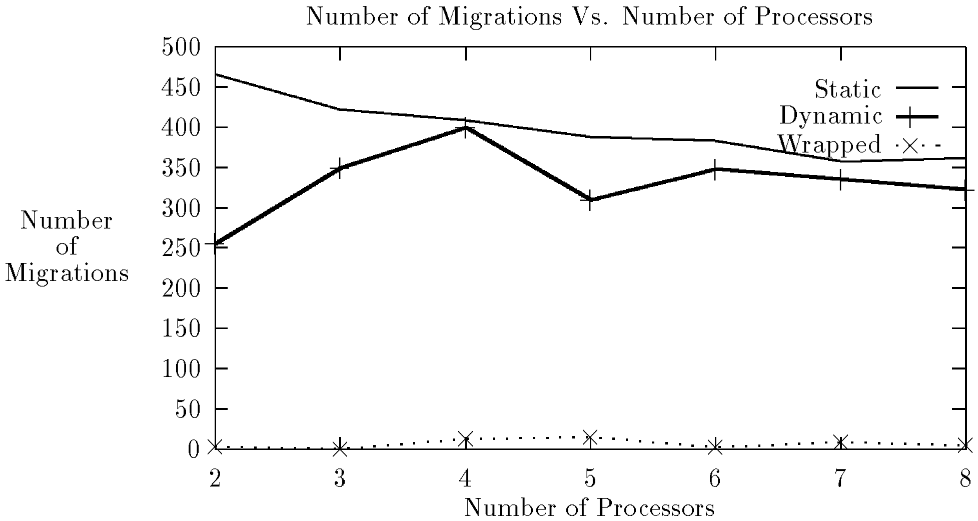

Figure 4.44: Rectangular Workload ($k = 8$, $N = 1024$, $K = 2$)

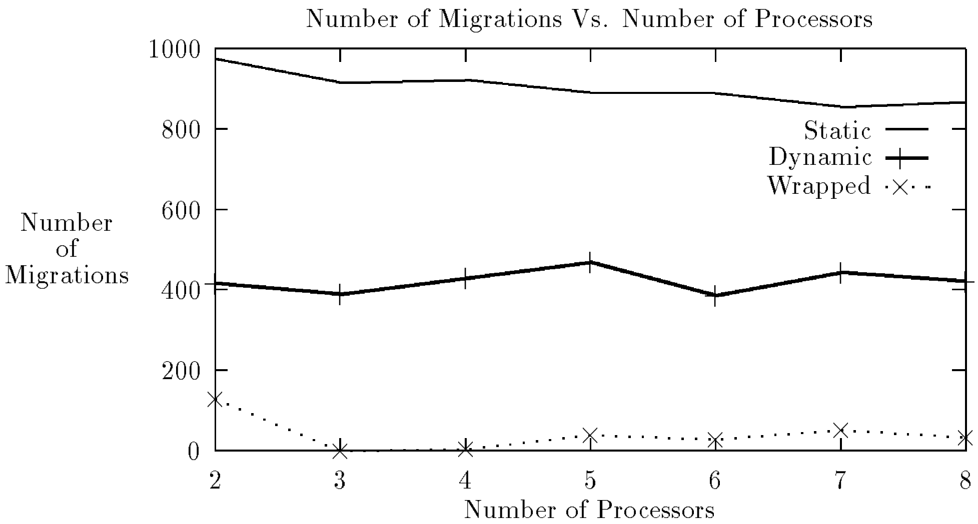

Figure 4.45: Rectangular Workload ($k = 8$, $N = 1024$, $K = 4$)

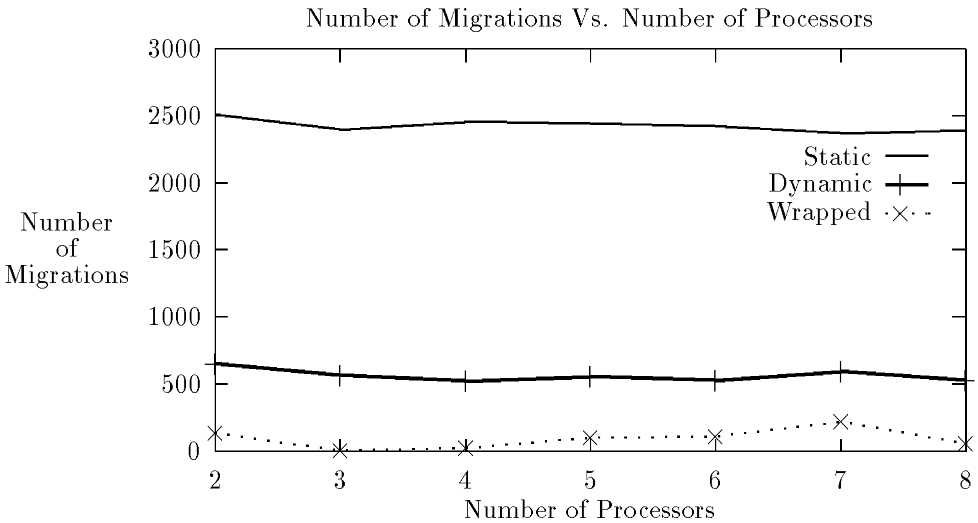

Figure 4.46: Rectangular Workload ($k = 8$, $N = 1024$, $K = 10$)

4.4 Research Conclusions

The major conclusions from this research are:

1. By reducing the number of iterations that need be migrated, and thus better respecting processor affinity, the two proposed algorithms perform significantly better than the *static* algorithm, even for severely unbalanced workloads. Further, the results indicate that there is no substantial overhead from either the *dynamic* or the *wrapped* allocation of iterations.

2. The working set size has a direct bearing on the performance of affinity scheduling algorithms. The best performance is usually obtained when the working sets of the application fits entirely in the cache (or local memories) of the multiprocessor.

3. For moderately unbalanced workloads, there is no noticeable difference in the performance of the four algorithms under consideration in the single processor case. In the multiprocessor case, by contrast, there is a marked improvement in the performance of the affinity scheduling algorithms when compared to that of the GSS (which uses a central work queue). This is because a single work queue requires the frequent movement of data among processors, since every processor must first load the data it needs into its local cache. The resulting communication overhead degrades performance.

4. Both the GSS and the *static* algorithms may show a marked degradation in performance for severely unbalanced workloads. In such severely unbalanced workloads, the first chunk of iterations may contain most of the work, leaving the remaining iterations very little work to balance the load. For such workloads, the dynamic readjustment of partition sizes and also the wrapped assignment of iterations provides for superior load balancing capability. The *dynamic* and *wrapped* algorithms thus seem very robust with respect to the type of workload.

5. The *dynamic* algorithm requires only a small number of outer sequential loop iterations to "converge" (i.e., the initial partitions stabilize to yield roughly constant ranges). For very small outer sequential loop bounds (< 4), convergence cannot be achieved, however, and there is no perceptible difference between the *dynamic* and *static* algorithms.

4.5 Summary

Using the synthetic application, different workloads, each varying in the structures and extent of load imbalance, were considered. The two main metrics used to evaluate the performance of the three affinity algorithms were: *application execution time* and *scalability*. Scalability was measured by counting the *number of iterations migrated* as a result of the proposed algorithms. When compared to the *static* algorithm, both the proposed algorithms achieved a significant reduction in the number of iterations that were migrated, thus signifying better scalability. For moderately unbalanced workloads, of the three algorithms under consideration, the *wrapped* algorithm provides the best performance – an improvement of approximately 50% (depending upon the workload) over the *static* algorithm when evaluated using the *application execution time* metric. Using the same metric, the *dynamic* algorithm was also better than that of the static algorithm by approximately 25% (again, depending upon the workload). However, for severely unbalanced workloads, both the GSS and the *static* algorithms show a marked degradation in performance for smaller number of processors. For such workloads, the performance of the *static* algorithm is comparable to that of the GSS algorithm (which clearly has the worst performance figures).

A perfectly balanced workload (iterations with constant execution times) was also used to assess the amount of overhead resulting from the *wrapped* and *dynamic* implementations. The results indicate that there is no appreciable overhead from either of the two implementations as all three affinity scheduling algorithms under consideration had identical execution times. The *dynamic* algorithm had a higher number of migrations due to the rather aggressive policy followed in determining the per-processor partition sizes, but this number was still quite small. Further, our results indicate that the dynamic reassignment of iterations needs a relatively small number of outer loop iterations to "converge". The Jacobi algorithm was also used to evaluate the performance of the proposed algorithms. The results obtained are consistent with those that were obtained using the synthetic workload.

As the results indicate, a major shortcoming of the *static* algorithm is its inability to handle severely unbalanced workloads. For such workloads, both the two proposed algorithms greatly outperformed the *static* algorithm.

Finally, it is important to note that the good performance of the *wrapped* algorithm does not imply that the *dynamic* algorithm need never be used. The wrapped assignment of iterations results in assigning consecutive iterations to distinct processors – thus violating spatial locality. When consecutive iterations access data that is in consecutive memory locations, a single cache miss may load data useful in multiple consecutive iterations. Due to the wrapped allocation of iterations, processors may not be able to take advantage of the data that is already present in the cache (as a result of a previous cache miss). Wrapping small intervals of consecutive iterations is a possible solution, but this will have to be traded off against the disadvantage of a more complicated mapping function.

Chapter 5

Conclusions

The research domain in this thesis is the scheduling of shared memory MIMD multiprocessors. In particular, the scheduling of units of work within a single application, when these units of work have "affinities" for particular processors (owing to the locations of the data they access) is considered. Affinity scheduling is studied in the context of a "one process per processor" program structure, in which the units of work being scheduled are the iterations of program loops that have been parallelized. This research specifically focuses on the case in which the loops being executed are unbalanced with respect to the amounts of computation represented by each iteration, a situation in which simple forms of affinity scheduling may become inappropriate.

Affinity scheduling assumes added significance in light of the fact that communication is a significant and an increasing source of overhead in present day shared memory multiprocessors. Until very recently, communication overhead was not a major issue in shared memory multiprocessors, mainly because processors were so slow that communication was a negligible percentage of the total completion time of the parallel application. Recent advances in VLSI and RISC technology have produced significant improvements in processor speeds, while there has only been a moderate increase in memory and interconnection network speeds. An important ramification of this growing disparity between processor speed and communication bandwidth is the potential inability of parallel programs to exploit any increase in processor speed as this increase in processing speed is not accompanied by a corresponding increase in the communication bandwidth.

Another motivation for affinity scheduling is the growing interest in the development of large scale multiprocessor systems with a single address space and coherent caches. Section 2.1.2 described these architectures in detail. Due to the huge sizes of the cached main memories (32 MB in the KSR1), typically most of the working set of a unit of work may be held in the memory of the particular processor on which it is run. As a result, the implications of a larger cache reload transient time are likely to be more pronounced in these machines. Thus, affinity scheduling algorithms are expected to be particularly important on these machines.

5.1 Thesis Contributions

This thesis has proposed two new algorithms (*dynamic* and *wrapped*) for the context of unbalanced workloads in which the parallel loop being executed is severely unbalanced with respect to the amounts of computation represented by each iteration, a situation in which simple forms of affinity scheduling are shown to be inadequate. An experimental study of the two proposed algorithms finds these algorithms to achieve both the objectives of maintaining affinity, and load balance, even for severely unbalanced workloads. Both algorithms achieve a significant improvement in performance when compared to the principal existing algorithm.

The static allocation of partition sizes to the processors in the *static* algorithm results in a significant cache-reload transient overhead that partially masks any affinity related gains. In the *dynamic* algorithm, by contrast, by providing for the dynamic readjustment of partition sizes, the initial allocation of partition sizes stabilizes to yield fairly constant values for each processor. The results in this thesis indicate that the *dynamic* algorithm requires only a small number of outer sequential loop iterations for the partition sizes to stabilize. For very small outer sequential loop bounds (< 4), the initial partition sizes used by the *dynamic* algorithm have still not converged to reasonable values, and thus the readjustment of partition sizes have not yet brought substantive benefits. Larger values of the outer sequential loop afford significantly greater potential for these initial partition sizes to converge.

The *wrapped* algorithm allocates chunks of iterations that are at a distance P (the number of processors in the system) from each other. This results in maintaining a good load balance by avoiding the assignment of all the time-consuming iterations to a single processor or chunk. Further, the two proposed algorithms significantly reduce the number of iterations that need to be migrated, thus better respecting processor affinity (and hence reducing the cache-reload transient overhead). The results from the experimental study suggest that the additional overhead due to the *dynamic* and *wrapped* assignment of iterations to processors is negligible. As a result, the performance of the two proposed algorithms is consistently better than that of the *static* approach. Finally, as results from using the various workloads indicate, the proposed algorithms are also more robust in the face of severely unbalanced workloads.

The results from this thesis also illustrate the fact that affinity scheduling performs the best when the working sets of the parallel applications fit in the caches (or local memories) of the multiprocessor. This has important ramifications for affinity scheduling algorithms on the new large scale multiprocessor systems like the KSR1. Although both caches and local memories used to be rather small, and could not always accommodate all the input for large scientific problems, this is no longer the case. Local memories have become larger every year, quadrupling their size every three years [28]. For example, the SGI Iris 4D/380S has 256 KB caches, while the KSR1 has 32 MB cached main memories. The huge cache sizes of these machines affords a greater potential for the reuse of cached data and hence an ideal platform to implement affinity scheduling.

5.2 Directions for Future Research

There are several extensions to this work that are possible. New affinity scheduling algorithms may be needed to address more general program structures, where multiple parallel loops may have different data access patterns within a single enclosing sequential loop. Also, the workloads considered in this thesis exhibited (for the most part) loops in which the parallel loop iteration times do not change from one iteration of the outer sequential loop to the next. A different approach may be required to handle instances where there is a variability in the execution times from one iteration of the (outer) sequential loop to the next.

Secondly, the work in this thesis concentrates primarily on the scheduling of independent parallel loops ("do-all" loops). Again, newer affinity scheduling algorithms may be needed to handle the scheduling of parallel loops with dependencies ("do-across" loops).

Existing work on affinity scheduling does not fully address the question of under what circumstances the migration of active units of work is beneficial. For example, in the approach proposed by Markatos and LeBlanc [43], if load imbalance occurs, the algorithm migrates iterations from loaded processors to idle ones by having an idle processor probe the work queues of all the other processors and remove from the queue with the most iterations. It is not clear if this approach is always appropriate, as the following example illustrates.

Consider the situation where a processor has nearly completed the execution of its current set of iterations, and some work still remains in its work queue. The idle processors in the system are unaware of the fact that this processor is close to completion and may remove work from the work queue of this processor. The resulting cache-reload transient causes an additional overhead. Under the given circumstances, it may have been more advantageous from a performance point of view to have executed the migrated chunk of work on the original processor itself – thus avoiding the overhead that migration would cause.

One possible solution is to record every time a processor migrates work from another processor. This information could then be used to decide the suitability of migration in any subsequent executions of the loop.

Consider the example shown in Figure 5.1. In this example, processor P_1 migrates work from processor P_2 and later (in the same loop execution), processor P_2 migrates work from processor P_j. Now, in the next execution of the loop, if P_1 again has an

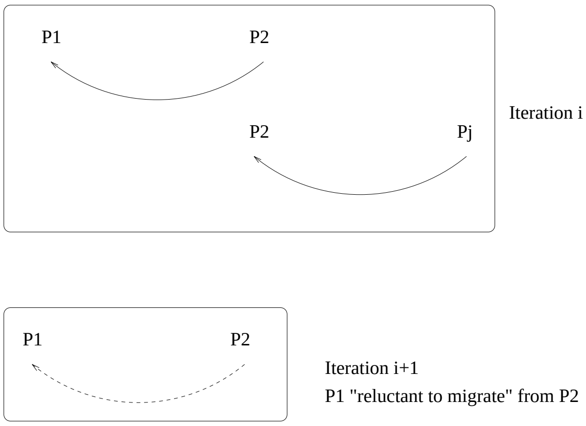

Figure 5.1: The "reluctant to migrate" policy

opportunity to migrate work from P_2, (i.e., P_1 becomes idle and P_2 has the longest queue of unfinished work) then it will be reluctant to migrate work – owing to the fact that P_2 (in the previous loop execution) finished its current allocation of work and ended up migrating work from P_j. Processor P_1 instead migrates $\frac{1}{r}$ of what it would normally migrate from P_2, where r is a policy parameter (e.g., we may choose $r = 2$).

This "reluctant to migrate" policy could be implemented in all three affinity scheduling algorithms described previously in Section 3.3, and the resulting performance gains (if any) characterized. By using this "reluctant to migrate" approach, it might be possible to quantify the circumstances under which the migration of active units of work is advantageous.

Finally, while the scheduling of shared memory multiprocessors has been an active area of research in recent years, there has been relatively little research on scheduling policies for message passing multicomputers, in comparison. Extending the results from this thesis to these architectures is another approach that may be worthwhile to explore.

References

[1] T. L. Adam, K. M. Chandy, J. R. Dickson, "A Comparison of List Schedules for Parallel Processing Systems", Communications of the ACM, Vol. 17, No. 12 (December 1974), pp. 685-690.

[2] A. Agarwal, R. Simoni, M. Horowitz, J. Hennessy, "An Evaluation of Directory Schemes for Cache Coherence", Proceedings of the 15th International Symposium on Computer Architecture, June 1988, pp. 280-289.

[3] A. Agarwal, B. H. Lim, D. Kranz, J. Kubiatowicz, "APRIL: A Processor Architecture for Multiprocessing", Proceedings of the 17th Annual International Symposium on Computer Architecture, May 1990, pp. 104-114.

[4] Anon.[1] et al., "A Measure of Transaction Processing Power", Datamation, Vol. 31, No. 7 (April 1985), pp. 112-118.

[5] T. E. Anderson, E. D. Lazowska, H. M. Levy, "The Performance Implications of Thread Management Alternatives for Shared Memory Multiprocessors", IEEE Transactions on Computers, Vol. 38, No. 12 (December 1989), pp. 1631-1644.

[6] T. E. Anderson, "The Performance of Spin Lock Alternatives for Shared Memory Multiprocessors", IEEE Transactions on Parallel and Distributed Systems, Vol. 1, No. 1 (January 1990), pp. 6-16.

[7] T. E. Anderson, B. N. Bershad, E. D. Lazowska, H. M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism",

[1]This article was put together by 24 computer professionals. Eight are academics, two are end-users, and 14 work for various vendors.

ACM Transactions on Computer Systems, Vol. 10, No. 1 (February 1992), pp. 53-79.

[8] J. Archibald, Jean-Loup Baer, "An Economical Solution to the Cache Coherence Problem", Proceedings of the 12th International Symposium on Computer Architecture, June 1985, pp. 355-362.

[9] B. N. Bershad, E. D. Lazowska, H. M. Levy, "PRESTO: A System for Object-Oriented Parallel Programming", Software – Practice and Experience, Vol. 18, No. 8 (August 1988), pp. 713-732.

[10] D. L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System", IEEE Computer, Vol. 23, No. 5 (May 1990), pp. 35-43.

[11] W. J. Bolosky, M. L. Scott, R. P. Fitzgerald, R. J. Fowler, A. L. Cox, "NUMA Policies and their Relation to Memory Architecture", Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, April 1991, pp. 212-221.

[12] R. Chandra, A. Gupta, J. L. Hennessy, "Integrating Concurrency and Data Abstraction in a Parallel Programming Language", Technical Report CSL-TR-92-511, Computer Systems Laboratory, Stanford University, February 1992.

[13] H. Cheong, A. V. Veidenbaum, "A Cache Coherence Scheme with Fast Selective Invalidation", Proceedings of the 15th International Symposium on Computer Architecture, June 1988, pp. 299-307.

[14] E. G. Coffman, Jr., R. L. Graham, "Optimal Scheduling for Two Processor Systems", Acta Informatica, Vol. 1 (1972), pp. 200-213.

[15] R. W. Conway, W. L. Maxell, L. W. Miller, *Theory of Scheduling*, Addison-Wesley, 1967.

[16] A. L. Cox, R. J. Fowler, "The Implementation of a Coherent Memory Abstraction on a NUMA Multiprocessor: Experiences with PLATINUM", Proceedings of the 12th Symposium on Operating Systems Principles, December 1989, pp. 32-44.

[17] D. H. Craft, "Resource Management in a Decentralized System", Proceedings of the 9th ACM Symposium on Operating Systems Principles, October 1983, pp. 11-19.

[18] H. M. Deitel, *An Introduction to Operating Systems*, Addison-Wesley, 1984.

[19] M. Devarakonda, A. Mukherjee, "Issues in the Implementation of Cache-Affinity Scheduling", Proceedings of the Winter 1992 USENIX Conference, January 1992, pp. 345-357.

[20] T. W. Doeppner, Jr., "Threads: A System for the Support of Concurrent Programming", Technical Report CS-87-11, Department of Computer Science, Brown University, 1987.

[21] D. L. Eager, J. Zahorjan, "Chores: Enhanced Run-Time Support for Shared Memory Parallel Computing", ACM Transactions on Computer Systems, Vol. 11, No. 1 (February 1993), pp. 1-32.

[22] D. L. Eager, J. Zahorjan, "Adaptive Guided Self-Scheduling", Technical Report 92-01-01, Department of Computer Science and Engineering, University of Washington, January 1992.

[23] M. Flynn, "Some Computer Organizations and Their Effectiveness", IEEE Transactions on Computers, Vol. 21, No. 9 (September 1972), pp. 948-960.

[24] R. J. Fowler, L. I. Kontothanassis, "Improving Processor and Cache Locality in Fine-Grain Parallel Computations using Object-Affinity Scheduling and Continuation Passing", Technical Report 411, Computer Science Department, University of Rochester, June 1992.

[25] J. R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic", Proceedings of the 10th Annual Symposium on Computer Architecture, June 1983, pp. 124-131.

[26] A. Gupta, A. Tucker, S. Urushibara, "The Impact of Operating System Scheduling Policies and Synchronization Methods on the Performance of Parallel Applications", Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, May 1991, pp. 120-132.

[27] E. Hagersten, A. Landin, S. Haridi, "DDM - A Cache Only Memory Architecture", IEEE Computer, Vol. 25, No. 9 (September 1992), pp. 44-54.

[28] J. L. Hennessy, D. A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann, 1990.

[29] E. Horowitz, S. Sahni, "Exact and Approximate Algorithms for Scheduling Nonidentical Processors", Journal of the ACM, Vol. 23, No. 2 (April 1976), pp. 317-327.

[30] S. F. Hummel, E. Schonberg, L. E. Flynn, "Factoring: A Method for Scheduling Parallel Loops", Communications of the ACM, Vol. 35, No. 8 (August 1992), pp. 90-101.

[31] N. P. Jouppi, D. W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines", Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems, April 1989, pp. 272-282.

[32] W. H. Kohler, "A Preliminary Evaluation of the Critical Path Method for Scheduling Tasks on Multiprocessor Systems", IEEE Transactions on Computers, Vol. 24, No. 12 (December 1975), pp. 1235-1238.

[33] L. I. Kontothanassis, "The Mercury User's Manual", Technical Report Draft, Computer Science Department, University of Rochester, September 1992.

[34] C. P. Kruskal, A. Weiss, "Allocating Independent Subtasks on Parallel Processors", IEEE Transactions on Software Engineering, Vol. 11, No. 10 (October 1985), pp. 1001-1016.

[35] E. L. Lawler, C. U. Martel, "Scheduling Periodically Occuring Tasks on Multiple Processors", Information Processing Letters, Vol. 12, No. 1 (February 1981), pp. 9-12.

[36] D. Lenoski, J. Laudon, K. Gharacharloo, W. D. Weber, A. Gupta, J. Hennessy, M. Horowitz, S. M. Lam, "The Stanford DASH Multiprocessor", IEEE Computer, Vol. 25, No. 3 (March 1992), pp. 63-79.

[37] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, J. Hennessy, "The DASH Prototype: Logic Overhead and Performance", IEEE Transactions on Parallel and Distributed Systems, Vol. 4, No. 1 (January 1993), pp. 41-61.

[38] S. T. Leutenegger, M. K. Vernon, "The Performance of Multiprogrammed Multiprocessor Scheduling Policies", Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, May 1990, pp. 226- 236.

[39] S. Lucco, "A Dynamic Scheduling Method for Irregular Parallel Programs", Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation, June 1992, pp. 200-211.

[40] S. Majumdar, "Processor Scheduling in Multiprogrammed Parallel Systems", Ph.D. Thesis, Department of Computational Science, University of Saskatchewan, April 1988.

[41] E. P. Markatos, T. J. LeBlanc, "Load Balancing vs. Locality Management in Shared-memory Multiprocessors", Technical Report 399, Computer Science Department, University of Rochester, October 1991.

[42] E. P. Markatos, T. J. LeBlanc, "Shared Memory Multiprocessor Trends and the Implications for Parallel Program Performance", Technical Report 420, Computer Science Department, University of Rochester, May 1992.

[43] E. P. Markatos, T. J. LeBlanc, "Using Processor Affinity in Loop Scheduling on Shared Memory Multiprocessors", Technical Report, Computer Science Department, University of Rochester, June 1992.

[44] C. McCann, R. Vaswani, J. Zahorjan, "A Dynamic Processor Scheduling Policy for Multiprogrammed, Shared Memory Multiprocessors", ACM Transactions on Computer Systems, Vol. 11, No. 2 (May 1993), pp. 146-178.

[45] J. C. Mogul, A. Borg, "The Effect of Context Switches on Cache Performance", Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems, April 1991, pp. 75-84.

[46] L. M. Ni, C. E. Wu, "Design Tradeoffs for Process Scheduling in Shared Memory Multiprocessor Systems", IEEE Transactions on Software Engineering, Vol. 15, No. 3 (March 1989), pp. 327-334.

[47] J. K. Ousterhout, "Scheduling Techniques for Concurrent Systems", Proceedings of the 3rd International Conference on Distributed Computing Systems, October 1982, pp. 22-30.

[48] J. L. Peterson, A. Silberschatz, *Operating System Concepts*, Addison-Wesley, 1985.

[49] C. D. Polychronopolous, D. J. Kuck, "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers", IEEE Transactions on Computers, Vol. 36, No. 12 (December 1987), pp. 1425-1439.

[50] D. Pountain, J. Bryan, "All Systems Go", Byte, August 1992, pp. 112-136.

[51] J. Rothnie, "Overview of the KSR1 Computer System", Kendall Square Research Report TR 9202001, March 1992.

[52] S. K. Sahni, “Algorithms for Scheduling Independent Tasks”, Journal of the ACM, Vol. 23, No. 1 (January 1976), pp. 116-127.

[53] M. S. Squillante, E. D. Lazowska, “Using Processor-Cache Affinity Information in Shared-Memory Multiprocessor Scheduling”, Technical Report 89-06-01, Department of Computer Science and Engineering, University of Washington, February 1990.

[54] M. S. Squillante, “Issues in Shared Memory Multiprocessor Scheduling: A Performance Evaluation”, Technical Report 90-10-04, Department of Computer Science and Engineering, University of Washington, October 1990.

[55] M. S. Squillante, R. D. Nelson, “Analysis of Task Migration in Shared-Memory Multiprocessor Scheduling”, Proceedings of the 1991 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, May 1991, pp. 143-155.

[56] P. Tang, P. C. Yew, “Processor Self-Scheduling for Multiple Nested Parallel Loops”, Proceedings of the 1986 International Conference on Parallel Processing, August 1986, pp. 528-535.

[57] S. S. Thakkar, M. Sweiger, “Performance of an OLTP Application on Symmetry Multiprocessor System”, Proceedings of the 17th Annual International Symposium on Computer Architecture, May 1990, pp. 228-238.

[58] D. Thiebaut, H. S. Stone, “Footprints in the Cache”, ACM Transactions on Computer Systems, Vol. 5, No. 4 (November 1987), pp. 305-329.

[59] R. H. Thomas, W. Crowthier, “The Uniform System: An Approach to Runtime Support for Large Scale Shared Memory Parallel Processors”, Proceedings of the 1988 International Conference on Parallel Processing, August 1988, pp. 245-254.

[60] J. Torellas, A. Tucker, A. Gupta, “Evaluating the Benefits of Cache-Affinity Scheduling in Shared-Memory Multiprocessors”, Technical Report: CSL-TR-92-356, Computer Systems Laboratory, Stanford University, August 1992.

[61] A. Tucker, A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed Shared Memory Multiprocessors", Proceedings of the 12th ACM Symposium on Operating Systems Principles, December 1989, pp. 159-166.

[62] T. H. Tzen, L. M. Ni, "Dynamic Loop Scheduling for Shared Memory Multiprocessors", Proceedings of the 1991 International Conference on Parallel Processing, August, 1991, pp. 247-250.

[63] R. Vaswani, J. Zahorjan, "The Implications of Cache Affinity on Processor Scheduling for Multiprogrammed, Shared Memory Multiprocessors", Proceedings of the 13th ACM Symposium on Operating System Principles, October 1991, pp. 26-40.

[64] A. Weinrib, G. Gopal, "Decentralized Resource Allocation for Distributed Systems", Proceedings of the 6th Annual Conference on Computer Communications (Infocom 1987), March 1987, pp. 328-336.

[65] J. Zahorjan, C. McCann, "Processor Scheduling in Shared Memory Multiprocessors", Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, May 1990, pp. 214-225.

[66] J. Zahorjan, E. D. Lazowska, D. L. Eager, "Spinning versus Blocking in Parallel Systems with Uncertainity", Proceedings of the International Symposium on Performance of Distributed and Parallel Systems, December 1988, pp. 445-462.

[67] J. Zahorjan, E. D. Lazowska, D. L. Eager, "The Effect of Scheduling Discipline on Spin Overhead in Shared Memory Parallel Systems", IEEE Transactions on Parallel and Distributed Systems, Vol. 2, No. 2 (April 1991), pp. 180-198.