ABSTRACT

VOUK, NIKOLA. **Buddy Threading in Distributed Applications on Simultaneous Multi-Threading Processors** (Under the direction of Dr. Frank Mueller)

Modern processors provide a multitude of opportunities for instruction-level parallelism that most current applications cannot fully utilize. To increase processor core execution efficiency, modern processors can execute instructions from two or more tasks simultaneously in the functional units in order to increase the execution rate of instructions per cycle (IPC). These processors implement simultaneous multi-threading (SMT), which increases processor efficiency through thread-level parallelism, but problems can arise due to cache conflicts and CPU resource starvation.

Consider high end applications typically running on clusters of commodity computers. Each compute node is sending, receiving and calculating data for some application. Non-SMT processors must compute data, context switch, communicate that data, context switch, compute more data, and so on. The computation phases often utilize floating point functional units while integer functional units for communication. Until recently, modern communication libraries were not able to take complete advantage of this parallelism due to the lack of SMT hardware.

This thesis explores the feasibility of exploiting this natural compute/communicate parallelism in distributed applications, especially for applications that are not optimized for the constraints imposed by SMT hardware. This research explores hardware and software thread synchronization primitives to reduce inter-thread communication latency and operating system context switch time in order to maximize a program's ability to compute and communicate simultaneously. This work investigates the reduction of inter-thread communication through hardware synchronization primitives. These primitives allow threads to "instantly" notify each other of changes in program state. We also describe a thread-promoting buddy scheduler that allows threads to always be co-scheduled together, thereby providing an application the exclusive use of all processor resources, reducing context switch overhead, inter-thread communication latency and scheduling overhead. Finally, we describe the design and implementation of a modified MPI over Channel (MPICH) MPI library that allows legacy applications to take advantage of SMT processor parallelism. We conclude with an evaluation of these techniques using several ASCI benchmarks. Overall, we show that compute-communicate application performance can be further improved by taking advantage of the native parallelism provided by SMT processors. To fully exploit this advantage, these applications must be written to overlap communication with computation as much as possible.

Buddy Threading in Distributed Applications on Simultaneous Multi-Threading Processors

by

Nikola Vouk

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial satisfaction of the
requirements for the Degree of
Master of Science in Computer Science

Department of Computer Science

Raleigh, North Carolina

2005

Approved By:

Dr. Vincent Freeh	Dr. Michael Rappa

Dr. Frank Mueller
Chair of Advisory Committee

Biography

Nikola Vouk was born on April 11th, 1980. Born in Zagreb, Croatia, he moved with his family to Raleigh, North Carolina at the age of 5. He received a Bachelor of Science in Computer Science from North Carolina State University in 2001, and continued his graduate studies at the North Carolina State University in Fall of 2002. With the defense of this thesis, he will receive a Master of Science in Computer Science degree from NCSU in May 2005.

Acknowledgements

I would like to thank my parents, Mladen and Maja, for being the best parents and for supporting me throughout my school years. I would like to thank Dr. Michael Rappa for his overwhelming support, generosity, faith and encouragement. I would like to especially thank my advisers Dr. Frank Mueller and Dr. Vincent Freeh for their invaluable advice and guidance.

Finally, I would like to thank all of my friends: Richard, Josh, Reid, Ben, my brother Alan, and especially Meeta Yadav for being very kind, supportive, helpful and encouraging. Without all of you, this thesis would not have been possible.

Table of Contents

List of Figures vi

List of Tables viii

1 Introduction 1

2 SMT Architecture 5
2.1 Background 5
2.2 Processor Architecture Design 9
2.2.1 Pentium 4 9
2.2.2 IBM Power 5 11
2.2.3 Operating System Support for SMTs 13
2.3 CMP and SMP 14
2.4 Conclusion 15

3 Hardware Synchronization 16
3.1 Current Thread Synchronization Primitives 17
3.1.1 Blocking Synchronization Primitives 21
3.2 Evaluation 23
3.2.1 Design 23
3.3 Conclusions 29

4 Scheduler Assisted Promotional Buddy Threading 30
4.1 Technical Details 31
4.2 Linux Kernel Buddy Threading Kernel Modification 31
4.2.1 Kernel Buddy System Specification 31
4.2.2 Implementation Details 32
4.3 Buddy Threading Performance Results 35
4.4 Promotional Buddy Scheduling and the Standard Kernel Thread Facilities 38
4.5 Buddy Scheduling for MPICH 39

5 MPICH MPI Communication Library 40
5.1 Asynchronous/Synchronous and Blocking/Non-Blocking Functions 40
5.2 MPICH Channel P4 Code Structure 42
5.3 Modified Code Structure 45
5.3.1 MPICH Re-Design 45
5.4 Evaluation 46
5.4.1 Synthetic Benchmark Results 46

6 ASCI Purple Benchmarks **50**
6.1 Implicit Radiation Solver (IRS) . . . 50
6.1.1 Description . . . 50
6.2 SMG2000 . . . 53
6.2.1 Description . . . 53
6.2.2 Messaging Profile and Execution Analysis . . . 53
6.3 sPHOT . . . 54
6.3.1 Description . . . 54
6.4 SPPM . . . 56
6.4.1 Description . . . 56
6.5 Sweep3d . . . 58
6.5.1 Description . . . 58

7 Benchmark Evaluations **60**
7.1 Experiment Setup . . . 60
7.2 Benchmark Evaluation . . . 60
7.2.1 IRS . . . 61
7.2.2 SMG 2000 . . . 63
7.2.3 sPHOT . . . 65
7.2.4 sPPM . . . 66
7.2.5 Sweep 3D . . . 67

8 Related Work **72**

9 Conclusions **74**

10 Future Work **76**

A Build Environment Instructions **77**
A.1 How to use this repository . . . 77
A.2 Setup . . . 77
A.3 Setup benchmarks . . . 79
A.4 Compiling . . . 80
A.5 Distributed Compiling: . . . 81
A.6 Testing benchmarks . . . 81
A.7 Miscelanious Tools . . . 82

Bibliography **83**

List of Figures

2.1 P4 SMT Internal Structure (After [BHMU02]) 10
2.2 Power 5 processor Internal Structure (After [RKT04]) 11

3.1 Basic Spin-Lock Implementation 16
3.2 MWAIT/MONITOR Kernel Internal Memory Design 18
3.3 MWAIT/MONITOR System Call User Space Memory Design 19
3.4 MWAIT/MONITOR Call User-Space Memory Design 20
3.5 Synchronization Primitive Notify and Wait() 23
3.6 Synchronization Primitive Call Overhead 24
3.7 Synchronization Primitive Latency Test 26
3.8 Synchronization Primitive CPU Impact: 28

4.1 Promotional buddy-scheduler Code 34
4.2 Promotional Buddy Scheduling Evaluation Kernel 35
4.3 Synchronization Primitive Latency vs Number of Background Tasks 36

5.1 Original MPICH Code Structure - High Level Design 41
5.2 Isend Message Latency 44
5.3 MPICH Modification High Level Design 45
5.4 MPICH Isend overhead benefit measured in floating point operations not running in KML 46
5.5 MPICH Isend overhead benefit measured in floating point operations with KML privileges 47
5.6 MPI_Isend() Test Kernel 48

6.1 IRS Communcation (After [VY02]) 52
6.2 SMG2000 Communcation (After [VY02]) 54
6.3 sPHOT Communication (After [VY02]) 56
6.4 sPPM Communication (After [VY02]) 57
6.5 Sweep3D Communication (After [VY02]) 59

7.1 IRS Results (Time) 61
7.2 IRS Results - Percentage Improvement over “Normal” 62
7.3 SMG2000 Kernel 63
7.4 SMG2000 Results - Percentage Improvement over “Normal” 64
7.5 SMG2000 Results (Time) 65
7.6 sPHOT Results 66
7.7 sPHOT Results (Time) 67
7.8 sPPM Results 68

7.9 sPPM Results (Time) 69
7.10 Sweep 3D Results 70
7.11 Sweep 3D Results (Time) 71

List of Tables

2.1 Processor Resource Sharing Comparison 12

3.1 Synchronization Primitives Notify() Call Overhead (lower is better) 25
3.2 Synchronization Primitives Notification Latency Table (lower is better) 26
3.3 Time to complete the fixed amount of work with interfering (lower times are better) 28

4.1 Promotional Buddy Scheduler Code Functions 33
4.2 Zero Background Interference Tasks with Promotional Thread Scheduling 37

5.1 MPICH Channel Sending Modes 42

6.1 Benchmark Summary 51

Chapter 1

Introduction

Transistor microprocessors have evolved through several distinct stages since the first successful ones were built at Intel in 1971. Each new stage brought greater performance, higher clock speeds, more dense circuitry and most importantly, greater efficiency in utilization of the whole processor. In the beginning was the simple single-instruction scalar processor capable of processing one instruction at a time. Hardware was very slow and software frequently had to wait for the hardware to catch-up. Processors were then pipelined to prevent parts of the processor from remaining unnecessarily idle when instructions were available to execute. In the 1980's, supercomputers improved on this single instruction execution by developing vector processing techniques (Single Instruction Multiple Data) such as the Cray supercomputers. This allowed one instruction to operate on multiple memory values to improve efficiency. The Cray-1 introduced the next evolutionary stage in processor design, super-scalar processors. This improved upon the basic processor by allowing multiple instructions to be issued in every cycle, which improve single processor efficiency and performance through parallelism. During the 1990's, out-of-order execution super-scalar processors improved primarily through die-shrinkage, processor clock increases and increased processor cache size. Hoare, in his 1978 paper "Communication Sequential Processes", described how traditional single sequential programs are sped up transparently through hardware (multiple functional units, multiple processors) or software (I/O controllers and multiprogramming)[Hoa78].

Because most microprocessors had one program context to execute all threads and due to the nature of current applications (word processors, web browsers, music players, Computer-Aided-Design. programs, etc.), application were not able to utilize all the available resources. Even those resources that were being used may be under-utilized due to memory stalls or dependences. To combat this, the processor designers pushed efficiency improvement onto the compiler writers with the very-long-instruction-word (VLIW) architecture and later the EPIC instruction set from Intel[HMR+00]. The compiler was tasked with scavenging the program control-flow diagrams for enough independent instructions to be encoded into a single long instruction word which the processor would then decode and process more efficiently because it has many instructions at hand to execute in a given cycle. Unfortunately, the basic blocks in most programs are too small to efficiently fill up the pipeline with VLIW instructions. The Intel Itanium processor (an EPIC instruction set processor) can encode eight instructions into one long instruction, but on average the compiler is only able to schedule 3-4 instructions together and the remaining is filled with non-operations (NOP). As Tullsen describes in his paper 'Simultaneous Multithreading: Maximizing On-Chip Parallelism', "The objective of SM[T] is to substantially increase processor utilization in the face of both long memory latencies and limited available parallelism per thread" [TEL95]. The new bottleneck in microprocessor performance was not the hardware itself, but the code executing did not have enough independent instructions to take full advantage of all the processor's resources. The Tera computer was one of the first supercomputers to attack this bottleneck[Wal91]. The Tera computer implemented fine-grain multi-threading with multiple contexts, register files and re-order buffers–one for each independent thread. An instruction is fetched from each context on every clock cycle in a round-robin order. Each context's state and instructions are independently tracked, and only the currently selected context may execute instructions. This solved one problem – starvation. If one process blocks, then another process can quickly issue instructions. But, this did not solve under-utilization of cycles by processes. Thus, Herata et al. proposed a refinement to fine-grained multi-threading called Simultaneous Multi-threading (SMT)[HKN$^+$92]. SMT refines fine-grained multi-threading to one set of functional units, re-order buffers and reservation stations while duplicating the processor context (register file). Multiple threads can execute simultaneously onto the same set of functional units. This allows for maximum efficiency by allowing any running process to issue instructions when there is an available execution slot. The problem of processor starvation has been solved by over-subscribing threads on the same hardware to take advantage of any slack one thread may provide.

This thesis investigates thread scheduling and pairing techniques to take advantage of simultaneous multi-threaded processors. We study how simultaneous multi-threaded processors can be used to improve performance of individual distributed applications by taking advantage of inherent program parallelism through SMT hardware's explicit parallelism. We study how operating system scheduling support and specialized CPU synchronization instructions can be used to reduce operating system synchronization latency to improve performance in conjunction with application parallelism. We show that using the low-latency Pentium 4 Prescott MWAIT memory monitor instruction, we can improve inter-thread communication by 70% over traditional synchronization methods during computation while simultaneously reducing CPU resource contention. In addition, the operating system scheduler can reduce inter-thread-latency even further through thread promotion buddy scheduling, whereby a given thread is paired with a 'buddy' thread that is always run alongside it on the same SMT processor. This provides up to 40% improvement in thread-response for certain applications. We describe our modifications to the MPICH MPI communications library to seamlessly divide the communication and computation operations, and we present our results showing 30% improvement in certain benchmarks.

To take advantage of the communication-computation parallelism, we exploit several asynchronous and synchronous communication facilities provided by the operating system and the MPICH communication library through blocking and non-blocking system calls. It is important to distinguish the subtle differences between asynchronous and synchronous, and blocking and non-blocking terminology. Andrews describes that "with asynchronous message passing, communication channels are unbounded queues of messages where...[the] execution of send does not block the sender"[And91]. The calling function is non-blocking. This allows the calling thread to continue its execution even though the operation has not yet been completed by the function. The completion of the non-blocking call is handled outside the scope of the requesting task (asynchronously) by the operating system or another independent task. Synchronous calls are defined as subroutines that retain control until the operation is fully completed. The caller can make no other progress in the meantime[Mil87]. A blocking operation is synonymous with a synchronous operation, because both block until their operation completes. Non-blocking operations are those defined to not block the calling function while the operation completes.

Examples of each type are frequently found in modern operating systems. The most basic write() system call operation is a blocking subroutine that synchronously transfers data through sockets to a receiver. Upon complete reception, the sender is able to progress. Failure in reception of data will result in an error. A blocking asynchronous example is a simple server performing a listen() awaiting new connections. The application blocks until some other process attempts to connect to the sockets being listened to. The listen() then completes and returns to the caller. A non-blocking synchronous operation does not make any practical sense, but non-blocking asynchronous functions are common. A select() system call polling on a set of file descriptors only informs the calling subroutine as to the number of ready sockets available. Select() does not wait for new connections. The operating system servicing system interrupts is a completely asynchronous event as well as a program servicing events.

Each of these examples lends itself to different amounts of parallelism that can be taken advantage of with SMT processors. SMT architectures are best for non-blocking asynchronous functions, such as servicing communication activity. We take advantage of SMT processors through the MPICH communication library. The MPICH library is a synchronous library with limited asynchronous capabilities. We modify it to become fully asynchronous and non-blocking to maximize overlap of communication and computation and leverage the SMT microprocessor.

In our discussions regarding synchronization primitives (the MPICH library modifications and the Promotional Buddy Scheduler), we will frequently refer to the interaction between a pair of involved threads that have distinct names according to their purpose. Our modified MPICH library is divided into two distinct threads: The computation thread and the communication thread. Sometimes the communication thread is also referred to as the helper thread, because designed the

MPICH modification such that the communication functions should be handled by a symbiotic helper thread. In addition, the MPICH library supports a listener thread to handle notification of new asynchronous data connections. These threads are actual executing tasks derived from the serial MPICH library. Our synchronization primitive kernels will frequently refer to these threads, because they were designed to model the functions used in our modified MPICH design. The Promotional Buddy Scheduler is in no way related to our modified MPICH library, but our library can utilize it. The Promotional Buddy Scheduler designates two tasks as buddies. The designating task is referred to as the master control thread while any co-scheduled tasks are buddy tasks. The most common usage of the Promotional Buddy Scheduler with the modified MPICH library is to designate the computation thread as the master control task and the communication thread as the buddy task, but buddying is not required. Finally, as the Linux kernel hardly distinguishes between processes and tasks, in this document all references to processes, threads and tasks refer to individual kernel scheduled entities known as tasks.

This thesis is structured as follows: In chapter 2, we summarize the history of SMT architecture and the current state of research. Chapters 3, 4 and 5 describe the MWAIT instruction as an optimal synchronization primitive, changes to the operating system to support promotional buddy scheduling and modifications to the MPICH library to split computation and communication in the library, respectively. Chapter 6 describes each of the benchmarks, and chapter 7 presents results of those benchmarks using the modified MPICH, thread synchronization and pairing techniques. We then conclude our results in chapter 9, and finally describe related work and present plans for future work.

Chapter 2

SMT Architecture

Simultaneous Multi-threading, as a processor hardware technique for improving thread-level parallelism, has been shown to increase processor efficiency (IPC), decrease memory latency, provide application-specific performance effects and to side-step performance bottlenecks and their solutions [TEL95, Luk99, LBE$^+$98, ST97, OSU99]. Other novel techniques to take advantage of SMT processores explore speculative pre-computation, improved branch prediction, task scheduling policies and starvation prevention policies. Several competing processor architectures are available that implement many of these techniques. We compare and contrast these architectures and conclude by investigating a complementing technology, chip multi-processing, to consider future trends in processor design [Luk99, WCT98, HPR$^+$02].

2.1 Background

Scientific computing has improved the performance of parallel processing applications on shared memory and massively distributed parallel architectures by increasing processor clock frequencies and creating ever larger computing clusters. Power consumption, increasingly complex processor designs and fabrications problems are limiting processor performance. As a result, recent research has focused on making more efficient use of processors by placing multiple cores on a single chip (Chip Multi-Threading) and multiple program contexts on a single core (Simultaneous Multi-Threading), rather than increasing processor pipelines and clock frequencies. Simultaneous Multi-threading was developed to counter inefficient use of processor resources on super-scaler processors by a single program creating horizontal and vertical waste, which other processes could utilize. Horizontal waste occurs when a processor is not utilizing all the functional units available in a given cycle, because of not enough instruction-level parallelism in the task or ready instructions. Vertical waste occurs when the processor pipeline stalls and zero instructions are issued to the processor in a cycle. Tullsen shows how modern super-scaler processors with minimal extra hardware (additional process contexts, additional instruction-fetch unit logic and additional re-order buffer (ROB) memory) can reduce horizontal and vertical waste of resources and hide memory latencies through transparent hardware thread-level parallelism [TEL95]. Allowing a second task to execute on the same processor core doubles the probability that there will be available instructions to issue in any given cycle. Even if one task's instruction stream has stalled on memory references, then the overall IPC of the CPU will not drop to zero, because the stalled memory references' latencies are "hidden" by continued execution of other tasks' instructions. SMT architectures have been shown to have no lower IPC than that of single context superscalar processors and on average a higher IPC [Ros05].

With increased processor utilization, issues like functional unit contention, cache collisions, and starvation may arise. These could lead to individual application performance slowdown, but overall faster execution for multiple processes. This is partly a result of programs being written with the assumption that the L1 and L2 caches are not being shared with any other process simultaneously, and until recently, neither were modern operating system schedulers written with those assumptions in mind. Even with proper operating system schedulers, multiple independent computationally intensive tasks that are running on the same CPU core simultaneously would experience slow-downs individually[RKT04] [Ros05]. Multi-threaded applications benefit from an inherent shared memory design, but its threads could be completely independent and act similarly to independent processes. These individual processes will likely take longer to execute. They individually will not be able to use all available cycles for themselves as a result of overlapping execution of multiple instruction streams and memory latency hiding. Though, overall the programs would take a less total time to run than if run serially. Threads of execution can take advantage of another thread's memory stalls to execute a few more instructions without having to wait for a timer interrupt and the resulting context switch. Other benefits of the SMT architecture include the ability to hide memory access latency and branch mis-prediction latency hiding. When an instruction stalls in the issue queue due to a load miss, any dependent instructions are also stalled. Even with re-order buffers of 256 instructions in size, starvation could occur because of the memory wall [WM95]. Tullsen predicts that re-order buffers need to be three times the size of their super-scaler counterparts in order for four to eight contexts to provide enough independent parallelism without stalling[TEL95].

Tullson and Eggers pioneered the research on SMT processors [TEL95]. They showed that individual processes might not always be able to issue to all functional units (horizontal waste), nor issue any instructions in a given cycle due to delays (vertical waste). On average, the CPU instructions per cycle (IPC) issued may be as low as 1.5 on an 8 issue super-scalar processor. Given enough process instruction streams, they were able to get an IPC between 6 and 7 on an 8 issue SMT super-scalar processor. It was shown that the limiting factors for SMT processors are issue bandwidth and cache conflicts. Eight tasks, running simultaneously, can quickly overwrite each other's cache contents and cause extra cache misses. It is important for issue bandwidth to match the number of functional units to maximize IPC and for the re-order buffer to grow accordingly, but with increased issue bandwidth and re-order buffer size comes greater cost and design complexity. The current Pentium 4 Prescott is a four-issue, seven functional unit processor with a 128 entry re-order buffer[Sto01]. The average IPC on this processor is between 0.5 and 1.5 IPC [Sto02].

It is important to distinguish several distinct strains of SMT processor research and their usage. Some research studies the effects of natural thread-level interaction occurring when random processes simultaneously execute. They want to know, in the general case, what improvements and limitations the additional contexts impose and how best to utilize them. Other studies focus on application-specific research, such as SMT performance on typical work loads of certain common tasks to determine their optimal configuration and benefit, while others study alternative applications of the hardware independent of the overlying software. Examples include improving branch mis-prediction and dynamic cache pre-fetching, which are two primary performance limitations of super-scalar processors [Wal91].

Many of the application-specific studies have focused on multimedia workloads using video and audio encoding. Lo et al. studied database workload behavior on SMT processors and concluded that the processor memory hierarchy is the primary limiting factor for database workload performance[LBE+98]. Databases do have enough inherent parallelism to support increased IPC on SMT processors, but modifications in the application memory layout and access are required to accommodate the shared memory nature of SMT processors. Chen et al. and Oehring studied multimedia applications and found that issue bandwidth, memory layout, and SMT-unaware algorithms are the main limitations in achieving improved performance in multimedia workloads [OSU99, YKME+02].

Another novel technique being studied is speculative pre-execution is a dynamic runtime form of pre-fetching that attempts to preload the cache ahead of the primary thread by skipping ahead of some instructions or functions [PSR00, HPR+02]. Speculative pre-execution can be performed on a function level, intra-function or based on array access patterns. The run-time environment or compiler spawns a thread of execution that executes slightly ahead of the main execution thread. It executes most of the program only cycles ahead of the original program to force unpredictable cache misses to be prefetched. The primary thread would benefit hundreds of cycles later by having that data in the local L1 or L2 cache. Some problems that can occur with speculative pre-execution include: the threads becoming out of sync by speculative threads going too far ahead of the main thread; Speculation of a wrong execution path; The speculation thread not far enough ahead, or falls behind causing CPU and cache conflicts. We have tested speculative pre-execution on the Pentium processor using a technique similar to Wang et al., and found that precise control over thread timings and code specific anomalies made using this technique very difficult.[HPR+02] The software implementation was too coarse that results in frequent run-behinds and run-aheads. Hence, techniques and architectures like the slipstream processor and threaded multiple execution use processor-level threading techniques to spawn speculative pre-execution threads[PSR00, WCT98]. A side effect of these speculative threading techniques is improved branch prediction. Because there is a speculative thread, the processor can determine whether a particular branch will be taken ahead of time. Speculative pre-execution threads execute hundreds or thousands of cycles ahead of the main application to force cache load misses to be pre-fetched just-in-time. Both techniques proved to have limited success, but complicated designs and a need for a large number of available contexts to reap a benefit has kept speculative pre-execution experimental. Others have used similar threading ideas for use in fault tolerance[RM00, VkPC02]. Currently, many fault tolerant computers replicate whole chips, memory controllers, even motherboards, but researchers have been able to show that SMT processors can be used for fault-tolerance purposes at the CPU level by running two copies of the same programming and comparing results. By running multiple copies of a program slightly offset from one another and comparing results of loads and stores, the processor can determine if a transient error has occurred and correct for it. The trailing process also benefits from active pre-fetching and temporal locality initiated by the leading pre-execution thread.

With many papers claiming cache contention as a primary reason for decreased program performance, Gropp et al. and others studied cache effects on SMT processors[SDR01, OSU99, TEL95, LBE$^+$98]. They investigated dynamic cache partitioning algorithms with varying eviction algorithms. They concluded that the cache hierarchy is the primary issue in making programs perform well on SMTs, because of too many cache conflicts. But, they conclude that giving threads private allocations of the cache does not improve performance of applications. Instead, threads should have access to the whole cache, because an application's working set of data is frequently larger than the allotted cache size and could fill more than half the cache. The loss of cache space and extra misses is beneficial to too few applications for widespread implementation. These observations may have led to IBM's decision to use, in its first generation Power 5 processor, a combination of hardware and software metrics monitoring L2 miss rates, instruction issue rates and instruction groups to determine whether the threads should be co-scheduled on the same SMT processor, re-scheduled, throttled or given other priority [Sto04].

Much of the research has focused on the simultaneity of SMT processors and on methods to best take advantage of SMT processors with current software. These studies investigate how to better utilize SMT techniques to overcome super-scaler processor deficiencies to get more efficient execution. Researchers have focused on solutions to the general problem of adapting existing algorithms and programs to the new shared L1/L2/Functional Unit environment. Conclusions reached by most papers determine that individual applications, or independent threads of an application, who simultaneously share the processor with other applications or threads may suffer individually, but all applications will run, in total, to completion faster together than if run serially. We contend that, aside from desktop usage, that the majority of applications run alone or expect to dominate the environment they are executed in. Many computing facilities have long run jobs in batches, whereby applications are queued and expect to control the computer solely. In practice, very few jobs run simultaneously, because of unpredictable runtime performance. These opposing viewpoints have to be resolved or accommodated in every computing facility. Given that most applications expect to run on a dedicated processor, innovative solutions to leverage the SMT hardware for these applications like speculative pre-execution, scheduler interaction or processor design can be used to boost performance [Sto04]. This thesis provides operating system and software support for individual distributed MPI applications to covertly take advantage of the underlying SMT hardware, while improving performance of traditionally serially run applications. The MPICH library is a system-provided application-independent static library that can facilitate this symbiotic relationship. We do not intend to make the most efficient use of the SMT processor. Instead, we promote taking advantage of the full processor capabilities to leverage the naturally existing parallelism in MPI programs. In addition, our operating system buddy scheduler thread promotion allows the application to best utilize the processor completely with very low latency thread synchronization primitives helping to mitigate inter-thread communication overhead.

2.2 Processor Architecture Design

Simultaneous Multi-threading, in broad terms, refers to having multiple architecture contexts attached to a single processor core that share a set of functional units. Implementing this broad definition can take many different forms that could affect how best to optimize and implement operating systems and software for such processors. Most research into SMT processors has focused on a modified Alpha processor that was designed, but never released. Since Intel Corporation bought Compaq, that research will likely emerge in some future Intel microprocessor revision. The current Pentium 4 processor implements a first generation SMT architecture that has some restrictions and a limited number of contexts. IBM has released (September 2004) the Power 5 SMT processor. This is a multi-core, multi-context processor that has benefited from all the research available, including that of the Pentium 4. We focus on the Pentium 4 processor and the Power 5 processor here, as they are the only mainstream SMT processors available now, and how operating systems and software have been optimized to handle these underlying architectures.

2.2.1 Pentium 4

The Intel Pentium 4 processor is the first commercial SMT processor available on the market. This first iteration is a two context processor with a claimed 5% increase in core area to accommodate the SMT hardware. The microprocessor is a standard Pentium 4 that implements the Netburst micro-architecture with trace cache [ia304]. This processor can run in multi-task mode (MT), where both contexts are active and some processor resources are split, or single-task mode (ST), where a single thread has access to all processor resources.

The modifications to the processor can be broken down into replicated, shared and partitioned resources. In accommodating the second context, the register renaming logic, instruction pointer, instruction TLB cache, return stack and the register alias tables were replicated. Resources shared between the two tasks includes the L1, L2, L3 and trace caches, the micro-architectural registers and the functional units. The most interesting and performance impacting feature is the partitioning of the resources. When the processor is in MT mode, Intel logically splits the re-order buffer (ROB) size, issue queue size, instruction queues and the L1 trace cache for each thread. Figure

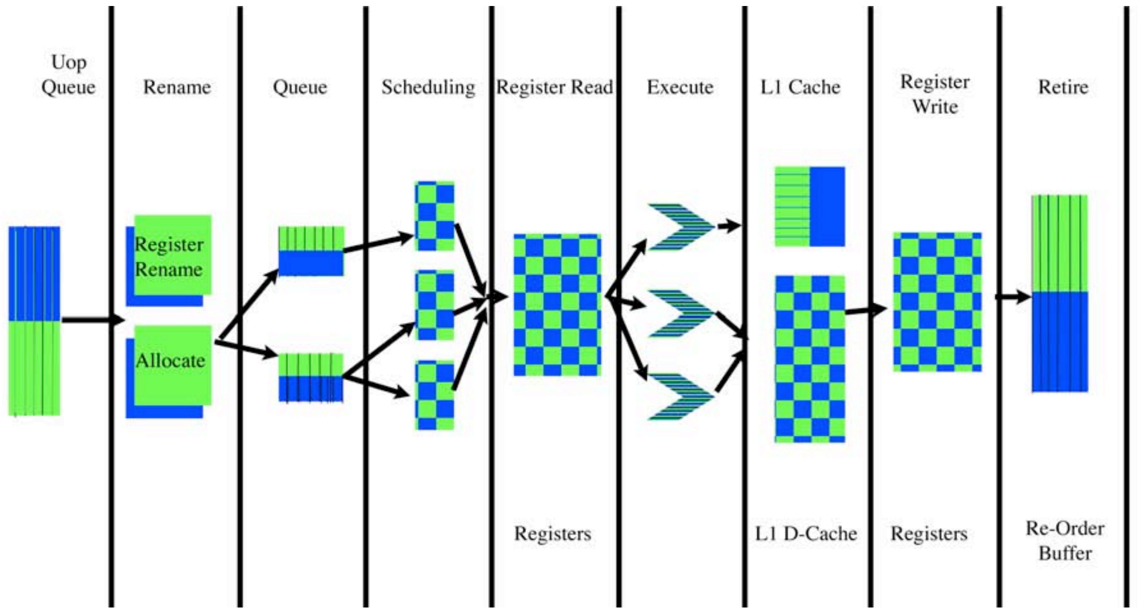


Figure 2.1: P4 SMT Internal Structure (After [BHMU02])

2.1 illustrates this internal division. The purpose of this division is to prevent process starvation, such that no one thread may starve if a greedy thread begins to consume processor resources.

The processor can dynamically partition and unpartition these queues depending on processor state. To ameliorate the negative impact of halving the internal data structures, there are two instructions available to unpartition the processor – MWAIT and HLT. These instructions are frequently used in operating system idle loops to free up resources for other applications. MWAIT triggers a quick transition to a low-power state from which the processor can wake up quickly upon executing a write to a given address range. Transitioning back-and-forth between ST and MT mode requires allowing one of the contexts to stop fetching and completing any waiting instructions, then recombining the resources with the other context. The thread-imposed limits are more soft, rather than hard limits, because queues are not physically halved, but each thread is only allowed to fill half the slots in any order. If in MT mode, the processor alternates fetching new instructions from both threads. The Pentium 4 can only fetch one instruction at a time from the user, but the instruction trace cache can issue 3 micro-operations per cycle to compensate [Sto01]. The majority of the processor after the fetch stage is oblivious to the SMT context. Aside from the division of

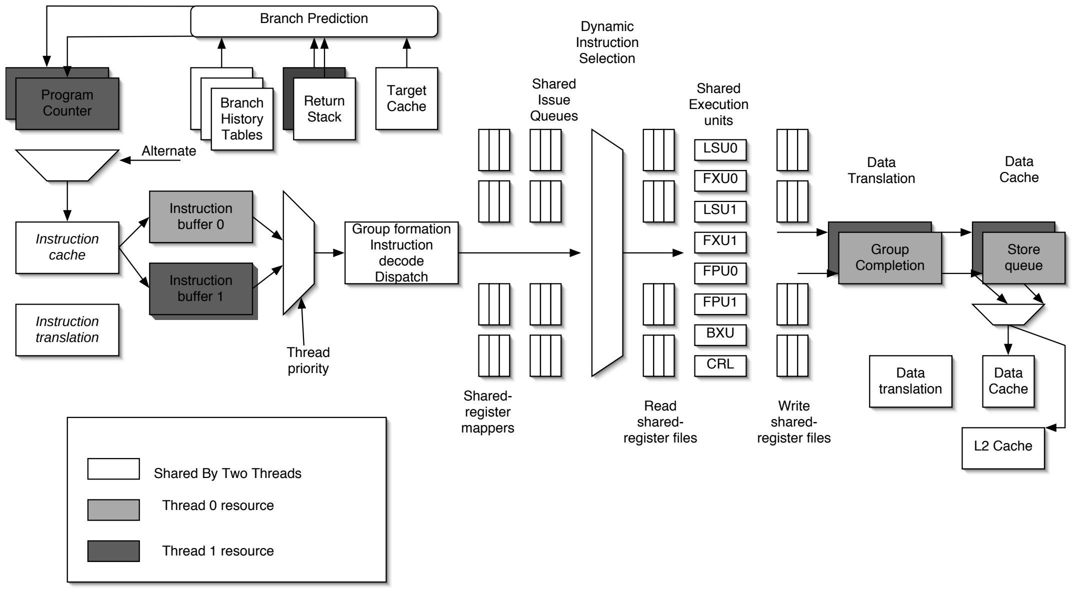


Figure 2.2: Power 5 processor Internal Structure (After [RKT04])

queues, the microprocessor treats both contexts equally.

2.2.2 IBM Power 5

The IBM Power 5 processor is the latest generation in the Power processor line, and the first Power processor to utilize SMT [RKT04]. This processor has been released after the Pentium 4. It is a dual core chip multi-processor with two contexts per core. The Power 5 relies more on processor metrics and priorities to control program flow than brute force division of internal data structures. Replicated resources include the program counter, the instruction queues, the return stack, group completion table and the data cache store queue. All other resources, including branch history tables, caches, instruction group formation, issue queues and functional units, are shared by both threads and access to said resources is controlled by task specific priorities. This task priority limits the number of instructions decoded from a task. This allows the other thread to utilize more of the CPU. Each core of the Power 5 processor supports two modes: single-thread and multi-thread similar to the Pentium 4. Single-thread mode allows one task to consume all of the CPU's resources, while MT mode allows both tasks to utilize the resources. Table 2.1 describes the modifications to the processors in order to accommodate MT mode.

There are some important enhancements in the Power 5 that improve upon the Pentium 4

Resource Type	Pentium 4 HT	Power 5
Replicated	Register renaming logic Instruction Pointer Instruction TLB cache Return Stack Register Alias Tables	Instruction Buffer Group Completion Table Store Queues
Shared	L1, L2, L3 caches Functional Units Micro-architectural registers	L1, L2, L3 cache register files branch history tables
Partitioned	μ op queue dispatch queues issue queues re-order buffer	none

Table 2.1: Processor Resource Sharing Comparison

implementation. The Power 5 implements a soft limit on the threads at the decode stage to control thread access. The Power 5 itself monitors the behavior of each task and their behavior running together in the resource-balancing logic. It monitors the Global Completion Table (GCT), L2 cache miss rate and the issue queues for signs of starvation and performance impeding situations. The Global Completion Table keeps track of completion of groups of instructions in the processor. Once all instructions from a group have completed, they are then committed to the register files. Processes that use more than a predefined number of GCT entries are throttled back with a lower priority. This results in fewer instructions being decode and thereby second thread does not become starved. If the hardware notices that a given thread incurs a predefined number of L2 cache misses, then the resource-balancing logic throttles the thread's instruction decoding until the cache congestion clears. If the logic notices a backup in some issue queue, it knows that a long running instruction, like a memory sync, has stalled the processor. The processor then flushes all the instructions of that thread waiting to be dispatched and holds decoding till the instruction completes. The resource-balance logic keeps the operating system up to date on changes in thread priorities, so it can attempt to optimally schedule the threads [Sto04].

The Power 5 processor has a similar instruction to that of the MWAIT instruction on the Pentium 4 Prescott chip, but cleverly hides it as a processor state. The ST mode is effectively split into two modes. One mode declares the second context to be a null thread from which not to decode any instructions. The other ST mode merely suspends decoding of the second thread until an external or decrementer interrupt occurs [RKT04]. The dormant thread then begins decoding and executing in MT mode.

The Power 5 processor has struck a good balance between the single context super-scalar processor mode and the multi-context SMT processors. It can provide higher processor performance than an equivalently clocked Power 4, while also being able to accommodate highly threaded applications and workloads. This provides the end user with the best of all worlds.

2.2.3 Operating System Support for SMTs

Operating systems are slowly adapting to accommodate SMT processors. All major operating systems on SMT processors have made changes to optimize for the SMT processors. Due to the sophistication of the Power 5, the AIX scheduler is the most advanced out of the Linux, AIX and Windows schedulers compared herein. It has the advantage of having an active feedback mechanism from a cooperative processor to work with. Settle et al. and Rossier both investigated feedback mechanisms for the Pentium 4 processor[Set04, Ros05]. Settle focuses more on cache metrics, while Rossier focuses more on thread IPC for optimal performance. They both found IPC increases from 3-10% depending upon workload. It is telling that the IBM Power 5 uses both of these monitoring metrics to determine thread priority and scheduleability. We will look at how the three major operating systems: IBM AIX, GNU Linux and Microsoft Windows XP each deal with SMT processors.

IBM AIX 5 Scheduler The IBM AIX 5 scheduler seems to be the most advanced scheduler available, because it also has the most advanced SMT architecture to work with. It, along with the resource-balance logic, work in tandem to determine symbiotic threads and destructive threads so it can schedule them accordingly. The Power 5 processor does its own monitoring of CPI and L2 cache misses to determine problem areas, takes evasive action and then notifies the AIX scheduler of the change. The CPU also exposes to the user and operating system an adjustable priority that provides flexibility in scheduling and assigning priority amongst threads. This scheduler/hardware combination seems to be the culmination of all the research for the past ten years, and performs the best among the studied mechanisms.

Linux SMT Scheduler The Linux scheduler has been modified slightly to accommodate SMT processors, but not extensively. All logical processors belonging to the same physical CPU are scheduled from the same run queue and penalties for migrating across logical CPUs were removed. In addition, it prefers tasks to be scheduled onto contexts of the same physical CPU, before moving across physical CPU queues. In addition, active load balancing now balances across physical CPUs, rather than all logical CPUs. This prevents two tasks running on a single physical CPU, while no task runs on a second physical CPU. Currently, there is no support in the scheduler, nor the default Linux 2.6 kernel for using run-time processor metrics feedback[Lov].

The Linux 2.6 kernel goes one step further to help performance on SMT Pentium 4 processors supporting the MWAIT instruction. The MWAIT instruction is used in the idle loop instead of halt in order to allow faster notification and processor wakeup when a task needs to be scheduled. Even so, kernel spin-locks are still NOP loops.

Windows SMT Scheduler The Windows SMT scheduler has been modified by Microsoft for the Pentium 4 HT processor. Their whitepaper does not mention if thread affinity favors moving threads across contexts in the same processor[Not], but it does say that the scheduler attempts to schedule to inactive real processors whenever possible. Windows has implemented kernel locks with the PAUSE instruction. The PAUSE instruction pauses dispatch of instructions for approximately 50 ns to prevent a competing thread from starving, while another thread holds a resource and cannot progress due to functional unit collisions. Windows XP and Windows 2003 .NET aggressively halt the processors in order to release the shared resources of the Pentium 4. Windows does not seem to use the MWAIT instruction.

Each of these schedulers described is meant to provide fair and balanced scheduling for a general purpose operating system. The modifications to make the promotional thread scheduler involve skewing this fairness into unfair rules that allow one application to dominate a processor and the operating system to maximize performance on an SMT processor. It is possible to modify any scheduler to grant exclusive use of a processor, our promotional thread scheduler explicitly keeps paired threads co-scheduled together. In addition, we provide a system-call interface for the user applications to use the special MWAIT and MONITOR instructions as synchronization primitives.

2.3 CMP and SMP

Another technique developed to increase hardware thread parallelism is chip multi-processing. Chip multi-processing places two microprocessor cores on a single chip die. The two cores share the same L2 cache, but are complete processors otherwise. Chip multi-processing provides for more tight-knit multi-processing, because the distance to memory and inter-processor communication time is reduced, but it does nothing for further improving efficiency of the individual cores. There has been much research into the merits of chip multi-processing verses simultaneous multi-threading [ST97, HNO97]. Chip multi-processing can provide increased performance due to increased functional units and caches, but at the expense of space and cost. SMT processors have the advantage of sharing the majority of hardware amongst the contexts with relatively low costs, but do have the issue of cache and functional unit collision. The consensus for upcoming processors (Power 5, Intel Pentium 4 Montecito) is to have two processing cores that are each threaded. The increased functional unit density and shared L2 cache of chip multi-processors helps overall throughput, while the additional contexts and shared L1 core cache helps coupled multi-threaded applications. This seems to be the best balance between CPU efficiency and performance, while also reducing costs by packaging more onto less silicon.

2.4 Conclusion

There has been extensive research into SMT processors: How best to utilize existing hardware; How to best schedule threads on the processors; How to best redesign current applications to take advantage of the SMT microprocessors. We see that the Power 5 is the first in a new generation of SMT processors integrating CMP, SMT and operating support to monitor and adjust thread execution in real time. We believe that CMP multi-processing with SMT capabilities will become mainstream in future processor designs. Applications are becoming more multi-threaded to further take advantage of these processors, but legacy application performance will not benefit from the SMT architecture. We are studying a new technique to provide seamless improvement in performance above and beyond the simple megahertz performance metric.

Chapter 3

Hardware Synchronization

In threaded applications, the common methods to notify threads of a change in state in the program are condition variables, semaphores and spin locks. These common synchronization primitives are optimized for speed, but may have sizable implementation overhead or excessive CPU resource consumption. Blocking on a semaphore or condition variable places the waiting thread on a wait queue and requires a reschedule to occur before the sleeping thread can be awoken. Spin locks are meant to be fas. They cater to very tight and short pieces of code, because anybody spinning on the lock is executing repeatedly a relatively costly memory load and a benign non-operation (nop) without releasing the CPU. This can create two problems for other executing threads. First, if the spinning thread loops for too long, the spinning thread will dominate whatever functional unit executes the CPU NOP. Secondly, the thread could be penalized by the operating system scheduler for consuming too many CPU cycles and may be rescheduled with a shorter execution time slice. Other less disruptive synchronization primitives have been proposed to address these problems. We study five special-purpose data synchronization structures in the Intel architecture. Four data structures are variations of a basic spin-lock (Figure 3.1) and the last is a Pthread condition variable[NBF96].

```
while (WaitVariable == KnownValue) {
        non-operation;
 }
```

Figure 3.1: Basic Spin-Lock Implementation

The issues of thread synchronization can be complicated and involve knowledge about the approximate orders of waiting time (e.g. microseconds, milliseconds, or seconds of time) required in order to choose an appropriate synchronization primitive. It also involves understanding in what program context the threads are running. The programmer must evaluate if any thread-interaction actions are required: Is one thread spinning idly for long periods while waiting for action from another thread; Does one thread wait in general for a very short period of time, while another does some relatively quick computation? The ultimate solution to these questions lies in finding a synchronization primitive that can be used in any of the above situations.

We focus here on the Intel Pentium 4 Prescott processor with hyperthreading (SMT) technology [ia304]. This particular processor features several hardware instructions that can facilitate thread synchronization including MWAIT/MONITOR, PAUSE, NOP, FNOP, Test-and-Set, Compare-and-Swap and HLT (HALT). We are focusing on the MWAIT and MONITOR instructions, because they are low-latency instructions allowing asynchronous hardware notification between threads without system interaction. The other instructions are used in traditional condition variables, semaphores and spin-locks and suffer from the problems described above[Mil87, NBF96]. The MWAIT and MONITOR instructions are fast with low-overhead, like traditional spin-locks, and resource saving, like condition variables. When programming for the Pentium 4 SMT processor, you must take into account that the internal processor queues are split when SMT is enabled to prevent process starvation. Execution of an MWAIT or HALT instruction frees up these queues and temporarily allows the other context to consume all of the processor's resources.

In our modified MPICH implementation, we use thread synchronization in a helper thread that waits for messages from a main computation thread. The helper thread performs quick check of the run queue and either a single MWAIT instruction, a timeout condition variable wait, or a single NOP or PAUSE instruction executes. If there was a requests waiting, then the helper thread executes the request before continuing on to check for network activity and loop again.

3.1 Current Thread Synchronization Primitives

Current thread synchronization techniques have converged on condition variables, semaphores and spin-locks. These software data structures are suported through instructions such as test-and-set, fetch-and-add and compare-and-swap to assure atomicity and exclusiveness of locks in an environment that normally does not provide such guarantees [And91]. Condition variables and semaphores almost always require operating system interaction to deliver signals or schedule threads (user-level threads excluded). We first investigate different types of synchronization primitives available in terms of their pros and cons.

Spin-Locks

Spin-locks are generally reserved for low-latency short loops that cannot suffer the cost of being interrupted or rescheduled. The problem with spin-locks is that they rapidly consume processor integer, fetch and issue resources due to their simplicity. They also present a known issue whereby a spin-lock results in a memory mis-prediction that results in a processor pipeline flush. A basic spin-lock is shown in Figure 3.1.

MONITOR and MWAIT Pentium Instructions

MONITOR/MWAIT was designed to complement the use of HLT and PAUSE instructions for efficient partitioning and un-partitioning of shared resources among logical processors sharing physical resources. MONITOR sets up an effective address range that is monitored for memory stores; MWAIT places the processor in an optimized state (this may vary between different implementations) until a write to the monitored address range occurs [ia304]. This simulates a condition wait or spin lock on the hardware level, but notification to the waiting thread does not require operating system intervention. As a result, this mechanism has a smaller latency than other primitives for the process to 'wake-up'. The process will 'wake-up' when any interrupt is received by the processor, or the memory has been written to. It does not place the processor into a state as dormant as the HALT instruction, but it does release the processor resources. If both contexts are executing the MWAIT instruction, then the processor will go into a low power state.

The Intel Software Reference manual describes the usage of the MONITOR and MWAIT instructions. Figure 3.2 depicts the basic design. Intel specifies that the MONITOR instruction

```
long monitorspin_kernel(long slot, long oldValue, unsigned long reps)
{
      local_irq_disable();
      do {
               __monitor(&MemoryValues[slot],0,0);
               local_irq_enable();
               __mwait(0,0);
               local_irq_disable();
      } while ((MemoryValues[slot] == oldValue) && mem && reps);
      local_irq_enable();
      return MemoryValues[slot];
 }
```

Figure 3.2: MWAIT/MONITOR Kernel Internal Memory Design

must be executed with interrupts disabled, and the MWAIT instruction with interrupts enabled [ia304]. The monitor instruction arms the Pentium 4 Prescott monitor hardware with an address range to monitor for write operations. Writes in the memory range will trigger a 'wake up' from the mwait suspend state. The monitor/mwait instructions are in a do..while loop, because the mwait instruction can complete execution upon a memory write or a processor interrupt. Therefore, the current value of the memory address must be checked against a saved original value to determine if a memory write has occurred or just an interrupt. If no write occurred, then another MONITOR and MWAIT call on the same memory range must occur. These checks are made with processor interrupts disabled to prevent race conditions.

In the first generation Prescott Pentium 4 processor, the MWAIT instruction requires ring-0 privileges and cannot be executed from user space. We added a system call to the kernel in order to access these instructions, but that left us with the dilemma of either having to access user memory every time we needed to check if our condition variable had changed (Figure 3.3), or use a kernel

```
long monitorspin_user(long* memoryID, long oldValue, long type,unsigned long reps)
{
      long buf = 0;
      local_irq_disable();
      do {
               __monitor((void*)memoryID,0,0);
               local_irq_enable();
               __mwait(0,0);
               local_irq_disable();
               access_process_vm(CurrentTask,buf,sizeof(buf);
      } while ((buf == oldValue) && mem && reps);
      local_irq_enable();
      return buf;
 }
```

Figure 3.3: MWAIT/MONITOR System Call User Space Memory Design

memory space variable that is accessed from user space through system calls (Figure 3.2). The system call implementation, refered to as MWAIT-Kernel-Space, requires both threads to execute system calls in order to interact with the MWAIT and MONITOR instruction. The waiting thread executes the MONITOR/MWAIT loop, as in Figure 3.2, while the setter thread may optionally call a function to set an internal kernel variable which the waiting thread is monitoring. We refer to the MWAIT implementation that monitors variables in user space memory as MWAIT-Call-User-Space.

In future implementations of the instruction, Intel plans to make this instruction available at ring-3 execution privilege level. In the meantime, we discovered the Kernel Mode Linux project developed by Toshiyuki Maeda[Mae02]. This project allows user processes to execute with kernel (ring-0) privileges, but retains all the memory and privilege protection afforded to 'regular' user processes. This allows us to execute any ring-0-only instruction from our user applications and it allows us to eliminate much of the overhead associated with normal system calls by allowing direct execution of them. We refer to this version as MWAIT-Call-User-Space. We offer these results alongside MWAIT-Kernel-Space, MWAIT-SysCall-User-Space and our other synchronization primitives. as a reference. Performance of our applications using Kernel Mode Linux (KML) can give us a closer upper-bound to performance by bypassing normal user-process restrictions, but, as in the case of MWAIT-Call-User-Space, we see some effects of using these instructions and their respective overhead in standard execution environments. Figure 3.4 shows the ideal simplified design allowed when executing MWAIT directly from the user-space process. We can now check the resulting values directly. One caveat exists: Because user-space memory is virtual memory and, thus, can be paged to disk, there is a chance that page-faults may occur when checking the values. Pinning the desired pages into memory would provide the best performance, but the likelihood of getting the page fault is less with MWAIT-SysCall-User-Space than with MWAIT-Call-User-Space, because there is no context-switch overhead and that memory was likely recently initialized and already in the cache.

```
long monitorspin(long* memoryID, long oldValue)
{
      local_irq_disable();
      do {
               __monitor((void*)memoryID,0,0);
               local_irq_enable();
               __mwait(0,0);
               local_irq_disable();
      } while (*memoryID == oldValue);
      local_irq_enable();
      return buf;
 }
```

Figure 3.4: MWAIT/MONITOR Call User-Space Memory Design

There is another flaw with the current implementation of the MWAIT instruction. It lacks feedback to the programmer as to why the MWAIT instruction completed execution. If the instruction was to somehow distinguish between completion due to an interrupt or to that of a memory write, then the user performing the write would not have to make an expensive system call in the MWAIT-Kernel-Space instance nor pass the old value of the memory address being monitored.

HLT Instruction

The HLT (HALT) instruction is not generally used in synchronization primitives, but for the operating system idle loop. We present it here to contrast its behavior with that of the MWAIT instruction, which was also designed for use in operating system idle loops. Executing the HLT instruction on a idle logical processor puts the targeted processor in a non-execution state. This requires another processor (when posting work for the halted logical processor) to wake up the halted processor using an inter-processor interrupt. The posting and servicing of such an interrupt introduces a delay in the servicing of new work requests. An enabled interrupt (including NMI and SMI), a debug exception, the BINIT# signal, the INIT# signal, or the RESET# signal will resume execution. If an interrupt (including NMI) is used to resume execution after a HLT instruction, the saved instruction pointer (CS:EIP) points to the instruction following the HLT instruction [ia304]. We do not use the HLT instruction in our experiments, because its purpose is for placing the processor in a halted state until the next regular interrupt occurs. Such large granularity is not useful for synchronization primitives, especially spin-locks.

NOP/FNOP Spin-lock

The non-operation instruction NOP and its floating-point counterpart FNOP are the most basic instructions used in spin-locks. The NOP and FNOP instructions are executed in the processor in one cycle, but do not make any state changes to the processor context. It is a very quick operation that can be executed in the integer and the floating-point functional units to keep the processor busy.

PAUSE Spin-lock

Intel created the PAUSE instruction to solve the problem of the NOP-spin-lock memory mis-prediction problem[Cor04]. The PAUSE instruction causes the processor to stop issuing instructions in a particular context for a single pipeline length. This frees up pressure on the CPU functional units, particularly the integer unit, and also relieves the branch prediction unit.

3.1.1 Blocking Synchronization Primitives

In this text, we refer to the Pthread versions of mutexes, condition variables and semaphores[NBF96]. Blocking semaphores may block the executing task and place it on a wait queue until some precondition is met.

Mutex

The most basic concurrency control method is the mutex. The mutex provides mutual-exclusion access to critical sections. Mutexes are made possible by special hardware instructions like test-and-set and compare-and-swap. These instructions allow, without any race conditions present, to execute a comparison and a memory modification in one instruction. The basic mutex creates a lock, which is held by one task. If any other task would like to acquire that lock, then it is placed on the wait queue by the operating system thread scheduler. These tasks will block indefinitely, or until the task holding the lock releases it. Any tasks waiting for the lock will then wake up and a race condition exists to who will get the lock. Whoever executes the special hardware instruction first gets the lock.

Semaphore

A mutex is the most basic kind of semaphore, a binary semaphore. Semaphores are basic data structures that have non-negative values and are operated on by two basic primitives: signal() and wait(). A semaphore is initialized to a positive non-zero value which is decremented upon every wait() call. If a wait() is called on a semaphore with value 0, then that calling task and all future tasks calling wait() on the same semaphore will be suspended until enough calls to signal are made to make the semaphore value greater than zero. Semaphores are good at allowing a limited number of tasks access to a resource.

Condition Variable

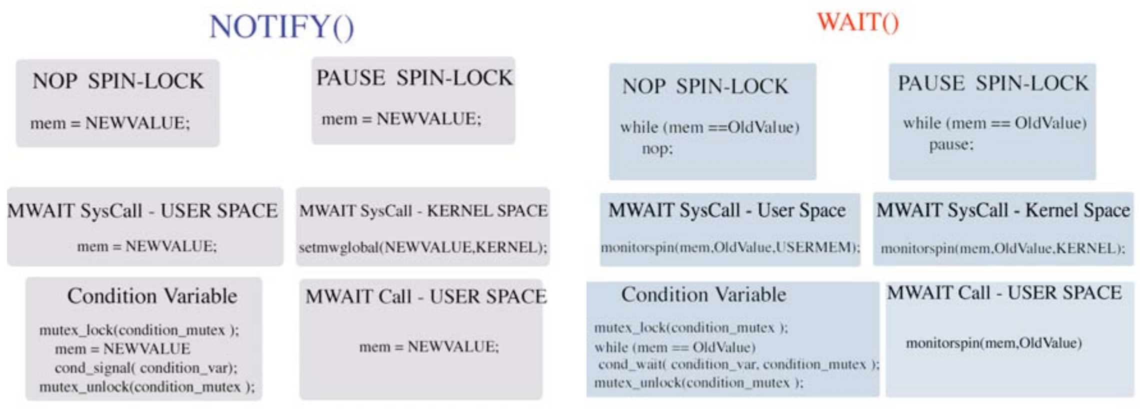
Condition variables are an improvement on the basic mutex, because they allow for fine control over access to critical sections. A condition variable assumes the executing task has control of a pertinent lock, and the task is awaiting some condition to be met, it can release the lock and suspend itself onto a wait queue. At the same time, it releases the lock, without calling an unlock(), and another process may then access the critical section. A controlling thread may then signal this condition variable announcing to an sleeping tasks that they may re-evaluate the conditions which they were waiting on. Once the signaling task lets go of the condition variable's lock, one of the waiting tasks atomically receives the lock and continues execution. This makes placing locks around a loop safe, because you can selectively give up the lock in the middle of the loop, and then re-evaluate safely after receiving the lock again. The overhead of having to reschedule the tasks is low, but it is still an order of magnitude higher than using a basic spin-lock. Condition variables are good for tasks that may wait a variable amount of time, or if many tasks are waiting.

In general, we know that the Monitor/Mwait and Pause instructions are meant to replace tight spin-wait loops, and halt is meant to place the processor into a non-execution state. We refer to our PAUSE spin-locks as PAUSE-Spin-Lock; NOP spin-locks as NOP-Spin-locks; MWAIT spin-locks as MWAIT-Call-User-Space and MWAIT-Kernel-Space depending on whether we pass MONITOR a user-space memory address or kernel-space memory address through a system call; MWAIT-SysCall-User-Space if we are executing MWAIT and monitor in user-space executing in kernel mode; and Condition variable primitive as Pthread-condition-variable. We do not categorize the MWAIT instruction as a blocking instruction, because it must be executed in a loop and no queuing occurs. The blocking of the task until a write occurs is an intended side-effect, because the instruction itself does not complete execution until that occurs. We chose to use condition variables over semaphores, because they were more flexible to use.

3.2 Evaluation

We designed experiments to measure a) the tangible latency of the different synchronization methods described above and b) the effect each synchronization method has on primary thread performance.

3.2.1 Design



```
mem = NEWVALUE;
```

```
mem = NEWVALUE;
```

```
mem = NEWVALUE;
```

```
setmwglobal(NEWVALUE,KERNEL);
```

```
mutex_lock(condition_mutex );
   mem = NEWVALUE
   cond_signal( condition_var);
mutex_unlock(condition_mutex );
```

```
mem = NEWVALUE;
```

```
while (mem ==OldValue)
    nop;
```

```
while (mem == OldValue)
    pause;
```

```
monitorspin(mem,OldValue,USERMEM);
```

```
monitorspin(mem,OldValue,KERNEL);
```

```
mutex_lock(condition_mutex );
while (mem == OldValue)
 cond_wait( condition_var, condition_mutex );
mutex_unlock(condition_mutex );
```

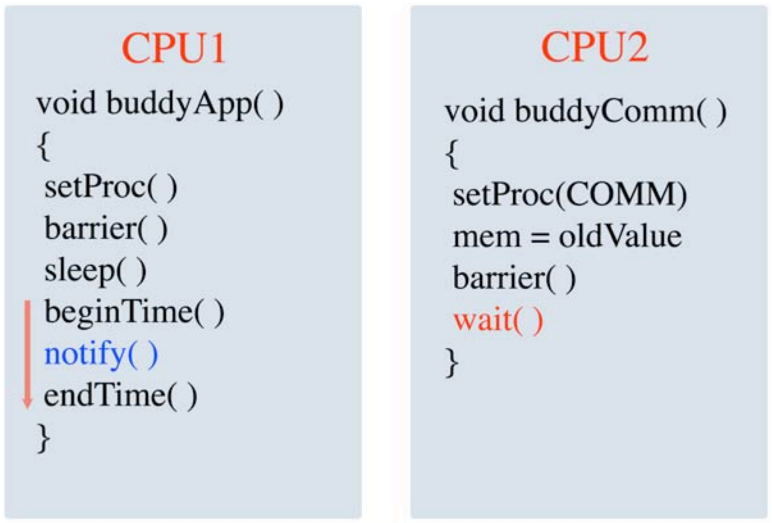
```
monitorspin(mem,OldValue)
```

(a) Notify Function Possibilities (b) Wait Function Possibilities

Figure 3.5: Synchronization Primitive Notify and Wait()

It is important to understand the impact of sharing the same processor between multiple threads. The spin-locks tested (Figure 3.5(a) and Figure 3.5(b)) show the design of the primitives executing some benign operation while waiting on an indicator variable to continue. Condition Variables, NOP spin-locks, Pause spin-locks and MWAIT spin-locks implement the non-operation differently. Condition variables place the waiting task on a wait queue; NOP spin-locks execute a 'nop' instruction that does nothing in the processor but uses a cycle of time within a functional unit. The PAUSE instruction instructs the processor context executing it not to issue new instructions until the PAUSE has retired from the processor pipeline (after about 20 to 30 stages for the Pentium 4 Prescott). MWAIT instructs the processor to go into low-power mode until a write is executed on a particular memory address or an interrupt occurs. Condition variables require the thread to deliver a signal to the other task and may involve system interaction. These varying techniques impose different amounts of overhead on the processor's resources, which may affect the performance of other tasks running on the same processor. We study three aspects to measure their performance: 1) The latency with which the waiting thread notices a change of state; 2) The call overhead of making the notification of change of state known to the other contexts; 3) The impact on the processor and resources that the waiting primitive has on the performance of other tasks on the same processor.

We are looking for the best balance between latency and performance. These tests are organized to reflect our ideal application model where one thread is engaged in work while another paired buddy thread is waiting to be notified to begin communicating. Figure 3.5(a) and Figure 3.5(b) show how each of our test synchronization primitives is implemented, and Figures 3.6, 3.7 and 3.8 show the test setups. We discuss each of the kernel benchmarks, analyze each synchronization primitive individually and then conclude.



```
void buddyApp( )
{
 setProc( )
 barrier( )
 sleep( )
 beginTime( )
 notify( )
 endTime( )
}
```

```
void buddyComm( )
{
 setProc(COMM)
 mem = oldValue
 barrier( )
 wait( )
}
```

Figure 3.6: Synchronization Primitive Call Overhead

Synchronization Primitive Notify() Call Overhead: In this test, we want to assess the impact of the calling thread's ability to maximize computation time when notifying the waiting thread. This notification should minimize the overhead communication to facilitate more overlap with computation. While the primitive-in-question is waiting for notification to continue, we measure the time to only notify the waiting thread. Figure 3.6 shows our test kernel. The arrow indicates the sections of code timed in the call overhead test. In our modified MPICH implementation, we want to impede the computation thread as little as possible.

Analysis

To maximize computation time for our computing threads and minimize overhead necessary to communicate with other threads. Table 3.1 presents our results for this test. PAUSE-Spin-lock, NOP-Spin-lock, MWAIT-Call-User-Space and MWAIT-SysCall-User-Space have the least overhead because they only assign a memory address. This takes about 300 to 500 nanoseconds (approximately

Synchronization Primitive	Synchronization Primitive Notify() Overhead (ns)	Synchronization Primitive Notify() Overhead (ns) (KML)	Percent Improvement by KML
MWAIT SysCall - User Space	449	330	36%
MWAIT SysCall - Kernel Space	3,407	925	368%
Pthread Condition Variable	5,239	1576	332%
PAUSE Spin Lock	527	317	66%
NOP Spin Lock	532	320	66%
MWAIT Call - User Space	N/A	390	N/A

Table 3.1: Synchronization Primitives Notify() Call Overhead (lower is better)

200 cycles), which is about the time it takes to fetch an instruction, schedule it, issue and execute the store and finally notify the other context of the store. Under KML, we see this memory overhead reduced slightly because of savings in servicing any interrupts that may occur in the middle of the store. MWAIT-Kernel-Space notify() must make a system call in order to adjust the memory. A system call takes approximately 1,200 nanoseconds to call and 1,200 nanoseconds to return from plus the memory write with some added logic overhead, which amounts to approximately the 3,407 nanoseconds that MWAIT-Kernel-Space requires. That accounts for the dramatic reduction in call overhead. It is still three times slower than MWAIT-Call-User-Space though. The Pthread-Condition-Variable primitive improves dramatically, too. Even though the Pthread library executes completely in user space, it still must notify other tasks through system IPC calls. Thus, it also incurs the system-call overhead that MWAIT-Kernel-Space does. It is clear that the overhead of making a system call has an order of magnitude impact in speed. Ideally, we would want our primitive to not require a system call.

Synchronization Primitive Latency: Synchronization Primitive Latency is the time from one thread signaling a change of state in the program to the time a waiting thread registers that change. Figure 3.7 shows our test kernel. The arrows highlight the sections of code timed in the experiment. Latency is very important for time-sensitive functions like sending and receiving data. This test is meant to model one aspect of our MPICH modification. In our MPICH modification, we split up the user and system portions of sending and receiving data, while in the original version of MPICH, the data is sent directly from the main computation thread. In our modified version of MPICH, the computation thread enqueues a request to send or receive data using a non-blocking communication function (MPI_Isend, MPI_Irecv) and then notifies the communication thread of that request. This communication thread is waiting for new user requests by monitoring an indicator

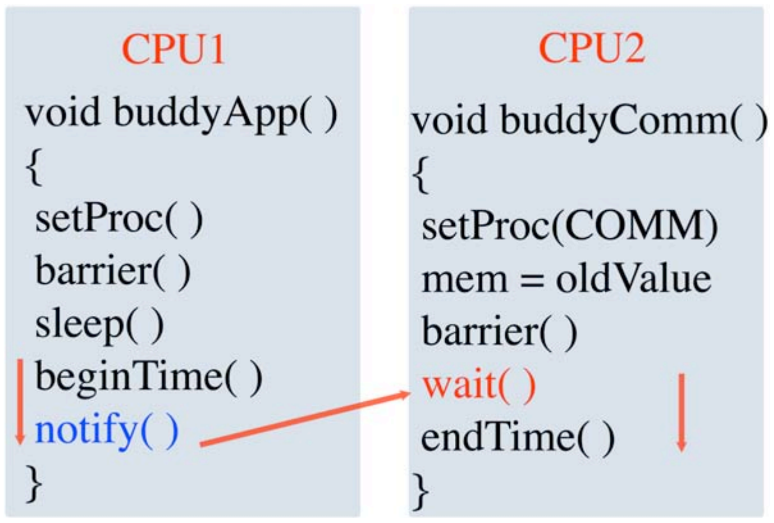


Figure 3.7: Synchronization Primitive Latency Test

variable. Monitoring of the indicator variable is performed in a loop that also checks for network activity. To give the processor a small period of complete control, the communication thread executes one of our synchronization primitives. The primitives should be quick and have low impact on the primary thread's performance. Latency is an important aspect in choosing the best primitive, but we must consider all the tests together in order to conclude which primitive is the best.

Analysis

Synchronization Primitive	Synchronization Primitive Latency (ns)	Synchronization Primitive Latency (ns) (KML)	Percent Change by KML
MWAIT SysCall - User Space	41,143	1316	32.26x
MWAIT SysCall - Kernel Space	2,909	869	4.35x
Pthread Condition Variable	10,757	3805	3.82x
PAUSE Spin Lock	143	135	1.59x
NOP Spin Lock	174	165	1.54x
MWAIT Call - User Space	N/A	424	N/A

Table 3.2: Synchronization Primitives Notification Latency Table (lower is better)

Our results in Table 3.2 show that Pause-Spin-lock and NOP-Spin-lock are the fastest to notify the waiting thread that a change of state has occurred in th KML and non-KML benchmarks.

This is because they operate completely in user-space memory. The two threads share the same address space, and bus communication between the processors is negligible. The overhead of the memory write dominates the latency overhead, so the system-call benefits would have little impact. It is interesting to note that the latency of the notify() for Pause-Spin-Lock and NOP-Spin-Lock is faster than the notify() call overhead shown in 3.1. Because the processes are sharing the same processor, the processor may notify the pending register load in the spin-lock that the value has changed right after executing the memory-store instruction on the indicator variable instead of waiting for write-back to the register table. MWAIT-Call-User-Space performs almost as fast as NOP and PAUSE spin-locks, but the overhead of putting the context to sleep and waking it up plus the latency for delivery of notification of a memory-store account for the extra nanoseconds delay over PAUSE and NOP. The most important observation is the improvement MWAIT-Call-User-Space has over MWAIT-Kernel-Space. We see that the latency is cut in half with the system-call overhead removed. MWAIT-Kernel-Space improves three-fold under KML. Under non-KML conditions, the waiting thread is engaged in a system call so that, at a minimum, it incurs the cost of exiting a system call. It also must possibly complete the MONITOR/MWAIT and interrupt disable and enable plus a comparison to verify that the value in memory has changed. Even with this overhead, MWAIT-Kernel-Space is at least 3 times faster than Pthread-condition-variable under KML or non-KML. A task waiting on a condition variable is placed in a wait queue that must be notified, rescheduled and re-assigned the lock, which partly involves system functions. Pthread-condition-variable is still four times faster than MWAIT-SysCall-User-Space. MWAIT-SysCall-User-Space has the worst performance primarily due page faults and accesses crossing over between kernel and user memory. Because MWAIT-SysCall-User-Space accesses user-space memory from kernel-space, it must add those pages to the kernel page table, access user memory, and it will frequently incur a page-fault. Using promotional paired scheduling and memory locking, it is possible to reduce this latency to approximately 15,000 nanoseconds under non-KML conditions. We see that KML reduces this to almost zero. Any interrupts needed to access user memory are handled without context switching and, thus, a 3,000% (32x) improvement over the non-KML version occurs under KML. The results demonstrate how important system overhead is to the performance of the functions. Any system calls degrades performance by an order of magnatude immediately for these micro-benchmarks. Our ideal primitive should not have to make system calls at all. PAUSE-Spin-lock and NOP-Spin-lock still look like the best primitives, and MWAIT-Call-User-Space is the second best.

Contention of Synchronization Primitives with CPU Bound Work: This test demonstrates the resource impact the synchronization primitives have on the performance of the computation thread. Figure 3.8 shows our test kernel. The arrow highlights the code timed in this experiment. The primary thread executes a fixed number of floating-point instructions (work) while the helper thread waits for the indicator variable to change. We use floating-point instructions to simulate the work because we want to emulate our ideal compute-communicate model. The faster

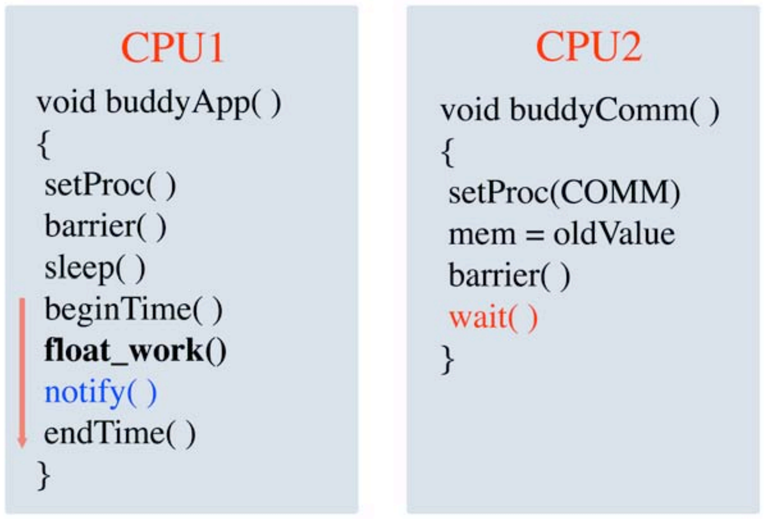


Figure 3.8: Synchronization Primitive CPU Impact:

Synchronization Primitive	Fixed Work Completion (Workload) (ns)	Fixed Work Completion (Workload) (ns)	Percent Improvement by KML
MWAIT SysCall - User Space	36,489	35,342	3.25%
MWAIT SysCall - Kernel Space	39,339	35,267	11.54%
Pthread Condition Variable	41,565	36,128	15.05%
PAUSE Spin Lock	61,284	58,865	4.11%
NOP Spin Lock	62,063	58,908	5.36%
MWAIT Call - User Space	N/A	34,641	N/A

Table 3.3: Time to complete the fixed amount of work with interfering (lower times are better)

the work completes, the less interference the helper thread imposes. This experiment also mirrors our modified MPICH implementation. The helper thread waits for the main thread to enqueue a function call while the main thread is performing computations. The helper thread should impede the main thread as little as possible while occupying as few resources as possible.

Analysis

In looking at the results under work-load conditions (Table 3.3), we observe that NOP and PAUSE spin-locks impose greater processor overheads than any of the other primitives. They are twice as slow at executing the work-load. Between MWAIT-Call-User-Space, MWAIT-SysCall-

User-Space , MWAIT-Kernel-Space and Pthread-Condition-Variable, there is very little difference in processor impact, but Pthread-Condition-Variable always has the highest time to completion most likely due to the extra logic required. Again, we see MWAIT-Call-User complete quicker than any other primitive indicating that it has the least processor impact and least overhead of all. The performance improvements of the KML versions are due to reductions in system overhead and quicker interrupt processing. Each of these constructs is aimed at minimizing impact on the processor's resources. MWAIT spin-locks act like a multi-cycle PAUSE instruction with the added benefit of re-joining the internal queues while the task waiting on the condition variable is taken out of the scheduling rotation and remains in a wait queue. NOP-Spin-Lock and PAUSE-Spin-Lock perform the worst because they are active constructs consuming resources every processor cycle thus reducing performance of any other threads executing concurrently. Even though the NOP and PAUSE instructions are executed by integer functional units and the master thread was executing floating point units, the slowdown could occur because of extra pressure on the issue and fetch units. Our ideal primitive should minimally affect the performance of other threads.

3.3 Conclusions

Even though NOP and PAUSE spin-locks have low notification overhead and small latencies, we see from our final workload test that NOP and PAUSE spin-locks consume the most CPU resources. The NOP and PAUSE spin-locks perform worse primarily due to interference with the execution of the primary communication thread. MWAIT-SysCall-User-Space has a high notification overhead combined with high latency in a non-KML environment, which makes it highly unsuitable. With KML and no system overhead, MWAIT-SysCall-User-Space has slightly higher latency and overhead than MWAIT-SysCall-Kernel and MWAIT-Call-User-Space. It is not a good choice for our ideal candidate because of too much overhead. Pthread-condition-variable provides solid middle-of-the-road performance. That is why it is the choice for most synchronization situations. It has good latency, low overhead and minimal processor impact. But MWAIT-Kernel-Space out performs Pthread-condition-variable. It has less call overhead and low impact on other concurrent threads. It is faster than MWAIT-SysCall-User-Space because it does not have to access user-space memory repeatedly, but suffers in notify() call overhead requiring a system call. KML improves MWAIT-SysCall-Kernel performance incrementally, but MWAIT-Call-User-Space is the best synchronization primitive to use. It has very low notification overhead, just slightly less than a plain memory store. In addition, it has very low notification latency. It is faster than any other MWAIT primitive and Pthread-condition-variable. Finally, it has the lowest overall impact on the processor allowing for the best performance of the master thread using MWAIT-Call-User-Space.

Chapter 4

Scheduler Assisted Promotional Buddy Threading

"Buddy threading" is a novel concept developed in this thesis that forces co-scheduling of processes to ensure that they run simultaneously at all times. Some processes have a symbiotic or dependent relationship requiring frequent data interaction. Due to the nature of their execution, these types of threads should to be scheduled simultaneously to improve parallelism, reduce IPC latency, reduce system overhead and possibly exploit certain shared resources such as processor cache.

To facilitate buddy threading, one process of the co-scheduled pair is designated as master control thread by the programmer. This control thread assigns itself threads (also called 'buddies') guaranteeing that they be co-scheduled together. The buddy threads will always run as long as the control thread runs, unless the buddy thread transitions off the run queue due to a sleep, forced preemption or other I/O related call. The control thread is not guaranteed to be executing always when the buddy thread is running. In addition to being run together, the buddy task runtime metrics were modified such that it would receive maximum sized time slices, no penalty and always first priority when the primary thread was also actively executing. In effect, the buddy monopolizes the particular context it is assigned to. This could lead to excessive unnecessary use of resources, so the buddy thread should endeavor to sleep or use the MWAIT, HALT or PAUSE instructions to relieve CPU resource contention when not engaged in computation. The Linux 2.6 operating system scheduler was modified to support this promotional buddy-threading behavior. This scheduling policy moves away from the classical round-robin weighted average scheduling policy built-in to the Linux Kernel. Instead, its intended behavior is to provide an unfair share of processor time to certain co-scheduled processes to facilitate lower Inter-Process Communication latency and quicker process execution. By unfairly promoting a particular process buddy, any tasks assigned to that thread should complete in less time than when being scheduled amongst other threads on the same processor. There is an additional benefit to threaded applications that may be sharing the CPU. Each co-scheduled application will run with the complete processor at its disposal.

The benefits of having two processes running simultaneously are two-fold: They are able to take advantage of temporal and spatial locality; and running simultaneously, there is effectively zero system overhead in inter-thread and inter-process communication. Buddy scheduling eliminates the latency of a buddy thread having to be co-scheduled with the master thread and minimizing IPC between the master thread and its buddy thread.

4.1 Technical Details

The current 2.6 Linux kernel does recognize and support SMT-enabled processors as described, but there is only limited scheduler support implemented to accommodate the special resource contentions of processes running on SMT-enabled processors. These modifications affine processes to stay on the same real and virtual processor. This works well to keep caches valid, but our need is to assign pairs of processes to always run co-scheduled on the same real processor and to remain scheduled together. Processes may not run co-scheduled if the control thread is servicing interrupts, sleeping in the I/O queue, or otherwise not executing actively on the processor.

Modifying the scheduler entailed adding two elements: a Linux task pointer to a processor buddy and a task pointer to the master thread. Control threads identify their corresponding buddy task by the buddy tasks assigned to themselves. A buddy task has a master thread if one is defined in its task structure. Our test program calls our custom system call to manipulate these variables and assigns tasks to be co-scheduled. Whenever the scheduler (running on CPU 0) schedules a task that has a buddy thread associated, it will migrate that buddy task to another context on the same processor, and, if necessary, interrupts that CPU in order to execute that CPU's scheduler. Once signaled, the buddy thread's CPU scheduler will always promote the buddy thread to the top of the run queue while the master task is executing. But if a control thread is not set or currently executing, then the scheduler will schedule the next task using the default scheduling algorithm. We rely on the scheduler load balancer to prevent non co-scheduled processes on buddy dominated CPUs from being starved.

4.2 Linux Kernel Buddy Threading Kernel Modification

4.2.1 Kernel Buddy System Specification

Kernel buddies are sets of tasks that are scheduled to run simultaneously at all times with a process' control thread. These tasks are co-scheduled together, and execute together as long as the control thread has not blocked or been descheduled. Kernel buddies are designed for threaded or multi-processed applications that would benefit from having certain threads always loaded together on separate processors or contexts. Our primary focus is on simultaneous multithreading applications wanting to benefit from cache localities and minimizing inter-thread/process communication. A running control thread designates another thread as a "buddy". This buddy thread has the following properties:

1. Whenever the control thread is running, the buddy thread should also be running. Preferably in a context of the same processor;
2. If a control thread is executing and the buddy thread is not executing, then the buddy thread should be scheduled next to run on the same processor. if possible) in a separate context;
3. Multiple threads can assign the same thread to be its buddy, but each thread may only have one buddy;
4. A buddy thread may not have a buddy of its own and have only as many buddies as available contexts on the system;
5. A buddy thread is not required to run, unless a minimum of one of its control threads is running;
6. A buddy is distinguished by its Process ID number (PID);
7. A buddy thread is defined by the control thread. If a thread has a valid task assigned to its control thread value, it is considered a buddy. Once that value is cleared or the control threads becomes invalid, the thread ceases to be a buddy.
8. Self-buddying is not allowed;

4.2.2 Implementation Details

We implemented the promotional buddy scheduler in the Linux 2.6 kernel scheduler framework. We added several system calls to access the promotional buddy scheduler, and we used several built-in system calls to manipulate schedulers and tasks. Table 4.1 outlines the main functions we used. Hyperpin() and getrpid() are the two system calls we specifically added.

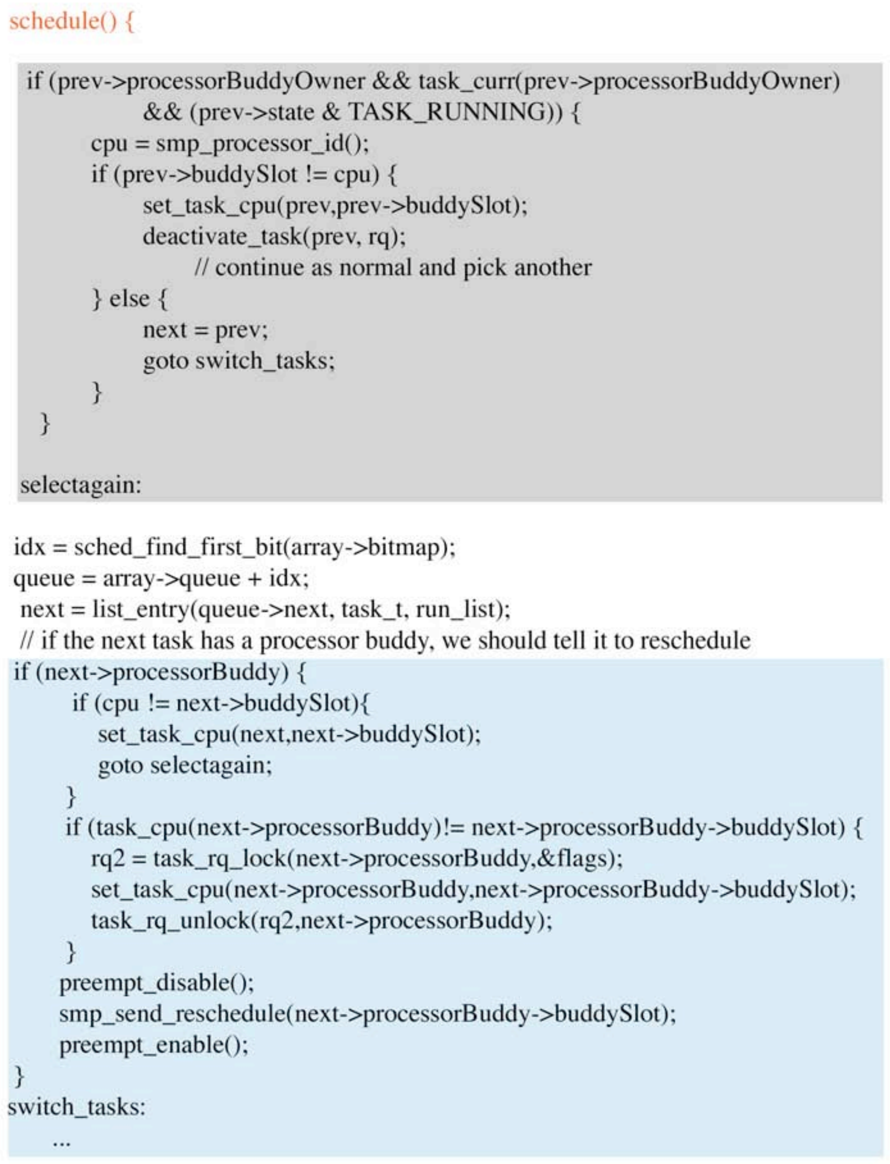
Formally, all threads are assigned a unique process id and treated as separate processes in the Linux kernel. An Application passes the system call 'hyperpin()' a pair of process id numbers (PID) and a base processor number. The first argument is the master thread PID, and the second argument passed is the buddy thread's PID. Each function can set for itself one master thread or one buddy thread, but threads with both enabled are treated as a buddy in the current implementation. This can easily be extended to multiple buddies and control threads, but at the expense of scheduling overhead. In addition to the buddy and control thread specification, a base processor id is passed to the kernel to which the control thread will be pinned to. If possible, the buddy thread

Table 4.1: Promotional Buddy Scheduler Code Functions

Function Name	**Purpose**
smp_processor_id()	Returns which processor the current process is running on
set_task_cpu()	Affines a task to a processor
deactivate_task()	Takes a scheduler off of the run queue and places it onto the wait queue
sched_find_first_bit()	Looks for Highest Priority Queue with available task
task_rq_lock()/task_rq_unlock()	Locks and unlocks the run-queue of the given task
preempt_disable()/preempt_enable()	Disables and enables interrupts
smp_send_reschedule()	Send a CPU interrupt to reschedule
hyperpin()	Designates and initiates scheduling of control and buddy tasks
getrpid()	Returns to the process the true process id number of the calling task

will be pinned to a context on the same physical processor in order to benefit from locality. Current processor identification techniques count logical processors sequentially from physical processor numbers. Future iterations will let threads roam across logical processors of a physical CPU.

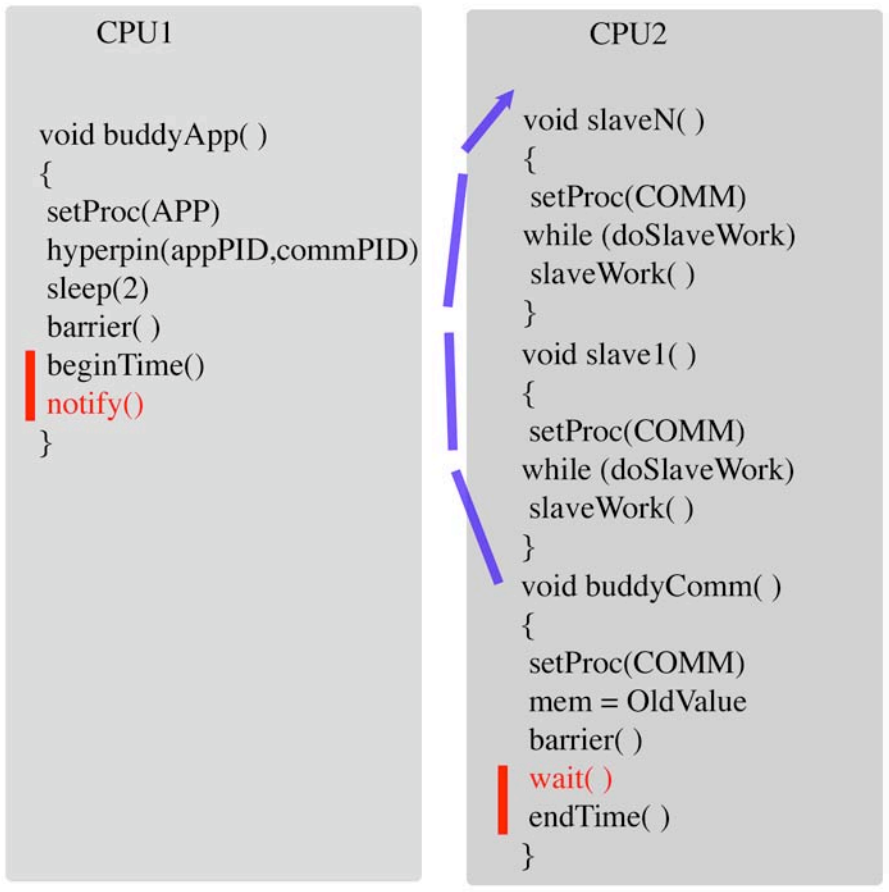
The PID is unique task identifier in Linux, but hidden by recent POSIX compliant Linux C libraries[DM03]. Processes have access to their thread-specific PID through a custom system call getrpid() which they pass when calling the hyperpin function. This sets variables inside the calling task structure and buddy task structure defining each other as buddies. It also assigns the processes to particular logical processors. When a master thread is scheduled, the buddy task is promoted to the head of the run queue and has its time slice and priority increased (by lowering the priority value) and if it is not executing, the scheduler is called on that logical processor. Every time the buddy thread time slice runs out, the scheduler first checks to verify that the master thread is still running. If so, then that buddy task is immediately re-scheduled, re-prioritized and continues execution. Keeping the process timeslice high and the priority high allows the process to minimize system overhead during the master thread's execution. Figure 4.1 shows a code-fragment of the scheduler code exhibiting the modifications. The Linux scheduler is a modified Priority Queue Scheduler with constant time task selection. Each process priority ranges from 1 to 140. Priorities above 100 are assigned to user tasks and those below 100 are for real-time and system tasks. The system will call schedule() whenever a task time-slice runs out, an interrupt occurs, a process blocks in some system call or a process voluntarily gives up the CPU. Each task has a pointer to its owner and that of a buddy thread to be paired with. Only one of these is used at any given time. Before choosing a new task, we check if the current task is a buddy task and whether its new state is on the run queue. If so and if the task is running on the correct processor, then any preempted task or I/O blocked task is not scheduled again. Instead, another task is chosen according to the default scheduling policy. Once a new task is chosen, it is checked for any buddies assigned to it. If there



```
schedule() {

 if (prev->processorBuddyOwner && task_curr(prev->processorBuddyOwner)
                && (prev->state & TASK_RUNNING)) {
         cpu = smp_processor_id();
         if (prev->buddySlot != cpu) {
                set_task_cpu(prev,prev->buddySlot);
                deactivate_task(prev, rq);
                       // continue as normal and pick another
         } else {
                next = prev;
                goto switch_tasks;
         }
  }

 selectagain:

 idx = sched_find_first_bit(array->bitmap);
 queue = array->queue + idx;
  next = list_entry(queue->next, task_t, run_list);
  // if the next task has a processor buddy, we should tell it to reschedule
 if (next->processorBuddy) {
         if (cpu != next->buddySlot){
             set_task_cpu(next,next->buddySlot);
             goto selectagain;
        }
        if (task_cpu(next->processorBuddy)!= next->processorBuddy->buddySlot) {
           rq2 = task_rq_lock(next->processorBuddy,&flags);
           set_task_cpu(next->processorBuddy,next->processorBuddy->buddySlot);
           task_rq_unlock(rq2,next->processorBuddy);
        }
       preempt_disable();
       smp_send_reschedule(next->processorBuddy->buddySlot);
       preempt_enable();
 }
switch_tasks:
      ...
```

Figure 4.1: Promotional buddy-scheduler Code

is a buddy assigned and that buddy is in the run queue, then that task is awoken by sending that task's CPU a reschedule() signal. If the newly selected task is not on the correct CPU, then it is rescheduled and new task is chosen.



```
void buddyApp( )
{
 setProc(APP)
 hyperpin(appPID,commPID)
 sleep(2)
 barrier( )
 beginTime()
 notify()
}
```

```
void slaveN( )
{
 setProc(COMM)
while (doSlaveWork)
 slaveWork( )
}
void slave1( )
{
 setProc(COMM)
while (doSlaveWork)
 slaveWork( )
}
void buddyComm( )
{
 setProc(COMM)
 mem = OldValue
 barrier( )
 wait( )
 endTime( )
}
```

Figure 4.2: Promotional Buddy Scheduling Evaluation Kernel

4.3 Buddy Threading Performance Results

We evaluated the promotional buddy scheduler using the kernel in Figure 4.2. The areas marked with vertical bars indicates the timed portions of code. This benchmark demonstrates the benefits of promotional buddy-scheduling on loaded systems. We start with a master control thread and a buddy thread that are bound to their respective contexts, and then add zero to five CPU-bound background interference tasks that are bound to the buddy thread's context. The buddy threads must compete with a varying number of interference threads all contending for access to the CPU. The interference tasks are executing the same work load that the buddy task executes. This experiment simulates our modified MPICH environment sending data. The task's data or resource requirements will conflict with that of another interfering task. We evaluate how well the promotional scheduler co-schedules the buddy task with the master task and the effects of the co-scheduling on the latency.

For comparison with already user-available performance-enhancing scheduling techniques, we completed the same latency test by setting the buddy task's nice level to -20, which is the highest priority a user can assign himself. Evaluating the results of promotional-scheduling against that of regular scheduling (Figure 4.3), we see that promotional-scheduling out-performs regular scheduling and increased priority scheduling in all cases except for the zero background task case, which we discuss later. Figure 4.3 refers to a non-KML implementation. The KML kernel results reflected those of Figure 4.3 and are not presented here. We see from Figure 4.3 that promotional

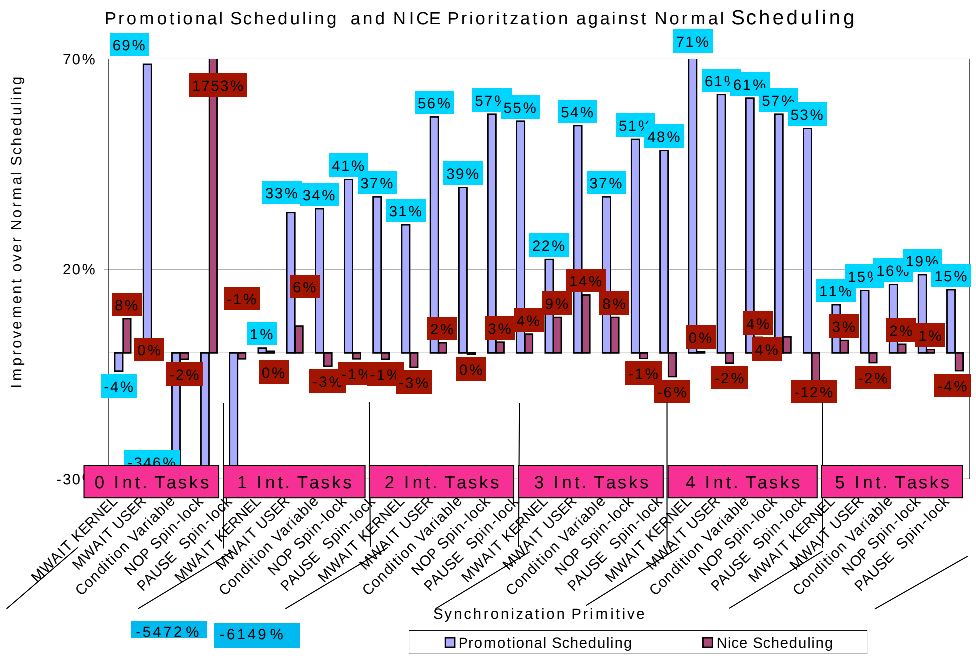


Figure 4.3: Synchronization Primitive Latency vs Number of Background Tasks

buddy-scheduling is almost always beneficial. The greatest benefit occurs when only one other task attempts to run alongside the co-scheduled tasks. This third background interference task is effectively locked out of the system by the master control task and its buddy task. In practice, the spare background threads may get rescheduled onto the master's processor and preempt the master thread. The promotional buddy scheduler does not provide behavior as in a two executing task test, because the the master task must be preempted to service interrupts and other system tasks. The interference tasks can then be scheduled to run. When the control thread is scheduled again to run, the background task will be interrupted to allow the buddy task to execute, but this will require an interrupt to be serviced and an smp_send_reschedule() to be issued. This is one reason the efficiency of promotional buddy-scheduling may degrade. It is designed not to lock out other tasks from running on the system, but to give certain tasks priority access to all the hardware available. Table 4.2 displays the zero background interference task data. We see that with zero background

Synchronization Primitive	Normal Scheduling (ns)	Promotional Buddy Scheduling (ns)	Normal Scheduling (ns) (KML)	Promotional Buddy Scheduling and Percent improvement (ns) (KML)
MWAIT syscall User Space	41,143	12,650 (325%)	1316	22,018 (1,674%)
MWAIT from Kernel Space	2,909	3,100 (-7%)	869	8,308 (-956%)
Pthread Condition Variable	10,757	43,843 (-416%)	3805	22,000 (-578%)
PAUSE Spin Lock	143	10,626 (-7,430%)	150	7,627 (-5,084%)
NOP Spin Lock	174	10,629 (-6,109%)	180	7,502 (-4,168%)
MWAIT Call User Space	N/A	424	N/A	7,763

Table 4.2: Zero Background Interference Tasks with Promotional Thread Scheduling

interference tasks, all synchronization methods are penalized except MWAIT user-memory. The NOP and PAUSE spin-locks perform the worst relative to their normal-scheduling times, but in all other tests the PAUSE spin-lock performs exceptionally well. We are not sure why most of the synchronization methods perform worse for the zero background interference task case, but we suspect that this is due to the scheduler penalizing the threads for consuming too much of the CPU, or because MWAIT-SysCall-User-Space it causes many interruptions and reschedules that allow for quicker servicing of interrupts.

The MWAIT-SysCall-User-Space method performs three times better with promotional buddy scheduling because the extra system interrupts involved in servicing the user-to-kernel memory copies get quicker service. We saw a similar improvement when testing MWAIT-SysCall-User-Space latency in conjunction with promotional thread scheduling. MWAIT-SysCall-User-memory improved 325% while the other synchronization primitives performed poorer than without promotional thread scheduling (in the zero background interference task case) while the same MWAIT-SysCall-User-Space test performs worse in the KML test.

There is quite a large improvement in performance with promotional buddy-scheduling after the addition of one task. We see a relative linear improvement for every additional interference task added. Finally, with five background interference tasks, the performance improvement decreases dramatically over that of the four background task case. At this point, the processor may be saturated. The exact reasons are left for future work. MWAIT-Kernel-memory has the lowest performance increase because it dominates the CPU by blocking interrupts and thereby preventing the system from interrupting the processor. As more background interference threads are added, the additional benefit decreases because the other threads are scheduled when the main thread is running. The performance benefit degrades with additional tasks, presumably because the other tasks are allowed to run while the buddy task is engaged in I/O or on the wait queue. NOP and PAUSE do well because they do not require operating system assistance.

The investigation into the causes of the poor zero background interference task performance are subject to future work. In our modified MPICH experiments, we are dealing with only two threads on the processor at any time. Thus, based on these results, we are not going to test promotional threading in our MPICH evaluation benchmark experiments. In addition, we are looking into the possibility of improving promotional scheduling by adjusting the task's time-slice upon reschedule to avoid the tasks from being penalized for excessive CPU usage.

Each of the data points is an average of 10,000 synchronization events. The magnitude of improvement between zero background tasks and one background task is three orders of magnitude. The zero-interference-task case has a range of only 40 microseconds while the one-background task test has a range of over 100 microseconds.

4.4 Promotional Buddy Scheduling and the Standard Kernel Thread Facilities

The promotional buddy scheduler has two main functions: It affines a master control task and its buddies to logical contexts of the same processor, and it insures that all buddy tasks of a master control process are given priority scheduling over all other tasks. Processor affinity and priority adjustment are user-accessible functions provided by most modern operating system task schedulers, but most of these schedulers are written as round-robin fair-use schedulers. Tasks are obligated to give up performance for the sake of fairness. Real-time schedulers must schedule tasks on time in order to assure completion before a deadline. None of these schedulers take into account inter-task requirements, whereby one task's performance may be dependent on another task's performance. In our model, we have a master task that has one or more buddy tasks. These buddy tasks are promoted by the scheduler to always run when the master control task runs. The master control task requires that these processes be instantly available in order to benefit from temporal cache locality, minimum system overhead and to take advantage of the parallelism provided by SMT processors. The promotional buddy scheduler recognizes these benefits and co-schedules the tasks to take advantage of the benefits. It is possible to use the promotional buddy scheduler with regular processors, while the added benefits of cache sharing would be lost, the benefit of not having to wait for your buddy task to be scheduled would still be realized.

4.5 Buddy Scheduling for MPICH

To take advantage of the buddy scheduling concept, buddy threads have to be on the critical path of a program's execution. In the case of our modified MPICH, a helper thread handles non-blocking send and receive calls and services any network communication. The main computation thread enqueues these requests into the request queue while the helper thread periodically checks for work. Latency in completion of network communication is critical in many computational codes due to lack of available work. In this case, the helper thread is a buddy of the computation thread. The helper thread is always scheduled alongside the computation thread in order to minimize task switch time.

In addition, the helper thread benefits from locality since data to be communicated usually resides in the cache at communication time. A more complicated model could also be used, such as sharing a set of contexts between a single communication thread with multiple computation threads. The computation threads would compete for a context, and the buddy would always be the communication thread. In this model, we are assuming that running two computation threads with identical workloads on the same processor will likely overextend the processor's resources.

It is possible to examine a multi-communication multi-computation thread model. Given enough processor contexts, such a model would allow for multiple computation threads (exceeding the number of available contexts) to be executing on some subset of contexts with access to several communication threads that are always co-scheduled alongside of them. Future IBM and Intel processors will have as many as 4 contexts on a CMP-SMT chip. Sun is also planning an 8 core 4 contexts per core processor allowing for a larger degree of parallelism and promotional thread buddy-scheduling. In the future, such experiments may be viable to stress-test the overlapping communication-computation model.

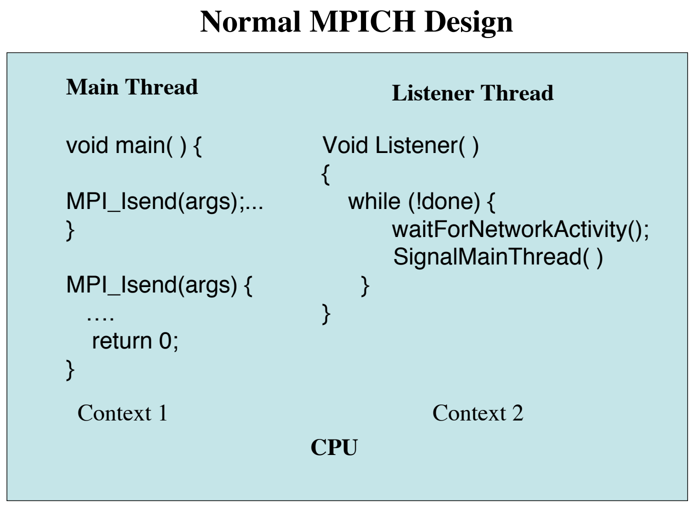
Chapter 5

MPICH MPI Communication Library

In order to take advantage of the parallelism in the SMT architecture, we modified the existing MPICH MPI parallel-communication Library from Argonne National Labs. This library was chosen because of its simplicity and portability. The design of the library has a higher user-level MPI API and a lower-level application device interface (ADI) that provides interoperability for many network interface devices and MPICH. Our test systems are Pentium 4 Prescott SMT-enabled processors running MPICH MPI applications on Linux kernel 2.6. Each instance of an MPI application will have a communication thread associated with it. This communication thread is paired with a computation thread executing on the main processor. As the MPICH library initializes, it spawns a buddy thread that, after initialization, performs all asynchronous communication functions including receiving unexpected messages. The Linux kernel was modified to provide this process association, such that when the main process is scheduled to run on the processor, the buddy task is loaded concurrently on the second processor.

5.1 Asynchronous/Synchronous and Blocking/Non-Blocking Functions

The original unmodified MPICH library is primarily a synchronous serial library. Execution always continues serially, even when data is received. Figure 5.1 illustrates the overall design of the MPICH library. There is separate process spawned by the P4 communications library, referred to as the listener thread, that checks for unexpected connections from other nodes and for queuing unread data. Upon notification of data availability (read) or when establishing a new connection, the listener() process sends a signal to interrupt the running computation process to execute the



```
void main( ) {

MPI_Isend(args);...
}

MPI_Isend(args) {
   ….
   return 0;
}
```

```
Void Listener( )
{
   while (!done) {
         waitForNetworkActivity();
         SignalMainThread( )
      }
}
```

Figure 5.1: Original MPICH Code Structure - High Level Design

handler that makes new connections and reads data. In this case, unexpected connection notification and unfinished data read are truly asynchronous processes, but the handling of these interrupts is serial and blocking. In fact, all network communication is serial and blocks in the standard MPICH network handler.

The MPI standard defines several functions as blocking (MPI_Send and MPI_Recv) and others as non-blocking subroutines (MPI_Isend and MPI_Irecv). The MPI standard refines our previous non-blocking definition as "*a nonblocking send start call [that] initiates the send operation, but does not complete it. The send start call will return before the message was copied out of the send buffer. A separate send complete call is needed to complete the communication*"[MPI]. The MPICH library, using the Channel P4 communication layer, does not strictly implement this definition of non-blocking. This definition is defined as an expected guideline in function behavior for the programmer, but the implementing library can implement the definition more strictly as MPICH does. There is no true non-blocking function call in MPICH using Channel P4, because the library was originally a serial execution library using TCP/IP connection-oriented communication. Table 5.1 describes the different methods that MPICH uses to send messages: Short, Eager and Rendezvous. These methods are differentiated by whether they send the header and message together (Short), separately (Eager) or the receiver connects back to the sender for the message after a connection is made to the receiver for the message header (Rendezvous). Each of these methods can be blocking or non-blocking, but any non-blocking implementation tends to look like the rendezvous protocol. The unmodified MPICH defines messages smaller than 128,000 bytes should be sent using the short

Sending Mode	Description
Short	Communications smaller than long (128,000) bytes are passed to the underlying TCP/IP layer as one packet including the message description header.
Eager (long)	Packets larger than “short” bytes and less than or equal to “long” (128,000) bytes are sent as two parts: the header describing the message, followed by the whole message. This option is not used in the default nor modified MPICH.
Rendezvous (very long)	The header is sent to the destination. The Receiver connects back to the sender to start transferring the data.

Table 5.1: MPICH Channel Sending Modes

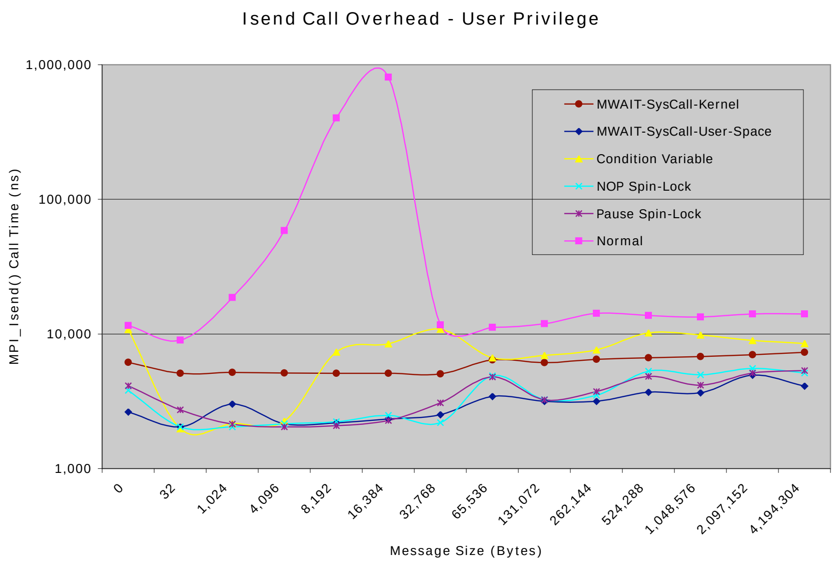
protocol. Short messages are always sent using a blocking send regardless of whether a blocking or non-blocking function was invoked. The underlying TCP/IP stack divides this message into packets and establishes a connection to the receiver using a blocking send. For messages larger than 128,000 bytes, non-blocking user functions must still connect to the receiver in order to send the header information of the message. The receiver then connects back to the sender to asynchronously read the packet. The MPICH library always makes a connection to the receiver, and, for small messages, the whole message is copied to the system and then sent to the receiver. These inconsistencies with the standard non-blocking definition and the limitations imposed by the TCP/IP protocol prevent true non-blocking functionality in the unmodified MPICH. In our modified MPICH library, we implemented the true non-blocking definition for non-blocking MPI routines.

5.2 MPICH Channel P4 Code Structure

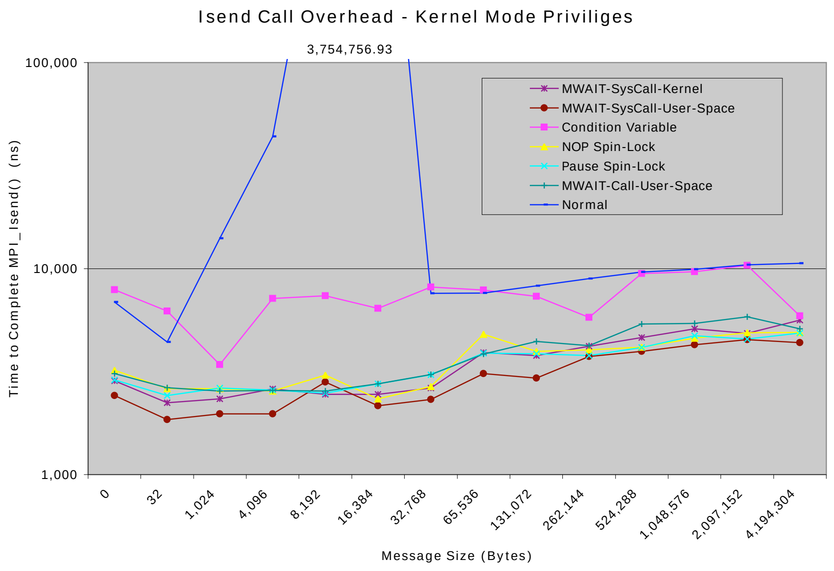
MPICH is currently implemented as a statically loaded library linked to the main MPI user program. The library invokes rsh or ssh to execute user programs on remote machines, and it supports a wide variety of underlying network integration and communication protocols. We use the P4 Communication library, which implements the Channel Interface, in our experiments using TCP/IP for message passing. The current code provides serial execution. If a user calls a blocking (MPI_Send/Eager MPI_Isend) or non-blocking call (MPI_Isend), the program executes the complete function until the system has copied the user buffer into system space. If the function call is blocking (MPI_Recv), then the caller task blocks in the underlying P4 read() call until data is received. To handle receiving data asynchronously and quickly, the underlying P4 layer spawns at startup a seperate listener process whose sole purpose is to wait in a select() system call on the network file descriptors and notify the MPICH library, through the system signal facility, that a new connection or message has arrived and needs to be handled. The user application is immediately preempted to begin reading the sockets or establish a new connection. Upon completion, the user program continues from where it left off. The user application cannot make progress on its own, even on non-blocking calls, unless it has created its own seperate thread for communication. We evaluated the efficiency of the non-blocking MPI function Isend() by accessing the CPU cycle-counter before executing the function and after it returns to the callee. Figure 5.2(a) and Figure 5.2(b) shows the time to execute and complete the MPI_Isend() transaction in a standard MPICH implementation and the modified MPICH with varying buffer sizes.

We move most of the overhead associated with Isend and Irecv into the communications thread, thereby freeing the master thread to continue computing. The standard MPICH, even though it MPI_Isend is non-blocking, must still assure that the user buffer has been copied to the system before continuing. Blocking Send and the Non-blocking Isend differ in whether the function waits for the data to be transfered to the receiver or not. In the modified MPICH implementation, we do not wait for the buffer to be copied to the system, nor do we wait for the receiver to acknowledge. We found that re-use of user buffers is infrequent, and in the cases where it does occur, it can be overcome through double-buffering.

The most obvious anomaly occurs at 16 KB and 32 KB for Normal MPICH in both KML and Normal execution modes. We narrowed down this spike to the Linux TCP/IP stack, and we suspect this to be a buffering effect, but we could not come up with conclusive answers. We do see that all versions of our modified MPICH avoid this pitfall, because they become true asynchronous non-blocking calls and the communication thread is assuming this overhead, thereby leaving the computation thread to do more calculations while awaiting completion of the non-blocking asynchronous send. It is interesting to note that the KML version of Isend() at 16 KB performs three times slower than the non-KML version. The system overhead may not be the problem and possibly something to do with the way KML implements page-faults. We leave the exact reason to future work. This anomaly gives our modified MPICH a great opportunity to get higher performance for small-size packets. It is, in fact, the area underneath the "Normal" MPICH Isend() Overhead curve and above any version of our modified MPICH that is the exploitable execution benefit. While the Isend() would be completing in the normal MPICH, the modified MPICH application can proceed in program execution. Pthread-Condition-Variable had the worst performance and the most erratic behavior. We see a very high overhead cost for Pthread-Condition-Variable probably due to the library and system requirements to implement the condition variables and deliver signals. Even with the system overhead marginalized, performance is only better for sizes that fall under the anomalous spike. While not executing in ring-0 mode, MWAIT-SysCall-Kernel is marginally better than Pthread-Condition-Variable. The high cost of the system-calls required for MWAIT-SysCall-Kernel penalize it as we see in the KML test, the MWAIT-SysCall-Kernel performs equally with the other primitives. MWAIT-SysCall-User-Space, NOP-Spin-Lock, PAUSE-Spin-Lock and MWAIT-Call-User-Space all perform very similarly with a plus or minus 1,000 floating-point opera-



(a) MPICH Isend() Call Overhead - Normal User Privileges



(b) MPICH Isend() Call Overhead - KML User Privileges

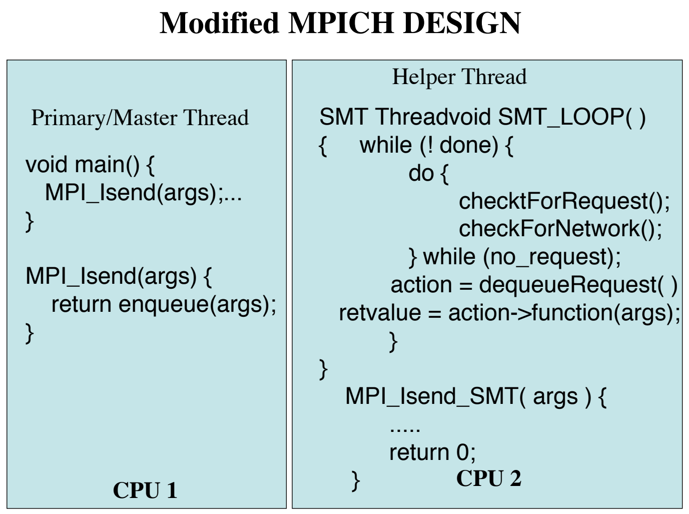
Figure 5.2: Isend Message Latency

tions (FLOPS). These tests reflect the results we achieved in our synchronization micro-benchmarks in Chapter 3.

5.3 Modified Code Structure

5.3.1 MPICH Re-Design

MPICH is a layered API library implementation with custom communication libraries provided by third parties to support different communication interfaces such as the Channel P4 library. The custom communication libraries are called directly from the MPI API call through the communication library ABI. These libraries are linked at configuration time. For maximum compatibility with different communications layers, modifications were made in the library at the API-ABI interface level. The user calls a shell function that enqueues the passed arguments into a global function call queue. Concurrently with the primary user thread, the assigned buddy thread (assigned during initialization) loops indefinitely waiting for functions to be called.



```
void main() {
  MPI_Isend(args);...
}

MPI_Isend(args) {
   return enqueue(args);
}
```

```
SMT Threadvoid SMT_LOOP( )
{     while (! done) {
            do {
                   checktForRequest();
                   checkForNetwork();
            } while (no_request);
          action = dequeueRequest( )
   retvalue = action->function(args);
          }
}
    MPI_Isend_SMT( args ) {
          .....
          return 0;
    }
```

Figure 5.3: MPICH Modification High Level Design

Code modifications to MPICH were designed to be as transparent to the application and underlying ADI device as possible, with the exception of asynchronous message reception. First, we began by creating a virtual API that mirrors the standard MPI API. For every MPI_ call, there was a complementary MPI_SMT_ call to handle it. The main modifications centered around the asynchronous calls: isend and irecv. This new communication thread, hereafter also refered to as buddy thread or helper thread, not only completes the asynchronous calls for the user thread, also called the 'main' or 'primary' thread, but the asynchronous communication. This buddy thread plays an important part in receiving data, because it supplants the original P4 listener process. This takes out two slow system interrupts in the code occuring from the use of signals and file descriptors for inter-process communication. Figure 5.3 describes the code modifications.

Whenever there is asynchronous communication between the nodes, instead of the main program having to stop computing to begin sending data, the helper thread assigned to the second context takes on the task. The main thread would then be free to continue computing, while the helper thread handles the overhead with the system. In addition, if the main thread wants to perform blocking calls, it will initiate these calls from the main thread, leaving the helper thread to continue receiving any unexpected data asynchronously. There is a caveat in that if the helper thread is performing an asynchronous operation, then unexpected messages may be delayed slightly. This can also affect the latency of Isend, because the helper thread is doing double duty.

5.4 Evaluation

5.4.1 Synthetic Benchmark Results

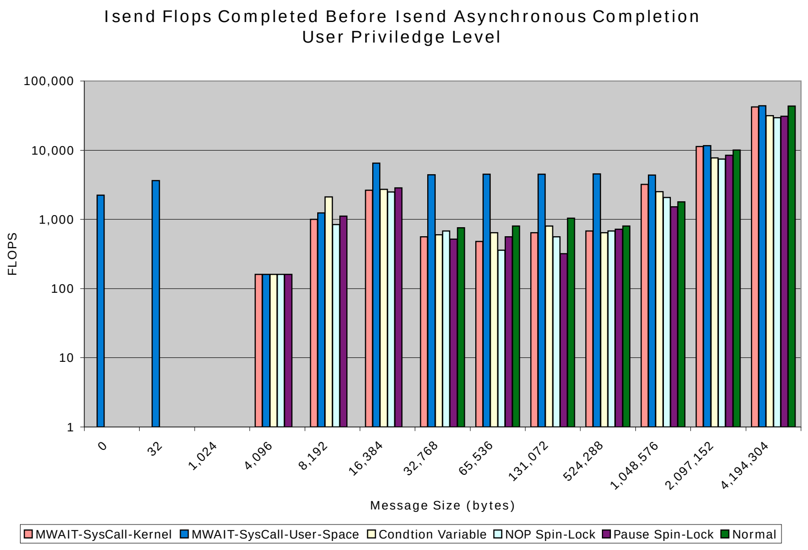


Figure 5.4: MPICH Isend overhead benefit measured in floating point operations not running in KML

To evaluate the modified MPICH performance, we created test kernels to capture the benefit of having a threaded asynchronous send call. Our experimental kernel is shown in Figure 5.6.

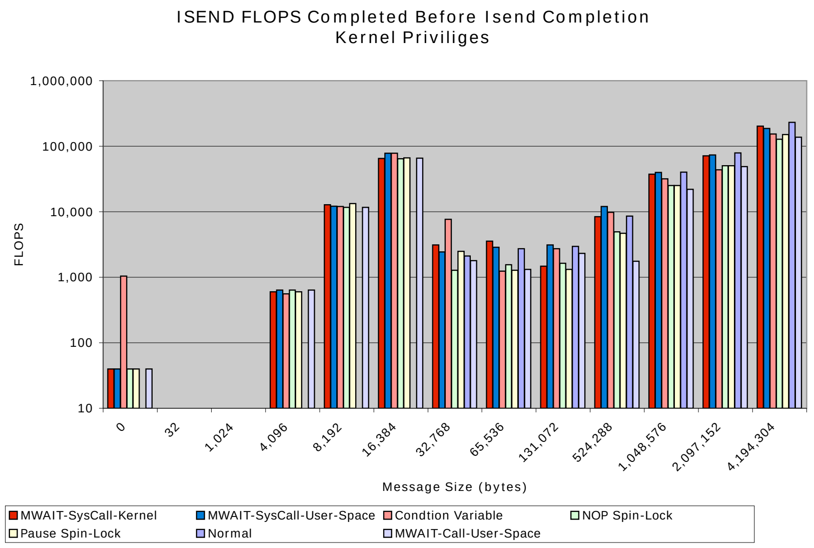


Figure 5.5: MPICH Isend overhead benefit measured in floating point operations with KML privileges

We tested each of the synchronization primitives against the standard base MPICH under privileged KML and regular execution mode to evaluate performance improvements. Our goal is to show that we can effectively utilize more of the CPU for computation using our buddy model than the standard serial method. We found that our threading model does not improve the network performance of the MPI library significantly because we do not affect the underlying P4 library's operation. Figure 5.4 and Figure 5.5 quantify the performance benefits shown in Figure 5.2. That potential described as the area under the Normal MPICH curve can be quantified in floating point operations executed while awaiting completion of the asynchronous call. These two bar graphs explicitly show the exploitable slack in the communication library. Looking at Figure 5.4 first, we see no benefits for using any of the synchronization primitives for less than 4 KB of data except with MWAIT-SysCall-User-Space. Messages are infrequently less than 32 bytes [VM02]. The most interesting messages sizes are those between 8 and 64 KB. This seems to be the area with highest latency for Normal MPICH, which we take advantage of. Normal MPICH does not have enough slack to perform computation simultaneously to messages possibly below 32 KB. Our modified MPICH, in contrast, can start taking advantage with messages as small as 4 KB. The number of computations Normal MPICH can asynchronously take advantage of is comparable to our NOP

```
Sender() {
  for msgSize = 0 to 2^20 bytes
       double c;
       long floats = 0;
       BeginTime()
       request = MPI_Isend(buffer,msgSize,receiver)
       while (!MPI_Probe(request)) {
           for num = 0 to 1000
                 c = c*c
           floats++
       }
       EndTime();
 }

 Receiver() {
    for msgSize = 0 to 2^20 bytes
        double c;
        long floats = 0;
        request = MPI_Recv(buffer,msgSize,sender);
 }
```

Figure 5.6: MPI_Isend() Test Kernel

and PAUSE spin-locks and condition variable primitives, but MWAIT-SysCall-User-Space leads with messages up to 4 MB with MWAIT-SysCall-Kernel close behind it. We believe the resource impact and overhead of NOP/PAUSE spin-locks and Pthread-Condition-Variables prevent the extra magnitude of calculations to be performed as message size increases. Performance evens out across the board as message size grows because memory becomes the limiting factor and Normal MPICH and our Modified MPICH utilize the low-overhead rendezvous messaging protocol of MPICH after 128 KB. As the system overhead is reduced by the KML environment, floating point operations increase for all test primitives. Figure 5.5 still reflects the anomalous behavior seen in the graphs of Figure 5.2, and we are able to further take advantage of it. Although we do see available floating point operations at 0 KB, it is an infrequent and unlikely case in real-world programming, so it is not interesting to study. We start seeing available slack again at 4 KB message sizes. The time to enqueue and synchronize with the helper thread is approximately equal to sending a message smaller than 4 KB. Under KML, the Normal MPICH performs extremely well after 32 KB. The lack of system overhead and quickness of the rendezvous protocol allows for performance on par with our modified MPICH and even better performance than our modified MPICH at very large message sizes. A final observation should be noted that MWAIT-Call-User-Space performed on par with other synchronization primitives in the floating-point test, but its advantage is the lower system overhead and faster latency, which we want to take advantage of.

Chapter 6

ASCI Purple Benchmarks

We ported five of the ASCI Purple benchmarks, IRS, SMG2000, sPHOT, SPPM and Sweep3d[VY02, VM02, Pur], to the Intel i386 Linux platform to test for real-world applications that can take advantage of the SMT processor. Table 6.1 summarizes each of the benchmarks. All of the benchmarks, except for Sweep3D, use non-blocking message passing to some extent to test our hypothesis.

6.1 Implicit Radiation Solver (IRS)

6.1.1 Description

Implicit radiation solver (IRS) [Car03] is an application to solve the radiation transport equations by the flux-limited diffusion approximation. The application uses an implicit matrix solution. IRS is written as a general diffusion equation solver, but the flux limiter imposes the speed of light as the maximum signal speed. The preconditioned conjugate gradient method (PCCG) is used to invert the matrix equation. Neither transformations, diagonal scaling nor the two-step Jacobi method are used to precondition the matrix. This PCCG method is known to not be completely scalable, but its limitations are accounted for in the benchmark test sets. The standard ISO-C code of the application uses a combination of MPI and/or OpenMP threads to scale.

The benchmark problem is a planar radiation wave diffusing through a regular rectangular mesh from one end to the other and out. The physical problem is a 10 x 10 x 10 cm^3 mesh with variable resolution. The mesh is normally divided into domains with one domain per processor. The number of spatial domains should not be less than the number of processors. Large amounts of communication occur only in transmission of domain surfaces, though the PCCG method does require two global reductions of four doubles for every matrix iteration. The benchmark time is made up of the global reduction speed plus the general surface communication speeds plus the computation

Table 6.1: Benchmark Summary

Application	Language	Problem	Primary MPI Functionality
IRS	C	A diffusion equation solver that solves the radiation transport equations by the flux-limited diffusion approximation	MPI_Irecv MPI_Reduce (47%) MPI_Waitany (32%) MPI_Allgather MPI_Allgatherv MPI_Barrier MPI_Bcast MPI_Gather MPI_Gatherv MPI_Isend MPI_Recv MPI_Send MPI_Ssend MPI_Waitall
SMG2000	C	Semicoarsening multigrid solver for linear systems	MPI_Waitall (60%) MPI_Irecv (35%) newline MPI_Isend (1%) MPI_Allreduce (.03%) MPI_Wait
sPHOT	F77	2-D photon transport code using Monte Carlo transport	MPI_Barrier MPI_Irecv MPI_Reduce MPI_Send MPI_Waitall
sPPM	F77	3-D gas dynamics problem on a uniform Cartesian mesh using a simplified version of the Piecewise Parabolic Method	MPI_Allreduce MPI_Isend MPI_Irecv MPI_Wait
Sweep3D	F77	Solver for the 3-D, time-independent, particle transport equation on an orthogonal mesh using a multidimensional wavefront algorithm	MPI_Allreduce MPI_Bcast MPI_Send MPI_Recv

speed. Comparisons should be measured in terms of speed per cell-iteration in one domain.

IRS_Speed = (execution time in microseconds) / (cells/domain * integral of iterations)

Messaging Profile and Execution Analysis

IRS is a highly optimized, compute-intensive application. MPI library calls account for only 7 percent of the execution time, and 47 percent of that time is spent in MPI_Allreduce. IRS

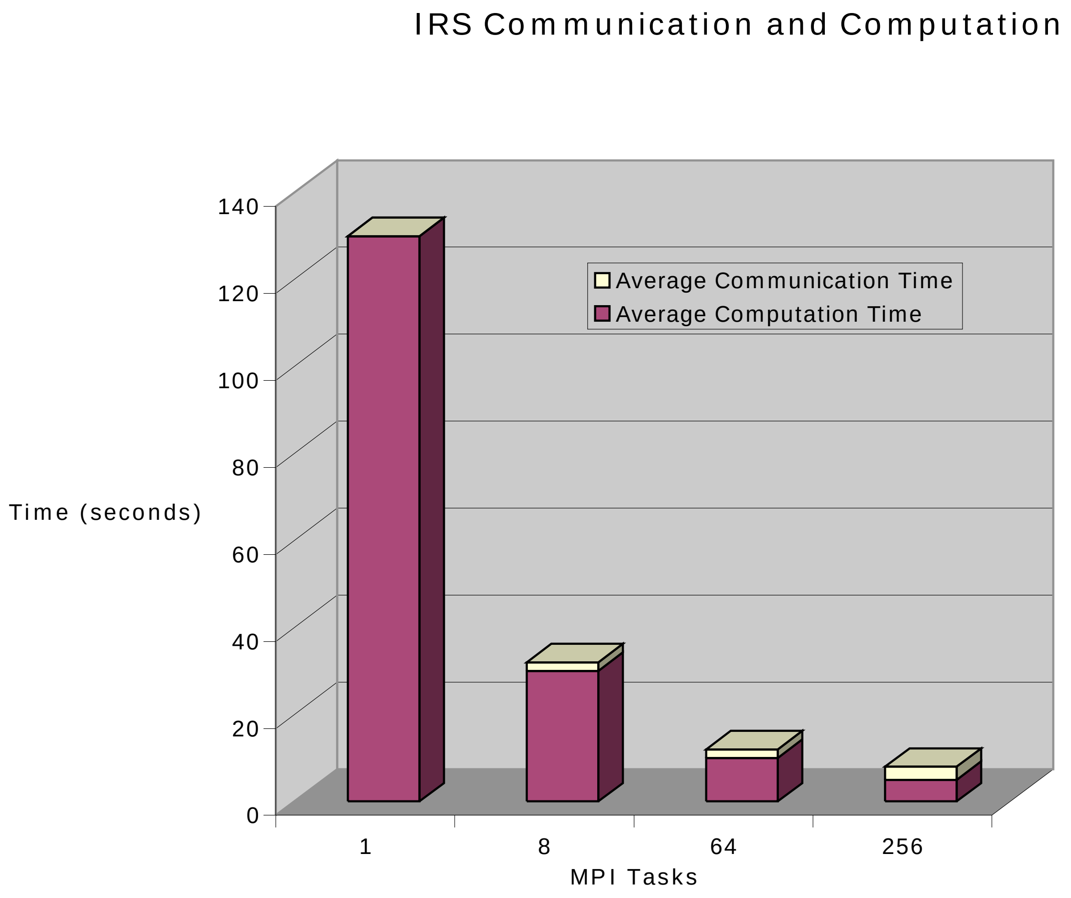


Figure 6.1: IRS Communcation (After [VY02])

is memory-bound, because 80 percent of the calculations require a memory access. This results in low utilization of the CPU and allowing for a greater density of threads per CPU. Execution time decreases linearly with the number of processes. The communication overhead decreases as the number of CPUs increases.

6.2 SMG2000

6.2.1 Description

SMG2000 is

> *parallel semicoarsening multigrid solver for the linear systems arising from finite difference, finite volume, or finite element discretizations of the diffusion equation on logically rectangular grids. The code solves both 2D and 3D problems with discretization stencils of up to 9-point in 2D and up to 27-point in 3D for the diffusion equation* $\nabla \cdot (D\nabla u) + \sigma u = f$ *. The driver provided with SMG2000 builds linear systems for the special case of the above equation, with Dirichlet boundary conditions of* $u = 0$*, where h is the mesh spacing in each direction. Standard finite differences are used to discretize the equations, yielding 5-point and 7-point stencils in 2D and 3D, respectively. To determine when the solver has converged, the driver currently uses the relative-residual stopping criteria. This solver can serve as a key component for achieving scalability in radiation diffusion simulations. Parallelism is achieved by data decomposition. The driver provided with SMG2000 achieves this decomposition by simply subdividing the grid into logical P x Q x R (in 3D) chunks of equal size.* [Car01]

The code is highly synchronous (blocking) with communication and computation patterns that exhibit surface-to-volume relationships causing efficiency to be determined by the size of the data blocks. SMG2000 is memory-access bounded. There are only one to two computations per memory access. The memory requirements for 3D problems are 54 times the local grid size times the size of a double plus some overhead. The overhead grows as the logarithm of the problem size.

6.2.2 Messaging Profile and Execution Analysis

As the number of processors increases, SMG2000 scales very poorly (Figure 6.2) due to an increasing amount of communication overhead relative to constant amount of computation time. 75 percent of the application time is spent in MPI library routines, and 98 percent of that MPI library time is spent in the non-blocking MPI_Isend, and MPI_Irecv calls and their respective message status routines. As the number of nodes increases, the floating-point operations decreases, but the fixed-point operations remains constant, while fixed-point operations dominate SMG2000 operations. Because of the frequent communications, SMG2000 is only able to execute about 1024 floating-point operations. On a 3 Ghz Pentium 4, this amounts to only 2 nanoseconds of computation for every communication. This small number of operations per communication per task requires low latency between the communication thread and the computation thread to prevent long backups in the message queue. The average message volume per task is 2.5 megabytes distributed to two-thirds of the other nodes in the network. With network communication latency between 60 microseconds and 20 milliseconds, the communication thread will have a queue of connections keeping it busy while SMG2000 computes. SMG2000 uses the integer function heavily, so we expect to see some conflict with the communication thread to occur.

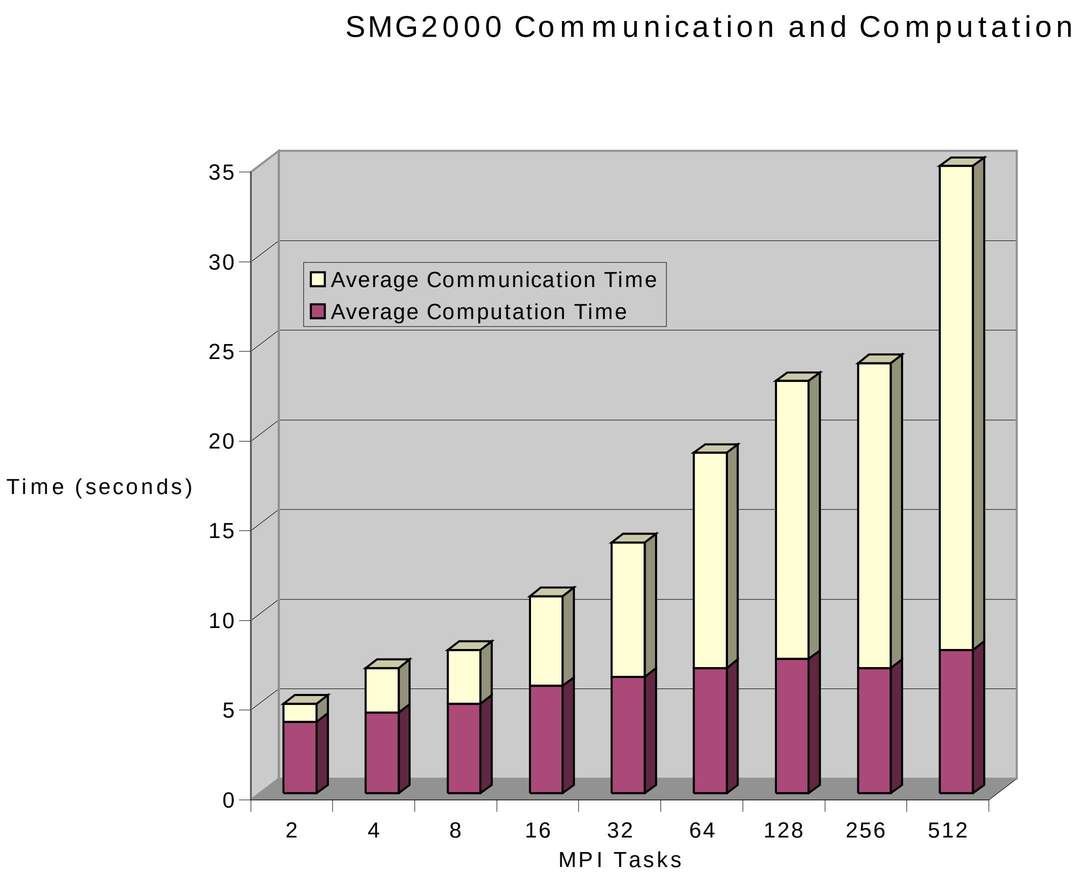


Figure 6.2: SMG2000 Communcation (After [VY02])

6.3 sPHOT

6.3.1 Description

sPHOT is

a 2D photon transport code. Photons are born in hot matter and tracked through a spherical domain that is cylindrically symmetric on a logically rectilinear, 2D mesh. Monte-Carlo transport solves the Boltzmann transport equation by directly mimicking the behavior of photons as they are born in hot matter, move through and scatter in different materials, are absorbed or escape from the problem domain. The logically rectilinear, 2D mesh in which particles are tracked, is internally generated. The mesh is small enough that a complete copy of the mesh will not only fit on each node in a parallel machine, but also fit into cache memory in most modern CPUs. Thus, this benchmark does not stress memory access.

Particles are born with an energy and direction that are determined by using random numbers to sample from appropriate distributions. Likewise, scattering and absorption

> *are also modeled by randomly sampling cross sections. The random number generator used is implemented in the code using integer arithmetic in a way that makes the resulting pseudo-random number sequence portably reproducible across different machines.*
>
> *The use of random numbers makes the code's output (edit variables) "noisy". This noise is a direct result of the discrete nature of the simulation. The level of the noise can be reduced by increasing the number of particles that are used in the simulation. Unfortunately, the level of noise in the answer decreases only very slowly with increasing computational effort. The noise is inversely proportional to the square root of the number of particles. If the noise is to be reduced to 1% of the value in a given simulation, it is necessary to run 10,000 times as many particles. Thus, high-quality (low-noise) simulations can become very computationally expensive. Parallelism is an obvious way to increase the number of particles.*[Bar01]

sPHOT is considered an embarrassingly parallel application. A copy of the 2D mesh is distributed to each computer, which generates its own random numbers and does its own particle tracking. Communication is limited to global scatter and gathers between the master and slave tasks to update global variables and distribute the meshes. This code utilizes OpenMP on every node and MPI for a message passing. Even if the program makes non-blocking send, the message size is so small that the Channel P4 communication layer will make a blocking send and complete the call.

Messaging Profile and Execution Analysis

sPHOT is so embarrassingly parallel that its communication overhead consists of 4 global data distribution operations regardless of scale. Increasing the size of the cluster decreases execution time exponentially. sPHOT spends at most 12 percent of the execution time in a barrier call. The average message size is only 4 kilobytes.

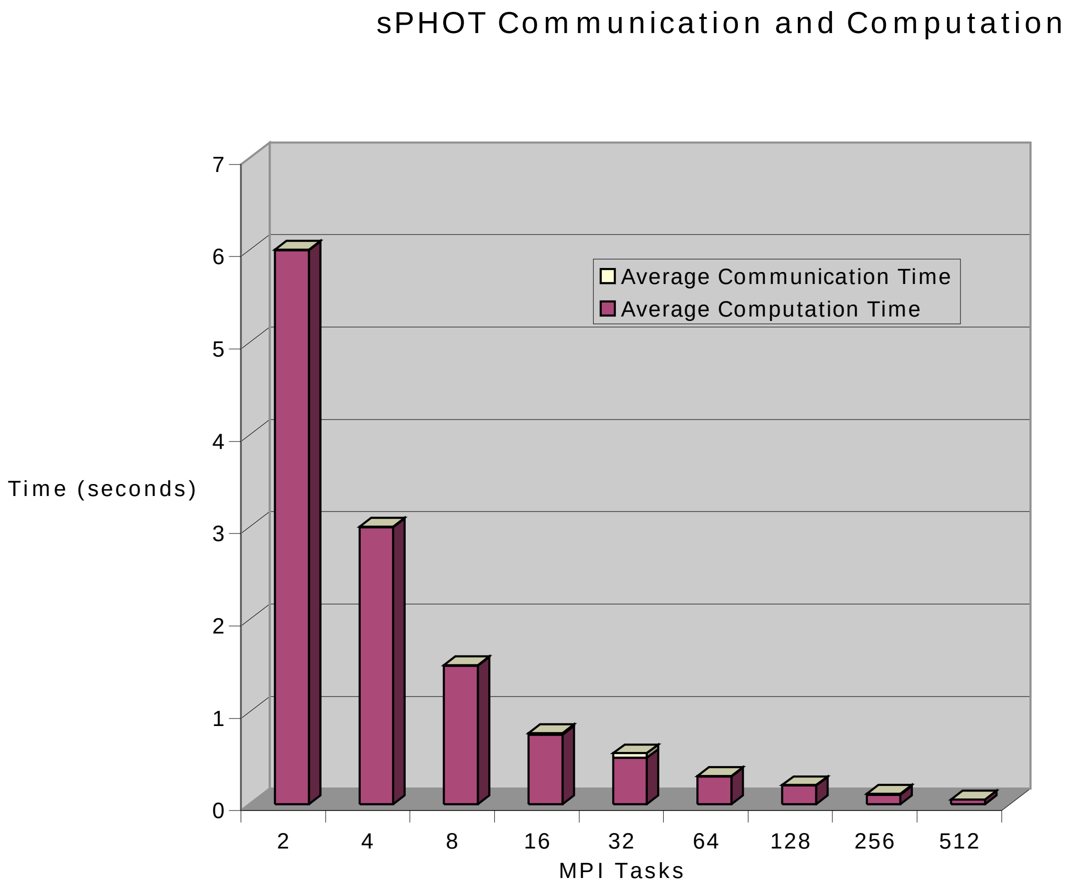


Figure 6.3: sPHOT Communication (After [VY02])

6.4 SPPM

6.4.1 Description

sPPM is

a benchmark solves a 3D gas dynamics problem on a uniform Cartesian mesh using a simplified version of the PPM (Piecewise Parabolic Method) codehence the "s" for simplified...The hydrodynamics algorithm involves a split scheme of X, Y, and Z Lagrangian and remap steps that are computed as three separate passes or sweeps through the mesh per timestep, each time sweeping in the appropriate direction with the appropriate operator. Each such sweep through the mesh requires approximately 680 FLOPs to update all of the state variables for each real mesh cell. Message passing is used to update ghost cells with data from neighboring domains three times per timestep and occurs just before each of the X, Y, and Z sweeps. Multiple threads are used to manipulate data and update pencils of cells in parallel.

The sPPM problem involves a shock propagating through a gas with a density discontinuity. The coordinates are -1:1 in x and y, and 0:zmax in z, where zmax depends on *the overall aspect ratio prescribed. A plane shock traveling up the +z axis encounters a density discontinuity, at which the gas becomes denser. The shock is carefully designed to be simple, though strong (about Mach 5). The gas initially has density 0.1 ahead of the shock; over 5dz at the discontinuity, it changes to 1.0.*[Eng02]

This code uses explicit and/or implicit threading techniques through POSIX threads and/or OpenMP for shared memory computation on nodes. Message passing is used for domain decomposition and to distribute data to each node.

Messaging Profile and Execution Analysis

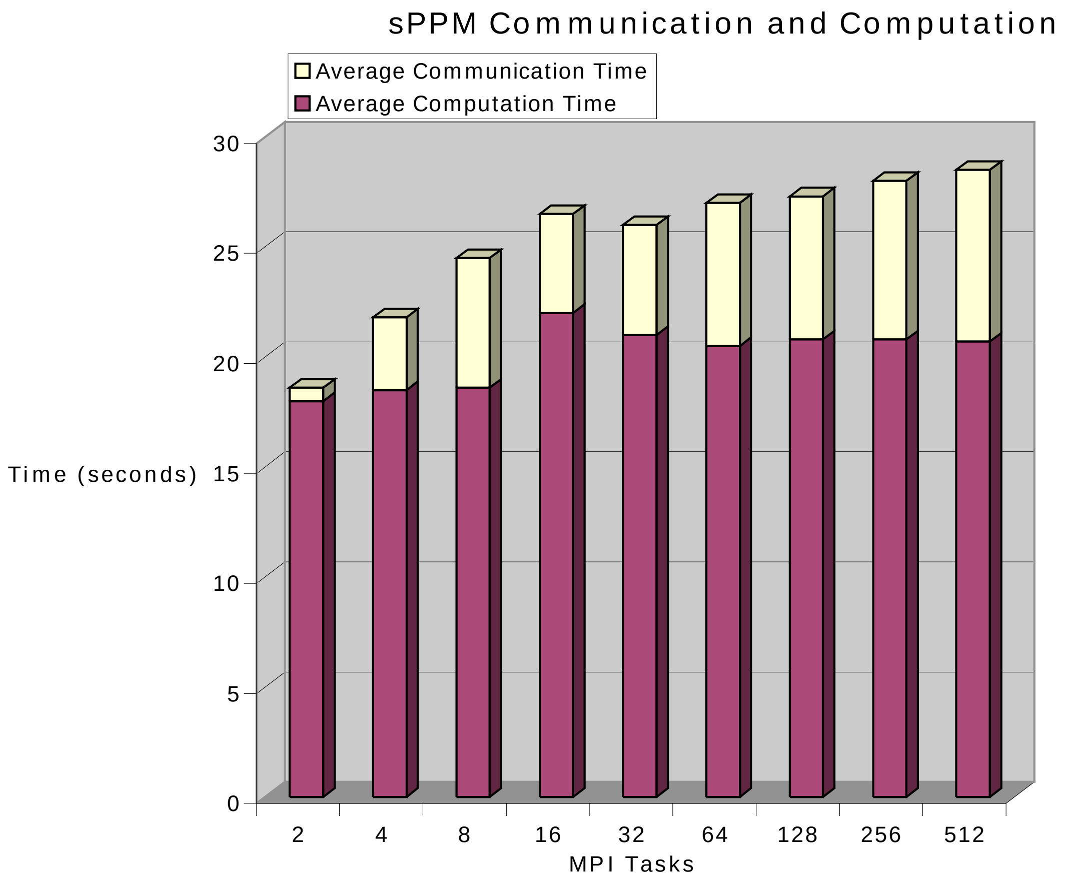


Figure 6.4: sPPM Communication (After [VY02])

sPPM is an application that can scale to thousands of processors with communication overhead of 4 percent (for 8 tasks or more) dominated by an MPI_Allreduce. Global MPI_Allreduces occur after each time-step, but communication between the neighbors occurs throughout the three sweeps. The average message size is 512 kilobytes with some messages as large as 1 megabyte. The large messages require rendezvous messaging entailing two separate messages. The modified MPICH should have an advantage here because the messaging is exclusively composed of MPI_Isends and MPI_Irecvs. As the number of nodes increases, the messaging overhead diminishes the added benefit of more processors. Each thread communicates on average 17 megabytes of data to the other nodes.

6.5 Sweep3d

6.5.1 Description

The benchmark code SWEEP3D

> *SWEEP3D solves a 1-group time-independent discrete ordinates (Sn) 3D cartesian (XYZ) geometry neutron transport problem. The XYZ geometry is represented by an IJK logically rectangular grid of cells. The angular dependence is handled by discrete angles with a spherical harmonics treatment for the scattering source. The solution involves two steps: the streaming operator is solved by sweeps for each angle and the scattering operator is solved iteratively. The two step solution coded in SWEEP3D is known as an inner iteration. A realistic Sn code would solve a multi-group problem, which in simple terms is nothing more than a group-ordered iterative solution on top of what SWEEP3D does. Groups are solved sequentially since there is strong coupling between groups due to downscattering. The multi-group iterations are known as outer iterations. Finally, a realistic ASCI Sn code would include time-dependence with thousands of time steps on top of what a multi-group code does.* [Car02]

As the number of groups increases, the number of inner iterations increases. Hence, the work and memory requirements substantially increase. Adding groups requires saving angle-dependent arrays for all grid points, angles and energy groups for old and new time steps. To solve each angle requires solving a series of 4 equations with 7 unknowns (boundary conditions complete the system of equations) resulting in the solution to the cell plus an input I, J or K for three other cells. This recurses until the solution is found.

The benchmark is currently coded with blocking sends and receives, but this 'embarrassingly' parallel code allows for independent calculation of all discrete angles. These can be converted to use non-blocking sends and receives to overlap communication of the next block of data with the computation of the current block. The benchmark would need to be altered to double buffer and stagger computations in order to maximize the benefit.

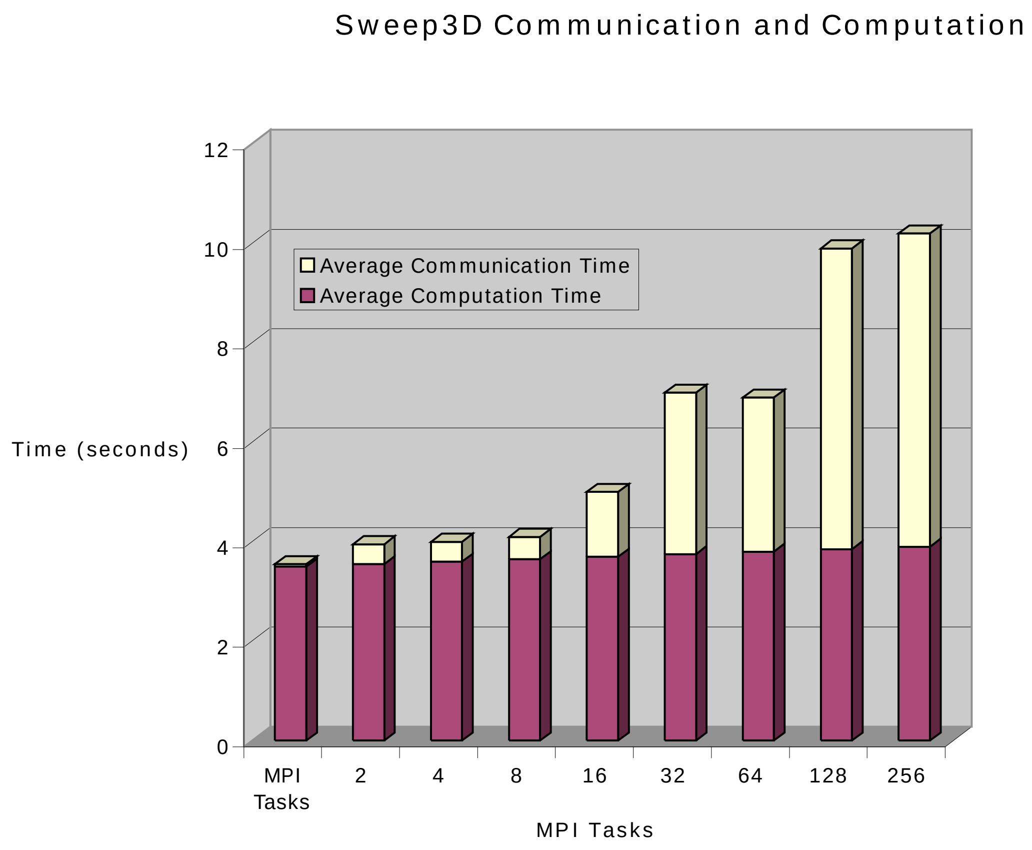


Figure 6.5: Sweep3D Communication (After [VY02])

Messaging Profile and Execution Analysis

Sweep3D is highly scalable code up to 64 nodes, but adding more nodes increases the computation overhead super-linearly. At 384 nodes, MPI_Send and MPI_Recv account for 63 percent of the execution time. MPI_Isend and MPI_Irecv makeup 98% of this time. It hits in the cache 99% of the time and spends the majority of time in one function that it iterates through every time-step.

Chapter 7

Benchmark Evaluations

7.1 Experiment Setup

The experiments were performed on a four-way cluster of computers connected via a gigabit ethernet switch. Each computer is equipped with an Intel Pentium 4 Prescott SMT processor from the Intel Corporation. Each processor has two logical contexts, which share 16 KB of L1 cache and 1 MB of L2 cache. Each node has 1 GB of DDR400 RAM memory, along with sufficient disk space. Two of the processors are clocked at 3 GHz, and the other two are clocked at 3.2 GHz. We believe the clock difference played little bias in these results as all experiments were performed on the same hardware. In all instances, benchmarks were scaled so they would not page to disk during the experiments.

We did not perform these experiments using our modified KML environment, nor utilize MWAIT-Call-User-Space. These results utilizes regular non-KML Linux operating system because KML is beyond the scope of this thesis. This is most unfortunate and we plan to incorporate this into future work. We make conclusions using the other available synchronization primitives.

Each machine runs Red Hat Fedora Core 2 Linux with Linux kernel 2.6.8. The kernel was compiled with default Red Hat kernel configuration enhanced by our personal extensions to enable promotional buddy scheduling and MWAIT instruction access. MPICH and the benchmarks were compiled using the Intel C and FORTRAN compiler version 8.1. All final results were run without debugging enabled, nor with X-windows active on any machine. In addition, each experimental data-point is an average of three runs.

7.2 Benchmark Evaluation

We present our benchmark results relative to the performance of "Normal" unmodified MPICH implementation. The Normal MPICH implementation is the standard MPICH library distributed by Argonne National Laboratory [Lab]. Higher Bars reflect improvement in time over the normal MPICH run-time. For each benchmark, we present results with absolute numbers and also normalized percentages against the standard MPICH library. Our benchmark experiments were made on the same four machines described above and in succession to keep the experimental setup as unbiased as possible. For experiments using Normal MPICH, SMT was turned off on the processor and the Fedora Core 2 single CPU kernel was used. SMT experiments always ran with SMT turned on and the SMP kernel.

We attempted to keep the problem size the same as we increased the number of nodes, but this was not possible in some cases, such as sPPM. To gain more insight into the results, we used Lawrence Livermore Laboratory MPIP and the Gnu Profiler tools when necessary for interpreting the data.

7.2.1 IRS

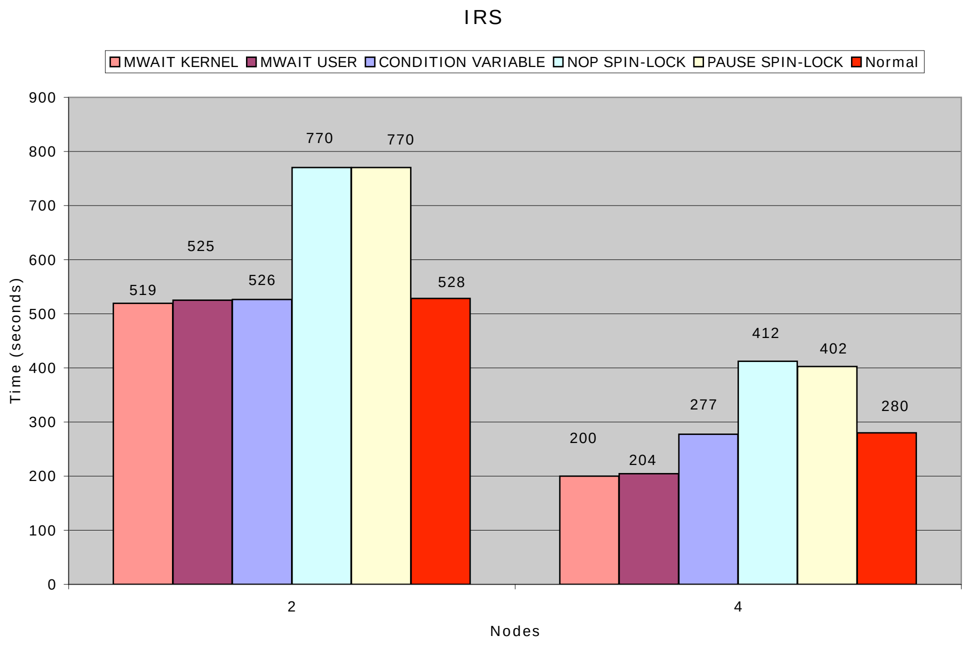


Figure 7.1: IRS Results (Time)

Our results are displayed in Figure 7.1 and Figure 7.2. They are displayed in two forms to provide reference to the meaning of the percentage value results. We received the best performance from the IRS benchmark. This benchmark is a long-running benchmark that scales very well and takes great advantage of the modified MPICH through the overlapping of work with asynchronous and non-blocking communication. The GNU Profiler was only able to attribute 87% of the CPU time to the application for our modified MPICH code (Condition Variable and MWAIT variations), while the Normal MPICH consumed 99% of the CPU time. The missing 13% was likely utilized in the MPI library during network communication.

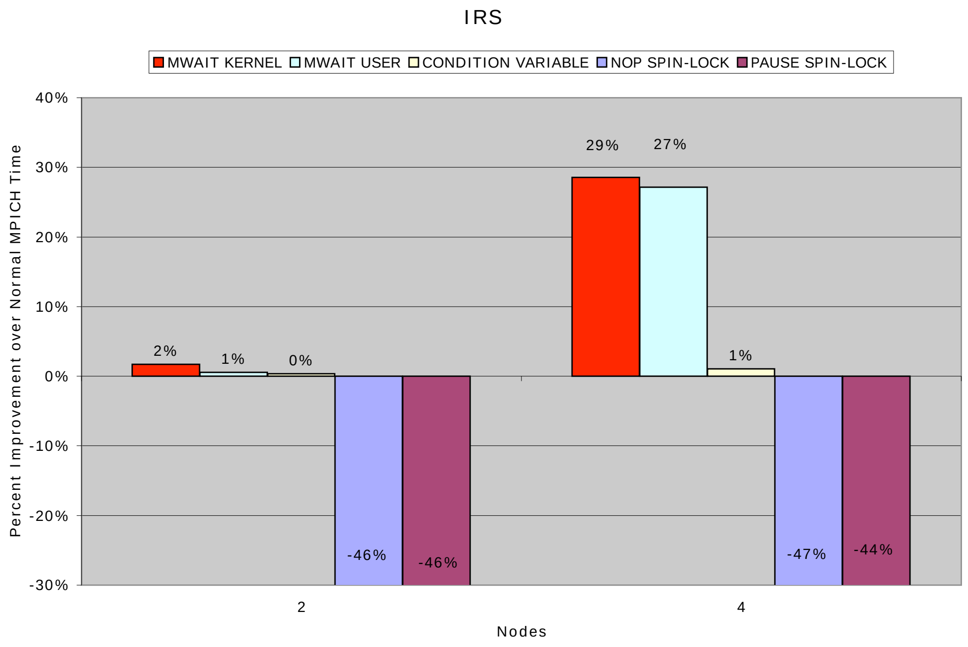


Figure 7.2: IRS Results - Percentage Improvement over "Normal"

On two nodes, IRS executes in 700 seconds, while on four nodes it executes in 200 seconds. This benchmark is very sensitive to interruptions and cache pollution. One function MatrixSolveCG consumes the most CPU time (40% combined), but the communication function rbndcom() saw the greatest savings. We see that rbndcom() requires 16 seconds in a two-node configuration and 9 seconds in a four-node configuration. The MatrixSolveCG function The rest of the savings were between 1 and 5 seconds per function distributed evenly across all functions. Considering that the majority of communication calls reside between every time-step, the overhead of being interrupted during these collective sections was hindering the benchmark's performance.

We see an equal performance increase for MWAIT-User-Space, MWAIT-Kernel-Space and Pthread-Condition-Variable MPICH because the communication overhead overlapped with the computation of the main application, even if it was null. This benchmark performs well for 4 nodes. With 2 nodes, we observer unchanged performance. We believe this is due to the added overhead of communication over the single node model. Also, the savings achieved are minimal with only two nodes because the duration of communication is minimal. In addition, the benchmark problem size was kept the same across the different sized nodes.

7.2.2 SMG 2000

```
  for (i = 0; i < num_sends; i++)
    {
      MPI_Isend (send_buffer, send_size, ((MPI_Datatype) 6),
                send_procs[i], 0, comm, &send_requests[i]);
    }
  }

 if (num_sends)
  {
    MPI_Waitall (num_sends, send_requests, send_status);
    ...
  }
```

Figure 7.3: SMG2000 Kernel

Our results are displayed in Figure 7.5 and Figure 7.4. They are displayed in two forms to provide reference to the meaning of the percentage value results. SMG2000 takes an average of 20 seconds to execute. It is an application spending 60% to 70% of its time waiting for non-blocking asynchronous calls to complete. The primary communication function is hyper_StructCoarsen(), which coordinates data with neighboring processes. This function dominates in the presence of communication. The NOP-Spin-lock and PAUSE-Spin-lock primitives are faster than either of the MWAIT variations or Pthread-Condition-Variable in executing MPI Calls, but they slow down and interfere with the application because there is CPU contention.

The main kernel in SMG2000 is an Isend/WaitAll combination illustrated in Figure 7.3. This code performs zero computations during point-to-point and collective message passing. 98% of

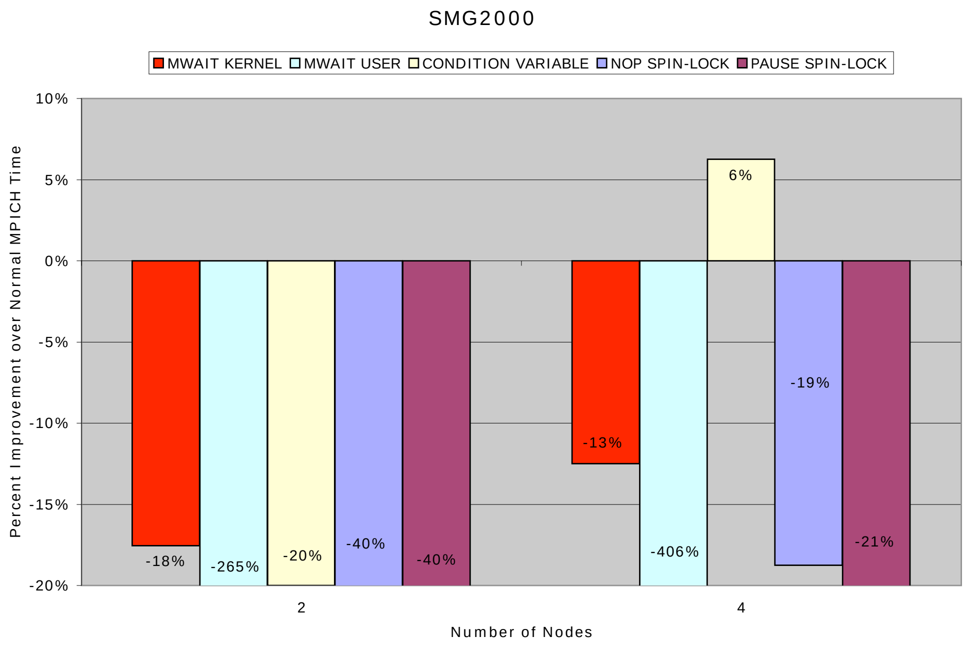


Figure 7.4: SMG2000 Results - Percentage Improvement over “Normal”

all the packets communicated are 2 KB in size. As we increase the number of nodes, we see that Normal MPICH beats all modified versions except for Pthread-Condition-Variable in the 4-node configuration. The overhead of queuing and dequeuing the accumulated Isend() calls with only 2 KB packets makes Normal MPICH faster in this respect. We see that this amounts to about 4 seconds in general. MWAIT-User-Space, in this case, shows its main weakness. Being in a such a tight call with so many Isends constantly sending data, the application must constantly suffer the kernel-to-user memory overhead. MWAIT-USER SMG2000 spends more time in Irecv() than in WaitAll(). In the 4-node case, Condition Variable MPICH beats NORMAL by one second and MWAIT Kernel by 4 seconds because it has the very low latency and system impact while remaining in User space. The MWAIT-KERNEL MPICH suffers from repeated system calls. SMG2000 problem size remained the same for all experiments.

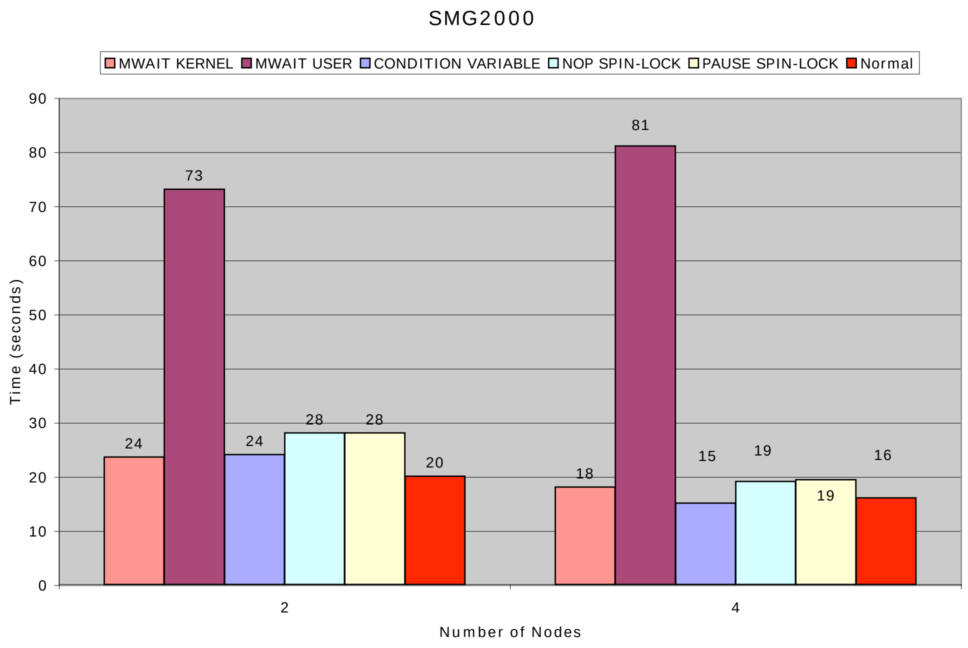


Figure 7.5: SMG2000 Results (Time)

7.2.3 sPHOT

Our results are displayed in Figure 7.6 and Figure 7.7. They are displayed in two forms to provide reference to the meaning of the percentage value results.

sPHOT is a highly CPU-bound benchmark that predictably showed little or no improvement. The benchmark runs for 15 seconds on average and has only 4 communication calls. Performance is flat for two nodes, because there is no communication involved and similarly for the two node experiment. In the four-node experiment, we see a 7% performance drop in Pthread-Condition-Variable MPICH, while MWAIT-USER-Space, MWAIT-KERNEL-Space and Normal MPICH show no difference. Again, NOP and PAUSE Spin-Lock versions degrade performance because of CPU resource conflicts. The beneficial effect of the MWAIT instructions on uniting the CPU internel queues are visible with this benchmark. Because of the paucity of communication and the similar profiles, the MWAIT-User/Kernel-Space and Normal MPICH benefit from complete use of the whole CPU. We start to see this effect with four CPUs and a mixture of communication and computation. Though the difference in time is only one second, that one second makes a big difference to an

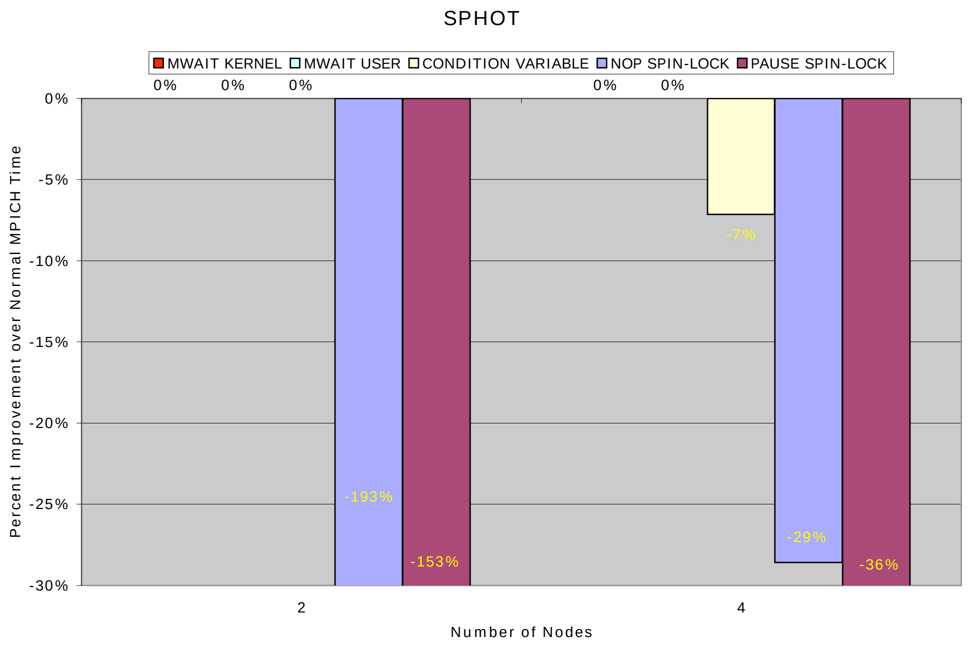


Figure 7.6: sPHOT Results

experiment lasting only 15 seconds. The size of this benchmark problem was kept the same across all experiments.

7.2.4 sPPM

sPPM is another benchmark that performs a lot of computation and then reduces the information from neighboring nodes. Nodes issue an MPI_Send() and the master node initiates an MPI_Allreduce(). Message sizes are 1 MB in size, which would beof benefit if it were overlapped with calculations, which is not the case. We see from Figure 7.8 that our MWAIT and Condition Variable MPICH modifications are on par with Normal MPICH for all tested cases. NOP and PAUSE MPICH modifications again reflect the CPU resource contention issue.

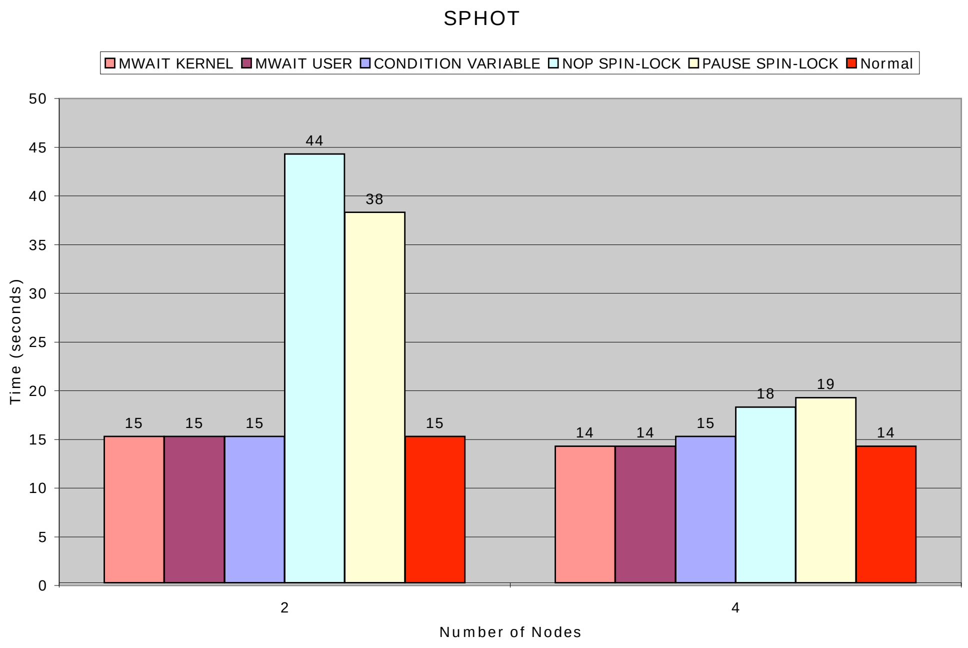


Figure 7.7: sPHOT Results (Time)

7.2.5 Sweep 3D

The Sweep3D benchmark executes for 80 to 150 seconds from two to four processors. It is dominated by the function sweep that consumes 98% of the run-time. After every sweep(), an AllReduce() is issued that uses blocking synchronous sends and receives to aggregate a global result. There is stub code in the benchmark to implement a non-blocking asynchronous call, and because messages are in the range from 16 to 512 KB, this application has potential for use in our approach if it was modified. Unfortunately, now it uses synchronous sends and receives, and we see from Figure 7.10 and 7.11 shows that there is no improvement again in any of the benchmarks. NOP and PAUSE MPICH conflict again during computation and during communication. In the modified MPICH, blocking synchronous sends and receives are handled by the computation thread, while the communication thread waits for new connections and requests. This also explains the degradation in performance of why NOP and PAUSE from two to four nodes. This benchmark problem size is kept the same across the benchmarks.

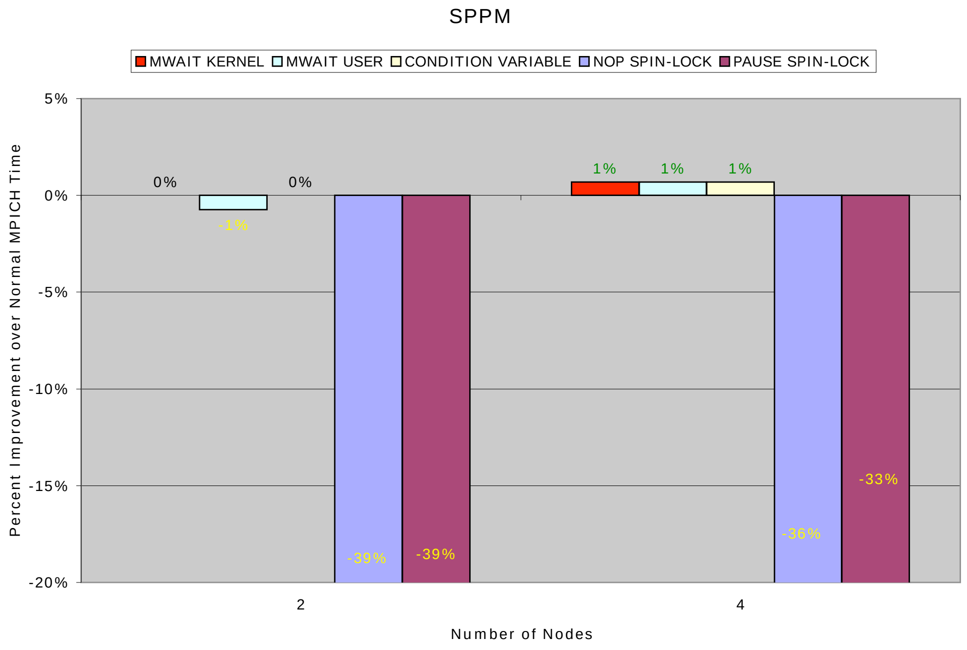


Figure 7.8: sPPM Results

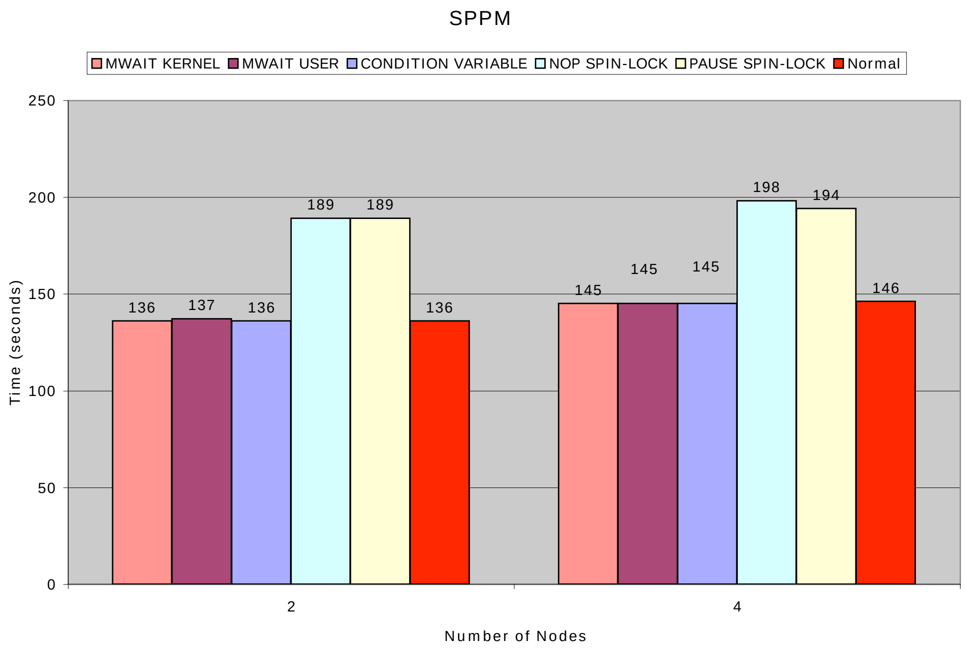


Figure 7.9: sPPM Results (Time)

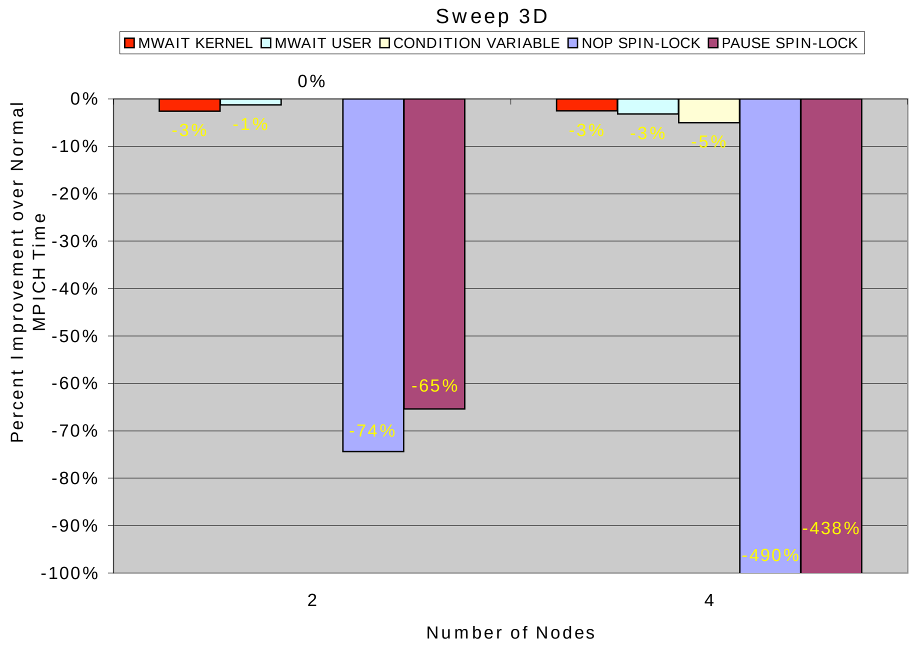


Figure 7.10: Sweep 3D Results

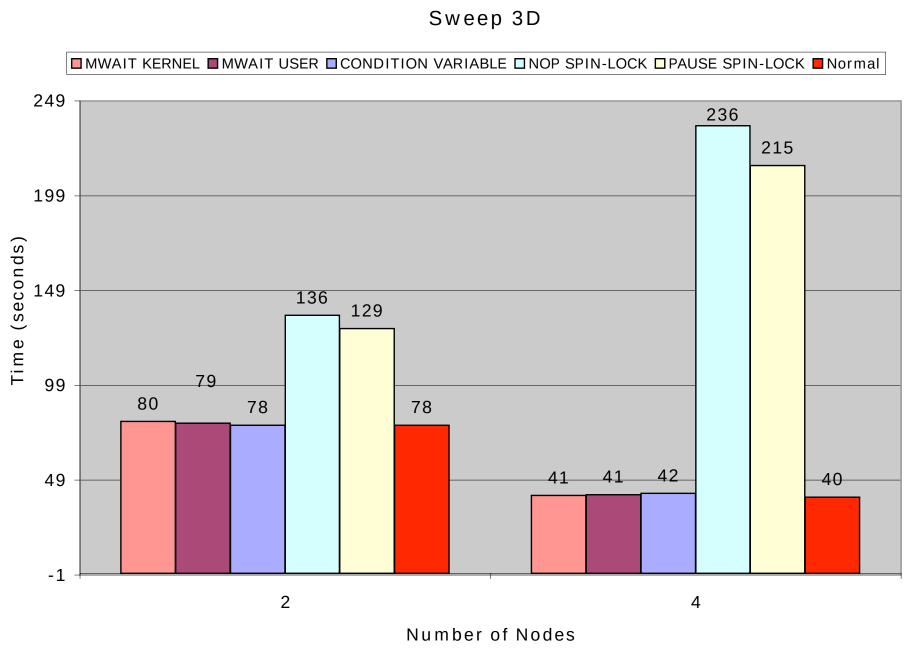


Figure 7.11: Sweep 3D Results (Time)

Chapter 8

Related Work

Parallelizing sequential problems and overlapping their overhead with computation has long been an approach to improving performance of many tasks in computers. Henry Ford popularized pipelining the building of cars to prevent workers from becoming idle and improve factory efficiency. Similarly, the IBM STRECH mainframe was the first to pipeline a microprocessor to improve execution efficiency, and the IBM 3084 was one of the first mainframes to have multiple microprocessors to execute multiple tasks at one time [IC]. Taking explicit advantage of SMT processors through asynchronous communication techniques has not been previously studied, but techniques to minimize program latency and throughput using asynchronous and synchronous techniques have been studied heavily. These problems frequently arises in processes demanding maximum efficiency such as process scheduling, processor pipelining, network communication protocols, disk access, etc. We considered several approaches on how to best parallelize communication libraries. Hoare outlined a parallel programming language and efficient synchronization primitives to take explicit advantage of multi-processors, independent controllers and other parallel features of microprocessors [Hoa78]. He promotes programming features to be as parallel and independent as possible to maximize efficiency. The BeOS operating system from Be, Inc. describes itself as a "pervasive multitasking" operating system. Almost every library and routine in the whole operating system, including the GUI interface, is non-blocking and asynchronous. In addition to the user's explicit non-blocking overlap, the operating system transparently breaks the application's tasks into smaller tasklets that are executed in parallel to maximize processor efficiency [Be]. Modern operating systems take advantage of hardware to allow slow tasks to complete while they take care of other tasks. Direct memory-addressing, modern hard disks, communication cards, video cards and any other peripheral use hardware interrupts to notify the operating system when a task has completed or needs to be completed. Modern operating systems take advantage of any hardware parallelism available to them. Our work is an extension of these concepts.

Promotional Buddy Scheduling is a special case of general-purpose scheduling algorithms and akin to gang-scheduling and symbiotic job scheduling. Snavely et al. study a convergent feedback scheduling algorithm that uses sampling along with custom prioritization techniques to discover the most efficient scheduling [STV02]. This technique may optimally schedule tasks, but it does not take into account thread-specific requirements, such as low IPC latency or locality. Dorai et al. optimize single-thread applications on SMT processors by minimizing the impact of all other tasks. In effect, they lower priorities of all background tasks or make them dependent on the primary application task [DYC]. This technique is similar to ours in that hardware and software are optimized for use by the primary tasks, and any subordinate helper tasks are low-priority low-use tasks. Wiseman and Feitelson take a similar approach to minimize process conflict on a larger scale [WF03]. Their Paired Gang Scheduler also uses a feedback scheduler to assign symbiotic tasks together to groups of processors in a cluster or multi-processor environment. They focus on pairing CPU-intensive tasks with I/O bound tasks on the same processor. This is a concept similar to our division of communication and computation. These works are focusing on single applications. None of them offer any user facilities like Promotional Buddy Scheduling or library-based transparent multi-threading to meet an application's specific needs.

There has been some research investigating methods to minimize the latency of communication through parallelism. Liu et al. show how the Infiniband network architecture can provide very low latency, high bandwidth communication through the use of zero-copy and one-sided communication [LWK^{+}03]. When a user calls MPI_Send or MPI_Isend, the underlying library sets up and enqueues the request, notifies the Infiniband network card and continues processing. The Infiniband card uses direct memory access to read directly from user memory (zero copy) without going into kernel mode or using further CPU cycles. This reduces most of the overhead associated with copying data into kernel space and forcing the kernel to initiate direct memory access (DMA). In addition, the Infiniband network cards support one-sided communication that allows it to send data directly into another node's user-space memory whereupon the remote node is notified that a message has arrived. This provides a close approximation to the simultaneity provided by the SMT processor and topics discussed in this thesis [LWK^{+}03].

The MPICH MPI library (Argonne National Laboratory) was chosen over LAM MPI (Indiana University) because of the maturity of its code base and simplicity in allowing for modifications [BDV94, GL97]. Both MPICH and LAM MPI are being modified constantly, and a second generation MPICH is under development that implements some asynchronous communication in order to further overlap communication and computation. MPI-2 one-sided communication sets up a virtual shared-memory allowing MPI nodes to send messages directly to other nodes' memory without intervention. The remote node is then notified that the data is in its memory and the communication completes. One-sided communication supports remote dynamic memory access (RDMA) using synchronous put and get operations on memory to simulate the shared memory in a distributed environment. In contrast, this thesis focuses on non-blocking sends and receives (Isend, Irecv) of messages and its potential to increase performance in distributed applications on SMT processors.

Chapter 9

Conclusions

We were able to successfully show that overlapping communication and computation through SMT hardware increases performance of scientific applications on SMT hardware. However, the magnitude of improvement is lower than expected. Choosing different benchmarks with greater communication and computation overlap that resemble our ideal model more closely is necessary. The Sweep3D benchmark documents describe how to modify the benchmark to almost completely overlap communication with computation. Other red-black computational kernels should also be explored. Nevertheless, we were able to see that our threaded communication model, low-latency synchronization primitives and promotional scheduler are effective and have great potential. The IRS benchmark is our showcase from this set. It improves 30% through communication overlap and fewer interruptions of the computation thread. We see benefits for sPHOT when using the MWAIT instruction over other primitives by rejoining the internal instruction queues of the processor to maximize performance. For sPPM, sPHOT and Sweep3D, we see that our modified MPICH performs equally well when using MWAIT-SysCall-USER/KERNEL-Space or Pthread-condition-variable for inter-thread communication and synchronization. We determined that we need to better support smaller packets in the future. SMG2000 illustrates that even though there are plenty of non-blocking calls, the overhead of using any of our threading techniques was either too high to provide a benefit or communication occurred in non-overlapping communication-computation sections. We were not able to use our promotional scheduler in these benchmarks, but we plan to use it in future work.

Taking advantage of the SMT hardware to perform symbiotic communication and computation in scientific applications can and does provide benefits under certain circumstances. Programs need to be specifically designed to take further advantage of SMT processors. We were able to support 4 out of 5 legacy applications on the SMT architecture without degrading their performance and, in some cases, improving it. The MWAIT instruction has great potential as an even faster synchronization primitive when available in user-space. Promotional scheduling can be used in distributed environments to provide maximum performance for users. Frequently, computing clusters deploy general-purpose operating systems built for fairness while the scientist needs every clock cycle and would be better-served with customized operating system kernels.

Chapter 10

Future Work

One of our performance limiting factors is the MPICH P4 communication sub-structure. It is well designed for networks up to 100 Mbps, but poor connection management, dependence on signals and system pipes hamper performance beyond 100 Mbps. Modifying the underlying P4 for higher throughput by improving the P4 layer may be more time-consuming than moving to another option such as the high speed, low latency Myricom Myrinet network [Myr]. This 800 Mbps network provides 7 micro-second latency and scales to thousands of nodes.

We want to investigate scaling this design to thirty-two and more processors. Vetter showed that for some computational programs, MPI communication increases super-linearly as the number of processors increases [VY02]. Frachtenberg and Vetter suggest that 16 to 64 processors should be enough to take advantage of substantial gains in performance due to the communication overhead [FPCcF01, VY02].

We would like to further explore promotional thread-scheduling in a gang-scheduler running on computational clusters. Our emphasis is on single application performance improvements, so each gang schedule would consist of all threads of a single program assigned to the processing elements. Implementing intelligent paired gang scheduling along with a performance counter feedback scheduler would allow higher densities of threads on the same contexts without degrading performance[FPCcF01, WF03, STV02]. Other individual programs would then be gang-scheduled for control of the processors. This gang scheduling would occur as an operating-system task scheduler as opposed to a global task scheduler that doles out groups of tasks to all processors available in the cluster.

We are in search of better applications to illustrate the benefits of our techniques. We are looking into red-black kernels and possibly more traditional network servers. In addition, we will be optimizing the current MPICH design for small message sizes and further reducing the overhead of thread synchronization.

Appendix A

Build Environment Instructions

A.1 How to use this repository

We setup a distributed benchmark compiling/management and cluster setup/management system through the use of shell scripting, rsync and distcc [Tri, Poo87]. This is a self contained multi-version environment that comes with compilers, libraries and other support files that can easily be setup on other computers to make a working compute node available. We have made every attempt to make this collection as portable as possible and as flexible as possible. The environment is based from a root directory ($ROOTDIR) from which all operations are based. The emphasis was to not depend on computer-local files, but carry a complete environment from machine to machine in order to minimize incompatibilities and maximize flexibility.

A.2 Setup

One could retrieve the latest version from Dr. Frank Mueller's CVS directory. Directions to setup for his CVS are at http://moss.csc.ncsu.edu/∼mueller/ . Checkout using cvs co -d [Your ROOTDIR directory] nvouk [your userid] . If you want to check out only a single benchmark, then you must minimally checkout the "bin" directory and the benchmark from "benchmarks/BENCHMARK_NAME". Be aware, though, that most items in the benchmarks directory that end in _thread should be deleted *except* mpich-1.2.5_thread.

All files mentioned are relative to the root directory ($ROOTDIR), unless explicitly stated. These steps are recommended for setup:

1. Basic items required are a set of computers inter-connected through a network. You should have administrative access to all machines if this is a brand new setup. The system works by designating a machine as a primary host from which all files will be pushed to other slave servers. Even though all machines should be able to act as primary host, you should pick just

one machine as a template machine to keep things clear. If you decide to work from multiple machines, you must remember to not make changes on other machines before synchronizing all machines. An alternative method that doesn't require synchronization is to use a distributed file system.

2. You should use bash as your primary shell, or rewrite conf/bashrc file in whatever shell language of choice.

3. You must add to your basic .bashrc file two lines:

```
declare -x ROOTDIR=$HOME // or other choice directory
. $ROOTDIR/bin/bashrc
```

   ROOTDIR is the reference directory under which all benchmarks,libraries, programs are placed. This does not have to be a home directory, but some other directory defined by ROOTDIR. You should keep the root directory the same across the cluster due to the way MPICH executes remotely.

4. The user will have to edit a few key files to customize for your specific setup. These files are used in the computer setup, so be careful in what you allow or don't allow. The files deal with letting users use 'rsh' and 'ssh' to login without a password. It also adjusts some firewall rules.

```
$ROOTDIR/bin/conf.pl
$ROOTDIR/bin/bashrc
$ROOTDIR/bin/hosts
$ROOTDIR/bin/hosts.allow
$ROOTDIR/bin/hosts.deny
$ROOTDIR/bin/hosts.equiv
$ROOTDIR/bin/rhosts
$ROOTDIR/bin/iptables
/etc/sudoers
/etc/group
```

   The user should look through conf.pl and edit the benchmarks (follow additional directions from the next section)

   These are the steps to get a node working manually. There is a utility called 'setupComputer' that can be executed from the template computer that executes most of these instructions. The only thing to be done afterwords is to copy the new machine's public key to the template computer, add it to the authorized_keys file in the bin directory, re-synchronize, and run 'setupCompile' on all machines.

In order to setup a machine, the user should use 'sinc' to have the repository copied to that machine. sinc username@machinename

The account needs to setup your user for use with 'sudo'. Sudo is important as it is used in several places and you will be typing your password annoyingly too much otherwise. Most of the time the user just needs to add himself to the 'wheel' group (gpasswd -a user wheel) and edit the /etc/sudoers file.

For communication, the user should run the command 'setupCompile'. This should be checked before running as it modifies several system configuration files. 'setupCompile' will generate an ssh dsa key, copy configurations for rsh and ssh and also an iptables firewall config. Be aware that there are some security considerations to consider here because of rsh's inherent insecurity.

The user will need to copy the id_dsa.pub key to your template server and then run 'sinc' again in order to get ssh to work through dsa keys. The account owner must be able to login without having to type a username or password. Copy the new machine's public key and add it to the authorized_keys file in the bin directory of the template computer. Re-'sinc' and run 'setupCompile' again to complete installation.

Once the last step has been accomplished, the user should be able to freely use rsh or ssh to execute and login to any machine across the network.

A.3 Setup benchmarks

This is a Multi-version compilation and testing system. This means that it supports several copies of the same program, but operated on slightly differently. This project is keyed off of the MPICH MPI library compiled statically against several test benchmarks. There are utilities, namely switch, which go through the set of benchmarks and change pointers in the ROOTDIR to 'alternate' versions of files. This can be applied to any type of directory, not just benchmarks. To change all of these features, edit the 'switch' utility.

The lynch-pin to making this system work are the three utilities 'switch','sinc' and 'buildall'.

- switch: Switch facilitates the multi-version system. Be-aware though, it works under the model that benchmarks have a 'normal' version of a program, which may be modified at will, but the same version must be used for 'alternate' versions. There is a normal and alternate MPICH library, which can be modified freely. New benchmarks added should be suffixed with '_normal'. Then, when a 'switch' occurs, the 'normal' version is copied and suffixed with '_thread'. 'Switch' checks for the '_thread' copy and if it exists, it does nothing but switch the pointers in the root directory.

Switch uses the '%benchmarks' hash in conf.pl to get a mapping between the short root directory name of the benchmark and the corrosponding long name. This helps to allow for changes in versions and keep things "straight". The symbolic links created in the root directory can be reference by other libraries to keep

- switchall:

  This is a corrosponding application to 'switch' that "switches" all the machines in your '@machines' array at once.

- sinc: Sinc, in the default configuration, it uses rsync to copy all items declared in the "@dirs2" array in conf.pl. Add/Delete items to this array for automatic copying. It uses the "@machines" array as a default set of machines. Pass '-h' to get all the options as you can synchronize individual directories from the ROOTDIR to other hosts. You should look at the "%machines" hash to adjust the communication protocol desired to access those machines (ssh/rsh/other).

- buildall: Buildall allows you to build all applications in a particular order with whatever customizations are needed in order to make it work. It can build particular projects or all of them (pass it -h to see options). It uses the '%benchmarkBuild' hash keys as input, and then executes the corrosponding instructions. Given no arguments, it will follow the order of items to build in the '@buildOrder' array. This allows you to build your whole system to your exact specific5ation.

- tell: Tell is a simple utility to control other nodes in your network (or outside). It takes a hostname as an argument and passes that to ssh, if the communication method is not defined in '%machines' hash. It then executes whatever commands are passed to it on that host.

- tellall: Corresponds to the 'tell' command, but tells all machines the same command.

A.4 Compiling

For the benchmarks provided, we used primarily the Intel 8.1 fortran and C/C++ compiler. We also provided support, in some cases, for gcc as in mpich, but only where necessary. The compilers are installed in $ROOTDIR/opt/intel_80_cc and $ROOTDIR/opt/intel_80_fc. Symbolic links are provided for future compiler upgrades. Adjust the $ROOTDIR/opt/cc and $ROOTDIR/opt/fc links accordingly.

A.5 Distributed Compiling:

We have also provided a facility for using a distributed compiling system, secure and insecure, distcc (http://distcc.samba.org/) and also the caching preprocessor ccache. We wrapped the necessary make commands into the command 'distmake', so you can use 'distmake' where ever 'make' is used. To use this properly, please modify the @machines and the @DISTCC_HOSTS arrays in $ROOTDIR/bin/conf.pl to adjust which machines are used for compiling. The utility automatically checks if a machine is up or down to allow for changing networks.

- setupdistcc: Used by distmake to start the distcc daemon on that machine.
- distmake:

  If you compile something using distmake, please use distmake throughout the build/install process, even if you need to use sudo or be root to install. The distmake utility checks if you are effectively the root user and does not distribute. If distributed compilation as root is desired, you will have to edit the distmake utility and take out the check or add an option.

  For distributed compiling using ssh communication, pass the -ssh option as the first option.

  To use a C compiler other than gcc, pass the type of compiler first, or if using ssh, then after the -ssh option. It recognizes icc, gcc, icpc, xlc, XlC, g++, pgcc, pgCC and their respective fortran versions.

  By default, distmake assigns three times as many tasks to pass to 'make -j' as there are machines. Edit the file to change the number of tasks assigned.
- iccbuild: Setups up the environment variables to use Intel fortran and C compilers.
- gccbuild: Setups up the environment variables to use GNU fortran and C compilers.

A.6 Testing benchmarks

We have written some automated testing tools to help ease testing of the various options.

- maketests: 'maketests' is a robust utility to test each individual benchmark, while also recording the output. The output is sent to $ROOTDIR/results and allows the passing of many modifications. Each of the benchmarks' Makefile was modified to accept these parameters. Pass '-h' to see all the options defined by 'maketests'.
- manualtest: This program attempts to execute an MPI benchmark manually instead of thread MPIRUN.
- dotests: An automated script that tests all benchmarks, threaded and normal.

- cmwait: This controls the kernel mwait modifications. If you happen to kill a process that is calling the mwait system call, you will not be able to break it without this tool or rebooting. Pass the command 'status' to check if MWAIT is enabled or disabled. Pass 'cmwait' on or off to enable and disable.
- killprog: This is a script to cleanup leftover programs. Pass it a program name to kill it.

A.7 Miscelanious Tools

These are some random utilities written to help development.

- gensmtthrds: Experimental utility. Modifications to each of the source files in MPICH are very similar, so We wrote an automated tool to do the process efficiently and consistantly. The tool pseudo-works, but needs some more fixing. Useful, in that it contains a template of the general modifications necessary for each file. Use on files in mpich/src/, except for mpich/src/ht.
- gccDefines: Contains the LIB, CC, FC, etc. environment definitions for the GNU GCC/G77 compilers. Used by gccbuild.
- iccDefines8: Contains the LIB, CC, FC, etc. environment definitions for Intel 8.X ICC/IFORT compilers. Used by iccbuild.

Bibliography

[And91] Gregory R. Andrews. *Concurrent Programming - Principles and Practice*. The Benjamin/Cummings Publishing Company, Inc, 1st edition, 1991.

[Bar01] Blaise Barney. Sphot - monte carlo transport code - http://www.llnl.gov/asci/purple/benchmarks/limited/sphot/sphot.read.html, September 2001.

[BDV94] Greg Burns, Raja Daoud, and James Vaigl. LAM: An Open Cluster Environment for MPI. In *Proceedings of Supercomputing Symposium*, pages 379–386, 1994.

[Be] Inc. Be. The media os - http://www.beatjapan.org/mirror/www.be.com/products/beos/mediaos.html.

[BHMU02] Marr Binns, Hill Hinton, Koufaty Miller, and Upton. Hyper-threading technology architecture and microarchitecture. *Intel Technology Journal*, 6(1):4–16, February 2002.

[Car01] Brian Carnes. Smg2000 semicoarsening multigrid solver - http://www.llnl.gov/asci/purple/benchmarks/limited/smg/smg2000_readme.html, September 2001.

[Car02] Brian Carnes. The asci sweep3d benchmark code - http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/sweep3d_readme.html, March 2002.

[Car03] Brian Carnes. Irs: Implicit radiation solver 1.4 - http://www.llnl.gov/asci/purple/benchmarks/limited/irs/irs.readme.html, July 2003.

[Cor04] Intel Corporation. *IA-32 Intel Architecture Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z*, 2004.

[DM03] Ulrich Drepper and Ingo Molnar. The native posix thread library for linux - http://people.redhat.com/drepper/nptl-design.pdf, January 2003.

[DYC] Gautham K. Dorai, Donald Yeung, and Seungryul Choi. Optimizing smt processors for high single-thread performance.

[Eng02] John Engle. The asci sppm benchmark code, February 2002.

[FPCcF01] Eitan Frachtenberg, Fabrizio Petrini, Salvador Coll, and Wu chun Feng. Gang scheduling with lightweigth user-level communication. In *2001 International Conference on Parallel Processing (ICPP2001), Workshop on Scheduling and Resource Management for Cluster Computing*, Valencia, Spain, September 2001.

[GL97] W. Gropp and E. Lusk. A high-performance MPI implementation on a shared-memory vector supercomputer. *Parallel Computing*, 22(11):1513–1526, 1997.

[HKN$^+$92] H. Hirata, K. Kimura, S. Nagamine, Y. Mochizuki, A. Nishimura, Y. Nakase, and T. Nishizawa. An elementary processor architecture with simultaneous instruction issuing from multiple threads. *In 19th Annual International Symposium on Computer Architecture*, pages 136–145, May 1992.

[HMR$^+$00] Jerry Huck, Dale Morris, Jonathan Ross, Allan Knies, Hans Mulder, and Rumi Zahir. Introducing the ia-64 architecture. *IEEE Micro*, September-October 2000.

[HNO97] Lance Hammond, Basem A. Nayfeh, and Kunle Olukotun. A single-chip multiprocessor. *Computer*, 30(9):79–85, 1997.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[HPR$^+$02] Wang H., Wang P., Weldon R.D., Ettinger S.M., Saito H., Girkar M., Liao S.S-W., and Shen J. Speculative precomputation: Exploring the use of multithreading for latency. *Intel Technology Journal.*, 6(1):22–35, February 2002.

[ia304] *IA-32 Intel Architecture Software Developer's Manual Volume 1: Basic Architecture*, 2004. 245470-012.

[IC] Inc. IBM Corporation. Ibm mainframes introduction - http://www-03.ibm.com/ibm/history/exhibits/mainframe/mainframe_intro3.html.

[Lab] Argonne National Laboratory. Mpich-a portable implementation of mpi.

[LBE$^+$98] Jack L. Lo, Luiz Andre Barroso, Susan J. Eggers, Kourosh Gharachorloo, Henry M. Levy, and Sujay S. Parekh. An analysis of database workload performance on simultaneous multithreaded processors. In *ISCA*, pages 39–50, 1998.

[Lov] David Love. Ht support in linux kernel - http://kerneltrap.org/node/view/391/1027l.

[Luk99] Chi-Keung Luk. Tolerating memory latency through Software-Controlled Pre-Execution in simultaneous multithreading processors. In *Tolerating Memory Latency* *through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors*, pages 40–51, 1999.

[LWK$^+$03] J. Liu, J. Wu, S. Kini, P. Wyckoff, and D. Panda. High performance rdma-based mpi implementation over infiniband, 2003.

[Mae02] Toshiyuki Maeda. Safe execution of user programs in kernel mode using typed assembly language. Master's thesis, University of Tokyo, http://web.yl.is.s.u-tokyo.ac.jp/ tosh/kml/, 2002.

[Mil87] Milan Milenković. *Operating Systems: Concepts and Design*. McGraw-Hill Book Company, 1st edition, 1987.

[MPI] Mpi 1.1 standard - http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html.

[Myr] Myricom. Myrinet - http://www.myri.com/.

[NBF96] Bradford Nichols, Dick Butler, and Jackie Farrell. *Pthreads Programming*. O'Reilly and Associates, Inc., 1st edition, Fall 1996.

[Not] Windows Platform Design Notes. Windows support for hyper-threading technology - http://www.2cpu.com/hardware/ht_analysis/hyperthreading.doc.

[OSU99] Heiko Oehring, Ulrich Sigmund, and Theo Ungerer. MPEG-2 video decompression on simultaneous multithreaded multimedia processors. In *IEEE PACT*, pages 11–16, 1999.

[Poo87] Martin Pool. distcc: a fast, free distributed c/c++ compiler http://distcc.samba.org/, 1987.

[PSR00] Zachary Purser, Karthik Sundaramoorthy, and Eric Rotenberg. A study of slipstream processors. In *International Symposium on Microarchitecture*, pages 269–280, 2000.

[Pur] ASCI Purple. http://www.llnl.gov/asci/platforms/purple/.

[RKT04] Balaram Sinharoy Ron Kalla and Joel M. Tendler. Ibm power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, March/April 2004.

[RM00] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient fault detection via simultaneous multithreading. In *ISCA*, pages 25–36, 2000.

[Ros05] Chad Rossier. Exploiting thread level parallelism in a simultaneous multithreading processor. Unprinted document for SIGPLAN SPPPP, 2005.

[SDR01] G. Suh, S. Devadas, and L. Rudolph. Dynamic cache partitioning for simultaneous multithreading systems. *http://citeseer.ist.psu.edu/suh01dynamic.html*, 2001.

[Set04] *Architectural Support for Enhanced SMT Job Scheduling*, September-October 2004 2004.

[ST97] Jack L. Lo ; Susan Eggers; Joel S. Emer; Henry M. Levy; Rebecca L. Stamm and Dean M. Tullsen. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Transactions on Computer Systems*, 15(3):322–354, 1997.

[Sto01] Joe Stokes. The pentium 4 and the g4e: An architectural comparison: Part 1. *Ars Technica - http://arstechnica.com/articles/paedia/cpu/p4andg4e.ars/1*, July 2001.

[Sto02] Joe Stokes. Introduction to multithreading, superthreading and hyperthreading. *Ars - Technica http://arstechnica.com/articles/paedia/cpu/hyperthreading.ars/*, October 2002.

[Sto04] Joe Stokes. Ibm's power5: A talk with pratap pattnaik - http://arstechnica.com/articles/paedia/cpu/power5.ars. *Ars Technica*, oct 2004.

[STV02] A. Snavely, D. Tullsen, and G. Voelker. Symbiotic jobscheduling with priorities for a simultaneous multithreading processor. *http://citeseer.ist.psu.edu/snavely02symbiotic.html*, 2002.

[TEL95] Dean M. Tullsen, Susan Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *Proceedings of the22th Annual International Symposium on Computer Architecture*, 1995.

[Tri] Andrew Tridgell. The rsync algorithm: http://samba.anu.edu.au/rsync/.

[VkPC02] T. Vijaykumar, k Pomeranz, and K. Cheng. Transient fault recovery using simultaneous multithreading - http://citeseer.ist.psu.edu/623918.html, 2002.

[VM02] J. Vetter and F. Mueller. Communication characteristics of large-scale scientic applications for contemporary cluster architectures. *http://citeseer.ist.psu.edu/vetter02communication.html*, 2002.

[VY02] J. Vetter and A. Yoo. An empirical performance evaluation of scalable scientific applications. *Super Computing*, 2002.

[Wal91] D. W. Wall. Limits of instruction-level parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System (ASPLOS)*, volume 26, pages 176–189, New York, NY, 4 1991. ACM Press.

[WCT98] Steven Wallace, Brad Calder, and Dean M. Tullsen. Threaded multiple path execution. In *ISCA*, pages 238–249, 1998.

[WF03] Yair Wiseman and Dror G. Feitelson. Paired gang scheduling - http://citeseer.ist.psu.edu/wiseman01paired.html, 2003.

[WM95] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: Implications of the obvious. *Computer Architecture News*, 23(1):20–24, 1995.

[YKME$^+$02] Chen Y-K., Holliman M., Debes E., Zheltov S., Knyazev A., Bratanov S., Belenov R., and Santos I. Media applications on hyper-threading technology. *Intel Technology Journal.*, 6(1):47–57, February 2002.