MANAGING THE OVERALL BALANCE OF OPERATING SYSTEM THREADS ON A MULTIPROCESSOR USING AUTOMATIC SELF-ALLOCATING THREADS (ASAT)

By

Charles R. Severance

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

1996

ABSTRACT

MANAGING THE OVERALL BALANCE OF OPERATING SYSTEM THREADS ON A MULTIPROCESSOR USING AUTOMATIC SELF-ALLOCATING THREADS (ASAT)

By

Charles R. Severance

While commodity processor based parallel processing systems have an advantage over traditional supercomputers in price/performance, traditional supercomputers retain a significant advantage over parallel processing systems in overall system dynamic load balancing. Traditional supercomputers can easily handle a mix of interactive, batch, scalar, vector, parallel, and large memory jobs simultaneously while maintaining high utilization. This work focuses on an effort to make a large commodity based shared-memory parallel processing system perform as well as a traditional parallel/vector supercomputer under dynamic load conditions with many users. A solution called Automatic Self-Allocating Threads (ASAT) is proposed as a way to balance the number of active threads across a multi-processing system. Dynamically matching the number of active threads to the available system resources improves performance by eliminating contention for resources. The approach used by ASAT is significant in that it is designed for a system running multiple jobs, and it considers the load of all running jobs in its thread allocation. In addition, the overhead of ASAT is sufficiently small so that it can be used as part of the startup processing for every parallel loop. Furthermore, the approach uses self-scheduling so it can be implemented in a run-time environment rather than in an operating system and not all jobs need to be using ASAT scheduling.

Dedicated to my wife Teresa, daughter Amanda, son Brent, and parents Russell and Marcia for all of their support.

ACKNOWLEDGMENTS

In an educational career which spans over 21 years at the same institution, there are many people who deserve my appreciation. I will note some of them here. First, I am deeply indebted to my advisor Dr. Richard Enbody. His tireless effort and boundless patience is very much responsible for this work. I would also like to thank my committee members, Dr. Lionel M. Ni, Dr. Abdul-Hossein Esfahanian, and Dr. David Tomanek. They have made this a truly remarkable learning experience. I would like to thank Dr. Sakti Pramanik who as my masters thesis advisor originally taught me the basics of research and publishing. I would like to thank my masters academic advisor the late Dr. Carl Page. And even if it has been so long ago that perhaps he has forgot he was my undergraduate advisor at Michigan State University, I thank Dr. Richard Reid. I would like to thank several other members of the Department of Computer Science at Michigan State University. I would like to thank Dr. Tony Wojcik for his support, guidance, and for giving me a chance to teach at Michigan State University in 1990. I would like to thank Dr. Charles R. Maccluer of the Department of Mathemetics for his advice and counsel over the years. I thank the teachers who over the years have had an impact on my education. I would like to thank Dr. Harold Grossman of Lyman Briggs for teaching the class during which I fell in love with computers (despite occasionally falling asleep in class). I thank Dr. Hans Lee for his CPS300 class in which I learned the important lesson to use an algorithm if one is provided on the program handout. I would like to thank Dr. Martin Keeney for teaching me to hand in assignments on time and that college was something to take very seriously. I would like to thank Dr. Abdul-Hossein Esfahanian, Dr. Richard Enbody, Dr. Eric Torng, Dr. Phillip McKinley, and Mark McCullen for the classes which they have taught so well that I learned the material and how to teach. From my sophomore year on, I was employed full-time at Michigan State University while also taking classes – hence the 21 years. During that time, many of my supervisors, employees and colleagues have been very understanding of my seeming obsession with getting a doctorate. I would like to thank Richard Moore, Dr. Lewis Greenberg, Dr. Paul Hunt, and Dr. Ted Bickart for their support and understanding. I would like to thank Dr. David Kuck, Dr. Paul Petersen, and the others of Kuck and Associates for their guidance on ASAT and for implementing ASAT in their compiler. There were many people who have supported, encouraged, and assisted me over the years in many ways including: Tom Davis, Richard Wiggins, Paul Wolberg, Ted Shotwell, Paul Barczak, Chuck Henrich, and Jerold McAllister. Last but not least, I would like to thank my fellow graduate students without whose support I would never have got this far including: Steve Wagner, Barb Birchler, Mark Brehob, Natawut Nupairoj, Hugh Smith, Jerry Gannod, and Travis Doom. I wish them all the best in their own academic pursuits.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

This thesis describes a tool called Automatic Self-Allocating Threads (ASAT) which addresses the problem of dynamic load balancing to make efficient use of low cost parallel processing systems under widely varying loads. ASAT has been implemented as part of the FORTRAN run-time environment for parallel applications. ASAT adjusts the number of active threads executing in an application as the overall system load changes to maintain thread balance across the entire system.

Most dynamic load balancing research focuses on the effective division of computations, data, or tasks among the threads of a running process or across several computer systems. This work focuses on how a single parallel processing system processing multiple unrelated processes each with multiple threads can most effectively make joint use of the overall system resources.

1.1 Motivation

The high performance computing and scientific computing market segments are making a dramatic transformation from the heavy use of vector supercomputers to medium scale symmetric parallel processing systems based on commodity processors. The peak performance of a high-end commodity Central Processing Unit (CPU) is within a factor of three of all but the fastest supercomputer CPU. This price/performance has caused a migration away from shared central computer resources toward dedicated desktop compute resources for many applications. However, regardless of how fast individual desk top systems become, many users still have applications which need more cycles, memory, or disk than are available on their desk top. In order to satisfy those users' quest for more cycles, parallelism is the only remaining way to provide 2-3 orders of magnitude performance increase over the typical desktop system. In this computing environment, parallel processing systems such as the SGI Challenge and Power Challenge have enjoyed excellent market penetration against the traditional Cray or Convex vector supercomputers. Computing centers have been purchasing these systems with 16 to 32 processors and large memories to replace central vector supercomputers.

However, unlike a workstation, such a large single resource must often be shared among a pool of compute users in order to amortize its cost. As we will show later, these parallel processors have excellent performance for a multi-processing load and excellent performance for a single multi-threaded compute application. Unfortunately, these processors still have trouble dealing with the combination of running a multi-processing load and a multi-threaded compute application simultaneously. Furthermore, these processors have difficulty in handling more than one multi-threaded compute application simultaneously without using Gang Scheduling. The problem occurs when there are more active threads than available processors. This limitation in flexibility remains a primary disadvantage with respect to older supercomputer systems.

This lack of flexibility on these commodity-processor based parallel processors has not significantly slowed the migration away from vector supercomputers. The price/performance of these systems is so superior that even if 50% of the cycles are wasted, these Symmetric Multi-Processing (SMP) systems are still the better purchase in many situations. When vendors of these systems are asked about the limitations of their systems in being used as a shared *computing* resource, they have one of several reactions:

- They suggest that because "CPUs are so cheap" you can buy some extra CPUs to make up for the inability to use the existing CPUs at 100%.
- They suggest that you "partition" (also known as space sharing) your 16-way parallel processing system into four 4-way parallel processing systems. In this way at least four people can operate without impacting each other.
- They suggest that you buy two or more of their systems. One system can be used for interactive use and the second system can be used as a "batch back end". Again this is a form of space sharing and limits the ability to achieve 100% overall utilization. It is also expensive to replicate features such as a very large memory across two systems to make the interactive environment precisely mirror the production environment.

The interesting running theme in each of these vendor approaches to solving the problem is that they entail purchasing more hardware. Also, in each of the vendor solutions above, the user community must take some of the responsibility for these systems inability to handle dynamic computational load. Few procurements actually take into account the ability to perform dynamic load balancing between a multi-processing and multi-threaded load when designing benchmarks for these low cost systems. Most benchmarks (user developed or industry-wide) are run on "stand-alone" systems with no other activity so effective utilization is not an issue. As we demonstrate later, the negative performance impact of a dynamic load is surprisingly large.

The purpose of ASAT is to allow the users of these new style parallel processing systems to "have their cake and eat it too". That is, they will have all of the price/performance advantages of the low-cost SMP systems with the flexibility of the vector supercomputers. Most importantly, when a user purchases a system with sixteen CPUs which supports ASAT, they will actually be able to expect to use 100% of those sixteen CPUs regardless of the dynamic load patterns on the system.

1.2 Execution Model

The goal of ASAT is to dynamically adjust the number of threads of a process so that the number of threads matches the number of available processors. The basic concepts of ASAT can be used in a wide variety of multi-threaded applications ranging from explicit, user-controlled threads to run-time environments for parallel implementations of functional languages. This work focuses on FORTRAN applications using automatic, compiler-generated parallelism. ASAT depends on an execution model in which a serial portion of the code is periodically executed between the parallel sections of the code. In Figure 1, the time profile of this type of application is shown.

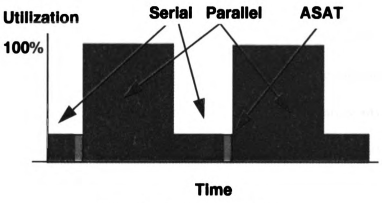

Figure 1 - Execution Model

The duration of the serial and parallel portions of the time profile varies widely, but many applications which are well suited to parallel processors have relatively short periods of serial execution followed by relatively long periods of parallel execution. ASAT depends on the existence of the serial code because it performs the thread adjustment as part of the startup of the parallel portion of the code as shown in red in Figure 1.

A FORTRAN program similar to the following will generate this common serial-parallel pattern when compiled with compiler-detected parallelism:

```
DO ITIME=1,INFINITY
  ...
   DO PARALLEL IPROB=1,PROBSIZE
     ...
   ENDDO
  ...
ENDDO
```

ASAT determines the proper number of threads to execute a parallel loop each time the parallel loop starts. The proper number of threads depends on the system's multi-processing load and other multi-threaded compute jobs.

1.3 ASAT Implementation

The key implementation detail in ASAT is determining the appropriate number of threads before each parallel loop without adding significant overhead to the loop startup. Each ASAT job evaluates the overall system load independently, eliminating any need for a central server process, shared data structure, or operating system modification. Because of its completely distributed load evaluation, ASAT jobs can efficiently operate with any combination of multitasking *and* multi-threaded jobs which are not using ASAT.

The actual goal of the of ASAT is not to precisely determine the number of threads which is appropriate, but rather to determine if the current number of threads is too large or too small. The situation in which there are an inappropriate number of threads is called "thread imbalance". When there more active threads than processors it is called "excess threads", and the situation where there are fewer threads than processors is called "wasted resources". When an ASAT process detects that there are more active threads than processors operating across the entire system it drops a thread. When an ASAT process detects that there are more processors than active threads across the entire system, it adds a thread. In this way, ASAT jobs always move toward overall system equilibrium by implicitly reacting to any combination of innumerable changes in system load.

A number of different approaches to determine the relative thread balance were tried [26, 27, 29, 30]. The approach which was the most successful was to periodically run a barrier synchronization and time the barrier passage. It will be shown later that that there is up to three orders of magnitude difference in barrier passage times when comparing thread-balanced to thread-imbalanced conditions. The system real-time clock and a usersettable parameter are used to insure that the ASAT evaluation is not run too frequently. This timed barrier must be run during the serial portion of the code, and any changes in the number of active threads takes effect when the next parallel loop is encountered.

For example, to use the public domain version of ASAT on the SGI Challenge[35], the calls to the ASAT run-time library must be added immediately before each parallel loop:

```
DO ITIME=1,INFINITY
  CALL ASAT_ADJUST()
  DO PARALLEL IPROB=1,PROBSIZE
     Work..
  ENDDO
ENDDO
END
```

The ASAT_ADJUST routine handles all the timer operations, performs the thread balance test if necessary, and adjusts the number of threads for the next parallel loop, if thread imbalance is detected. This routine also periodically determines if there are idle cycles and increases threads, if it is appropriate. The details of ASAT implementation are covered in Chapter 3.

1.4 ASAT In Operation

Using ASAT, an application can react to changes in system load regardless of the source of the changes. ASAT can react to other long running parallel jobs, medium length jobs such as compiler runs, bursts of interactive usage such as editing, load due to incoming network activity, load due to system server processes, excessive operating system overhead, and implicitly compensates for input-output activity on the system. Figure 2 shows how an application with ASAT generally operates when working on a four-processor system with variable load. In this figure, a single application using ASAT is executing while other users are using the system in different ways. As the load average increases due to other users, the ASAT application releases threads to maintain balance. Under high load conditions, the ASAT application only has one thread. As the other load decreases, the ASAT application adds threads, increasing its throughput by using the idle cycles.

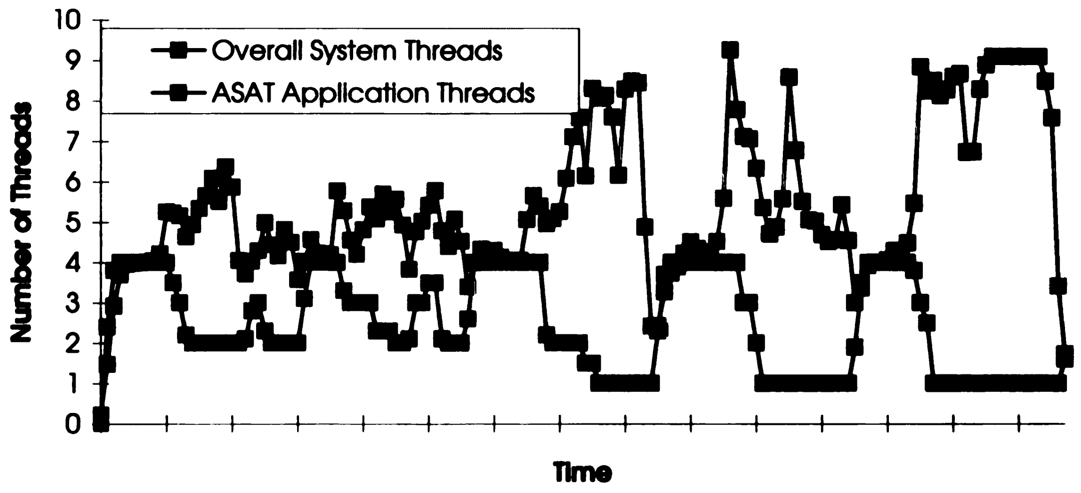

Figure 2 - Operation of ASAT Over Time

In the rest of this thesis, the dynamic load balancing ability of ASAT is shown to be excellent under a wide variety of different loads, and is studied on several computer architectures.

1.5 ASAT in a Compiler Run-Time Environment

The initial versions of ASAT required that library calls be manually added before the parallel loops as shown above. The ultimate goal is for ASAT to be supported directly by the compiler. In this latter case, there are two possible options for its implementation. In the first option, ASAT is transparently called before every parallel loop as part of the run-time library. In this way, all parallel users using that compiler will use ASAT processing (i.e. ASAT is not optional – like ASAP on the Convex). Another approach is to control its use through a compiler directive such as:

```
       DO ITIME=1,INFINITY
         ...
C$DOACROSS LOCAL(IPROB),THREADS(ASAT)
         DO IPROB=1,PROBSIZE
           ...
         ENDDO
         ...
      ENDDO
```

In this way, the user can decide when ASAT is used on a particular loop. The actual approach may vary from compiler to compiler and possibly even from system to system under the control of the system administrator.

There is one possible programming style which will be in conflict with a compiler run-time which provides ASAT without the awareness of the programmer. Some highly tuned parallel applications contain code which depends on a particular number of threads or that the same number of threads execute in every loop. A contrived example of this programming style is as follows:

```
         REAL*8 A(1000,4)
         IF ( NUM_THREADS() .GT. 4 ) STOP
   C$ DO_PARALLEL
         DO I=1,NUM_THREADS()
           DO J=1,1000
             A(J,I) = 0.0
           ENDDO
         ENDDO
   C$ DO_PARALLEL
         DO I=1,NUM_THREADS()
           DO J=1,1000
             A(J,I) = A(J,I) + 3.14159
           ENDDO
        ENDDO
```

The above code will work properly without ASAT as long as the number of threads never exceeds four. However, if ASAT implicitly adjusts (increases or decreases) the number of threads between the loops, the results will most likely be incorrect.

This style of programming can also be used to allow the programmer to manually perform load balancing, iteration scheduling, or control the patterns of access to memory when the compiler does not provide a robust enough solution for their particular application. Like iteration scheduling, these techniques often assume a fixed number of threads and may not work efficiently when the number of threads changes from one loop execution to another.

1.6 Previous Work

The first part of this section covers the previous work which relates directly to the dynamic management of the number of run-time threads for compute-intensive applications. The second part of this section covers the broader area of dynamic load balancing and places ASAT relative to other dynamic load balancing work. Chapter 2 covers some additional previous work combined with experimental results which develop a broader framework for the environment in which ASAT is intended to be deployed.

1.6.1 Dynamic Thread Adjustment Techniques

The dynamic thread management work most closely related to ASAT falls into three categories:

- Hardware approaches

- Software approaches which depend on operating system modifications
- Software approaches which depend on all the processes cooperating through a central data structure

The Convex C-Series [7] vector/parallel supercomputers implemented low-overhead dynamic thread count adjustment using significant hardware features collectively called Automatic Self-Allocating Processors (ASAP). In many ways the design of ASAT is inspired by ASAP. In [29] and in later chapters, the performance of ASAT will be directly compared to the performance of ASAP.

Cray Research's Autotasking [9] dynamically manages threads, but does not actually alter the number of threads during run-time. Through a combination of hardware, run-time software, iteration scheduling, and operating system support, Autotasking can operate efficiently across a wide range of load conditions. Autotasking allows the number of threads *executing* in a particular parallel loop to change during the execution of the loop as a result of load change through a cooperative agreement between the run-time library and operating system.

Both [1] and [37] proposed a solution which can dynamically adjust the number of run-time threads to accomplish overall load balancing. This approach depends on modifications to the operating system to maintain information regarding the load state of the system. In [1] the operating system actually removes an executing thread from the application when it detects excess threads. The application and run-time library had to cope with the loss of the thread and still insure correct program execution. In [Tuck93] this load information was periodically consulted by the SPLASH run-time environment which would then adjust the thread usage appropriately for each parallel section. In [1] the technique is called "scheduler activation's" and in [37] it is called "process control".

The significant negative performance impact of thread imbalance on these commodity processor based parallel processing systems was identified in [1,37,21]. In [37] the negative performance impact was broken down into its component causes and carefully measured using the SPLASH [SPLASH] benchmark applications. These issues are explored further in Chapter 2.

Recent work in [41,42,43] is most closely related to the current ASAT . This work seems to have evolved at about the same time as ASAT although ASAT was published earlier. The technique is called "Loop-Level Process Control" (LLPC). Like ASAT, LLPC does not require any operating system or hardware modifications. The primary difference between LLPC and ASAT is that LLPC communicates the overall system load information among the LLPC-enabled processes using a shared memory location. The primary limitation of the LLPC approach is that it is difficult for this approach to adapt to a changing multi-processing load or a multi-threaded load which does not use LLPC. This limitation could be mitigated by creating a "shepherd" process to periodically update the central data structure with operating system provided load information. Unfortunately, this would require the addition of a locking protocol which was eliminated from LLPC because of its negative performance impacts.

While the implementation of the LLPC run-time load test is inferior to that used in ASAT, they performed an excellent study [43] of the positive value of dynamically adjusting the number of run-time threads. This study showed that a wide range of applications (Perfect Club) [Perfect] demonstrate the parallel-serial pattern of execution necessary for all of these software-based approaches (ASAT, LLPC, Scheduler Activation's, and Process Control) to work effectively.

This thesis summarizes and extends a long history of ASAT publications. The initial work on ASAT for the SGI Challenge was described in [26]. A second version of ASAT was developed and ported to the Convex Exemplar [CONEXMP] and those results were reported in [27]. The results of ASAT on the SGI and Convex Exemplar were presented as a poster session in [28]. The ASAT performance on the Convex Exemplar was compared to the Convex ASAP hardware solution in [29]. In [30] the second version of ASAT was evaluated on the SGI Challenge. ASAT is currently integrated into a Beta version of a production compiler [11] run-time environment from Kuck and Associates. The performance of the Guide compiler implementation of ASAT when multiple processes are executed is presented in [31].

In summary, the previous and concurrent work makes a very strong case that some form of software-based thread adjustment is necessary for these systems to handle a diverse dynamic load efficiently. However, none of the earlier or current approaches were sufficiently well developed to be deployed in a production environment.

1.6.2 Dynamic Load Balancing

ASAT operates within the general framework of Dynamic Load Balancing. Dynamic Load Balancing is one of the classic topics in the parallel processing area of Computer Science [23]. Hundreds or thousands of parallel processors are not too useful if all but one of the processors is at a barrier waiting for a single processor to complete a computation.

There are three general computing entities which are the object of dynamic load balancing approaches: (1) computations , (2) data , and (3) tasks. The use of these dynamic load balancing approaches often follows computer architectures. Computations are balanced on shared memory systems. The data distribution approach is often used on clusters of workstations and multicomputers. Data distribution load balancing is the preferred technique for SIMD computers. Tasks are balanced on workstation clusters and message passing multicomputers.

1.6.2.1 Balancing Computations

When computations are being balanced, the location of the data which is being used in the computations is not considered in the load balancing algorithms. Data is either present in all of the memories or there is a single shared memory with uniform access characteristics. The general topic of scheduling for parallel loops on shared memory computers is one that is well studied. Many of these techniques are actually implemented in today's production parallel compilers [33]. The basic approach of these techniques is to partition the iterations of a parallel loop among a number of executing threads in a parallel process. The goal is to have balanced execution times on the processors while minimizing the overhead for partitioning the iterations and maximizing the potential for cache reuse. A number of scheduling techniques have been proposed and implemented. An excellent survey of these techniques is presented in [17]. These iteration scheduling techniques include Pure Self Scheduling (PSS), Chunk Self Scheduling (CSS), Guided

Self Scheduling (GSS) [24], Trapezoidal Self Scheduling (TSS) [39,40], and Safe Self Scheduling (SSS) [16,18]. These techniques, their impact on performance and their relationship to ASAT will be discussed in greater detail in Chapter 2.

1.6.2.2 Balancing Data

When data is being balanced, the challenge is to move data among the processors so that each processor operates on its local data and has a balanced workload. Often processors are working in a SIMD or SPMD mode where all of the processors are performing portions of a single computation. The simplest approach to balancing data is direct decomposition. In direct decomposition, the data structures are divided geometrically and the subsets are mapped onto processors maintaining the geometric relationship between the subsets. Each processor is assigned a uniformly sized section of the data structure. It is hoped that by balancing the distribution of the data structure, the computations associated with that data structure will naturally be balanced. The primary contribution of the High Performance FORTRAN (HPF) [12] language is to add directives to FORTRAN-90 which support direct decomposition.

Direct decomposition has a number of advantages including: 1) it is relatively simple to implement, 2) it scales very nicely–with more processors, more grids can be created, and 3) it takes advantage of geometric relationships present in many parallel processors (e.g. 2D Mesh, 3D Mesh, or Hypercube).

The primary disadvantage of direct decomposition is that the overall processor utilization can be very low when patterns of computation times are different for different areas of the data structures. With the right problem direct mapping can perform very well on SIMD or

SPMD system as shown in [3]. Interestingly, the limitation of direct decomposition is one of the problems which HPF-2 [13] is attempting to fix by adding non-uniform data distribution constructs to the language.

There are a number of approaches to non-uniform partitioning data in order to balance computations which are not suitable for direct decomposition including scattered decomposition [22], Orthogonal Recursive Bisection [2], and Eigenvalue Recursive Bisection [Will91], Parallel Simulated Annealing, Parallel Neural Network, Parallel Genetic Algorithms [20], and Real-Valued Indexed Arrays [32]. Unlike direct decomposition, these approaches are not supported by high level FORTRAN constructs but must be accessed using explicit calls to subroutine libraries or are only supported in object oriented languages such as C++ [4].

1.6.2.3 Balancing Tasks

Tasks are units of work which consist of both the data and computations. In the load balancing activity, the self-contained tasks migrate around the system seeking computational resources. Sometimes the tasks which are balanced are completely independent (e.g. an operating system) and other times, the tasks are all part of a single large problem which can be independently computed (such as Monte Carlo trials). Some approaches in this area of load balancing include the V- System[36], Sprite [10], Condor [15], Stealth[14], and Utopia [ZhoZhe93]. A formal model for this type of load balancing was proposed in [5] and [6].

1.6.2.4 Dynamic Load Balancing Summary

Of the load balancing approaches described in this section, the techniques which have the broadest use include:

- Iteration scheduling is in broad use on supercomputers and parallel processors including: Cray, Convex, SGI, SUN, DEC, and others
- Direct decomposition is typically done using HPF, in a subroutine library, or explicitly coded by the programmer. It is used on systems such as the IBM SP-2, MasPar, or networks of workstations (NOW).
- Balancing of unrelated tasks is often used in a distributed operating system or a batch queuing environment such as the Load Sharing Facility [19]. This approach is most prevalent on networks of workstations (NOW).

Within the broad framework of dynamic load balancing, ASAT expands the area of dynamic load balancing using processor self scheduling on shared-memory parallel processors.

1.7 ASAT Contributions

This work contains a number of significant unique contributions. No other solution to the problem of matching the overall number of threads to the number of processors has all of these features:

- Requires no special hardware to support its dynamic load balancing
- Requires no operating system modifications to support its dynamic load balancing

- Does not require that all of the processes running on a system use ASAT
- Does not require any centrally managed information for its operation
- Can be used without any compiler modifications
- Sufficiently well developed to be cleanly integrated into existing production compilers

The most significant contribution is the fact that when ASAT becomes generally available in production parallel compilers, shared memory parallel computers around the world processing a combination of multi-threaded and multi-processing load will immediately experience up to a 100% improvement in throughput.

In summary, ASAT is a straightforward, portable technique which can be implemented on a wide variety of parallel processing architectures and thread environments. ASAT takes a large step in moving the dynamic load balancing capability of the commodity-based parallel processors to be nearly equivalent to the capabilities long enjoyed on the traditional vector/parallel supercomputers.

CHAPTER

2.

THE STATE-OF-THE-ART IN SHARED-MEMORY PARALLEL PROCESSOR RUN-TIME ENVIRONMENTS

In this chapter, we examine the current techniques in use in production compilers on existing shared-memory parallel processing systems. We identify the strengths and weaknesses in the actual environments currently in use around the world for High Performance Computing. First, we examine the general field of choosing a distribution of iterations to threads based on the expected distribution of individual iteration times. Second, we examine how the Convex C-Series vector/parallel supercomputers uses hardware features to enable excellent dynamic load balancing. Finally, we examine the shortcomings of commodity-based parallel processing systems when trying to implement loop-based load balancing on systems experiencing dynamic load.

2.1 Processor Self Scheduling

The most common dynamic load balancing approach for shared-memory parallel processors is called processor self-scheduling. In processor self-scheduling, each processor determines which iterations of a particular loop it will process independently. Processor self-scheduling can be contrasted with a master-slave approach where tasks or messages are placed on a queue by a master process and independently removed by slave processes running in parallel. Consider the following pair of example code segments:

```
C VECTOR ADD
       DO PARALLEL IPROB=1,10000
          A(IPROB) = B(IPROB) + C(IPROB)
       ENDDO


C PARTICLE TRACKING
       DO PARALLEL IPROB=1,10000
          RANVAL = RAND(IPROB)
          CALL ITERATE_ENERGY(RANVAL)
       ENDDO
```

In both loops, all the computations are independent so if there were 10,000 processors, each processor could execute a single iteration. In [25] such a loop with no cross-iteration dependencies is called a DOALL loop. In the vector-add example, each iteration would be relatively short and the execution time would be relatively constant from iteration to iteration. In the particle tracking example, each iteration will choose a random number for an initial particle position and iterate to find the minimum energy. Each iteration will take a relatively long time to complete, and there will be a wide variation of completion times from iteration to iteration.

These two examples are effectively the ends of a continuous spectrum of the iteration scheduling challenges facing the FORTRAN parallel run-time environment.

2.1.1 Compiler Options on the SGI

The SGI compiler has options for programmer-controlled iteration scheduling provided as part of its parallel FORTRAN compiler [33] and [34]. Similar options are typically available on most parallel FORTRAN compilers. The iteration scheduling options for a parallel loop on the SGI include:

- Simple – At the beginning of a parallel loop each thread takes a fixed continuous portion of iterations of the loop based on the number of threads executing the loop.
- Dynamic – With dynamic scheduling, each thread processes a "chunk" of data and when it has completed processing, a new "chunk" is processed. The "chunk size" can be varied by the programmer, but is fixed for the duration of the loop.
- Guided Self Scheduled (GSS) – This is essentially a modification of "dynamic" scheduling except that large "chunks" are taken during the first few iterations, and the "chunk size" is reduced as the loop nears completion.

The two example loops above can be used to show how these iteration scheduling approaches might operate on a four-processor system. In the vector-add loop, simple scheduling would distribute iterations 1-2500 to processor 0, 2501-5000 to processor 1, 5001-7500 to processor 2 and 7501-10000 to processor 3. In Figure 3, the mapping of iterations to threads is shown for the simple scheduling option.

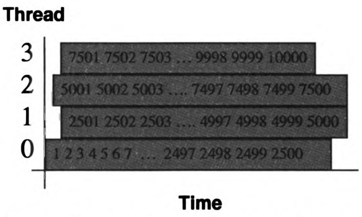

Figure 3 - Iteration Assignment for Simple Scheduling

Since the loop body (a single statement) is short with a consistent execution time, Simple scheduling should result in roughly the same amount of overall work (and time if you assume a dedicated CPU for each thread) assigned to each thread per loop execution. Even though the "work" might be exactly the same across the threads, there may still be a load imbalance at the end of the loop because of cache effect, interrupts, or timesharing. Also, there is sometimes a lag in getting the threads other than thread zero started processing the loop which can lead to some imbalance at the start of a parallel loop [10]. Later, we will precisely measure this thread startup skew on the SGI Challenge.

A further advantage of simple scheduling may occur if the entire loop is executed repeatedly. If the same iterations execute repeatedly on the same processors, the cache for each processor might actually contain the values for A, B, and C from the previous loop execution. The run-time pseudo-code for simple scheduling in the first loop might look as follows:

```
C VECTOR ADD - Simple Scheduled
     ISTART = (PROCESSOR_NUMBER * 2500 ) + 1
      IEND = ISTART + 2499
      DO ILOCAL = ISTART,IEND
        A(ILOCAL) = B(ILOCAL) + C(ILOCAL)
      ENDDO
```

Using the simple approach of giving a fixed number of iterations to each thread, is not always a good strategy. If it were used in the second loop example, it would result in poor load balancing given that the iteration times are long and varying. A better approach is to have each processor simply get the next value for IPROB each time at the top of the loop. That approach is called dynamic scheduling and it can adapt to widely varying iteration times. In Figure 4, the mapping of iterations to processors using dynamic scheduling is shown. As soon as a processor finishes one iteration, it processes the next available iteration in order.

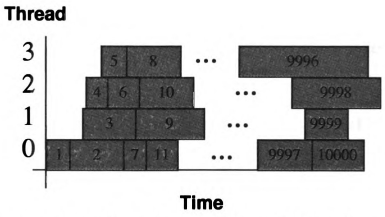

Figure 4 - Iteration Assignment in Dynamic Scheduling

If a loop is executed repeatedly, the assignment of iterations to threads may vary due to very subtle timing issues which randomly affect threads. Any remaining load imbalance is caused by thread start skew and the lengths of the last iterations executed by each thread. The pseudo code for the dynamic scheduled loop at run-time is as follows:

```
C PARTICLE TRACKING - Dynamic Scheduled
     IPROB = 0
     WHILE (IPROB <= 10000 )
       BEGIN_CRITICAL_SECTION
         IPROB = IPROB + 1
         ILOCAL = IPROB
       END_CRITICAL_SECTION
       RANVAL = RAND(ILOCAL)
       CALL ITERATE_ENERGY(RANVAL)
    ENDWHILE
```

While each of these loop iteration scheduling approaches works well for one loop, there is a significant negative performance impact if the programmer were to use the wrong approach for the loop. For example if the dynamic approach were used for the vector-add loop, the time to process the critical section to determine which iteration to process may be larger than the time to actually process the iteration. Furthermore, any cache affinity of the data would be effectively lost because of the virtually random assignment of iterations to processors.

Because the choice of loop iteration approach is so important, the compiler relies on directives from the programmer to specify which approach to use. On the SGI the following directives would be added to insure the right approach was taken in each of our example loops:

```
C VECTOR ADD
C$DOACROSS LOCAL(IPROB),SHARE(A,B,C),MP_SCHEDTYPE=SIMPLE
      DO PARALLEL IPROB=1,10000
         A(IPROB) = B(IPROB) + C(IPROB)
      ENDDO
C PARTICLE TRACKING
C$DOACROSS LOCAL(IPROB,RANVAL),MP_SCHEDTYPE=DYNAMIC
      DO PARALLEL IPROB=1,10000
         RANVAL = RAND(IPROB)
         CALL ITERATE_ENERGY(RANVAL)
      ENDDO
```

Figure 5 shows parallel performance of a simple application, much like the vector-add loop example, on an unloaded 4-CPU SGI with various iteration scheduling options. As expected, the Dynamic and GSS options add overhead to the loops and are the worst performers for this particular loop. Further, the “random” assignment of threads (Dynamic, GSS) to processors results in the loss of cache affinity which occurs when iterations are distributed deterministicaly among processors (Simple).

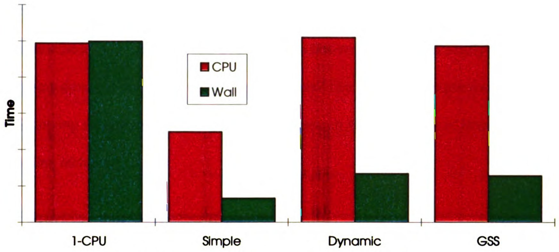

Figure 5 - Impact of SGI Iteration Scheduling Options

Even on an empty system, the critical section overhead and loss of cache affinity caused by using the dynamic iteration scheduling options on the SGI is apparent.

Iteration scheduling does not solve the problem of thread imbalance. Iteration scheduling choices should only be made based on the nature of the iterations. If an application is well suited to dynamic iteration scheduling, it has a chance of acceptable performance when there are excess threads. On the other hand, if an application is well suited to simple scheduling, converting it to have dynamic scheduling will have a significant negative performance impact as shown above. ASAT solves the problem of having more active threads than processors independently of iteration scheduling approach. ASAT enhances all the iteration scheduling approaches because when ASAT is enabled they can depend on threads which have nearly 100% access to a CPU.

While these example FORTRAN loops are two ends of a spectrum, there are a number of different types of processor self-scheduling approaches. Part of the challenge is to balance the cost (or existence) of the critical section against the amount of work done per invocation of the critical section. In the ideal world, the critical section would be free and all of the scheduling would be done dynamically. Convex's ASAP can nearly achieve the ideal using dynamic approaches with relatively small chunk size and expensive hardware. The following section summarizes some of the research into the choice of iteration scheduling technique.

2.1.2 Processor Self Scheduling Research

The primary focus of the dynamic load balancing of loop iterations has been in the study of the assignment of iterations to threads for different loop bodies. The challenge is to balance the overhead of distributing the iterations dynamically and the cost of waiting for an out-of-balance computation to complete.

The following is the breakdown of iteration scheduling techniques:

- Pure Self Scheduling (PSS)
- Chunk Self Scheduling (CSS)
- Guided Self Scheduling (GSS)
- Trapezoid Self Scheduling (TSS)
- Safe Self Scheduling (SSS)

Pure Self Scheduling (PSS) is the same as dynamic scheduling in the SGI compiler as described in the previous section. PSS allocates iterations to a processor dynamically for each iteration. As each processor reaches the top of a parallel loop, it enters a critical section to determine the next iteration to be performed. PSS has the potential for having ideal load balancing because its granularity is a single iteration, but the overhead of the critical section may be prohibitive.

Chunk Self Scheduling (CSS) is an improvement to PSS for situations where iterations are short but still have some variation. In CSS a "CHUNK" of iterations are "grabbed" during each critical section. CSS reduces the scheduling overhead, but can have problems in producing a balanced execution time for each processor [24]. The run-time would be modified as follows to perform the particle tracking loop example using a chunk size of 100:

```
IPROB = 1
CHUNKSIZE = 100
WHILE (IPROB <= 10000 )
  BEGIN_CRITICAL_SECTION
    ISTART = IPROB
    IPROB = IPROB + CHUNKSIZE
  END_CRITICAL_SECTION
  DO ILOCAL = ISTART,ISTART+CHUNKSIZE-1
    RANVAL = RAND(ILOCAL)
    CALL ITERATE_ENERGY(RANVAL)
  ENDDO
ENDWHILE
```

The choice of chunk size is a compromise between overhead and termination imbalance. Again, typically the programmer must get involved through directives in order to control chunk size.

Guided Self Scheduling (GSS) and Trapezoidal Self Scheduling(TSS) are more complicated approaches which dynamically alter the chunk size as the loop progresses to attempt to blend the benefits of PSS and CSS while minimizing their negative impacts.

In GSS [24], the chunk size is varied throughout the execution of the loop. Early in the execution of the loop, large chunks are processed, and as the loop nears termination, the chunk size is reduced. This technique has the advantage that for the early iterations, overhead is minimized because many iterations of the loop execute for each critical section execution. As the loop nears termination, GSS approaches pure self scheduling. In GSS the chunk size is reduced in a geometric fashion as the loop proceeds. One example would be to have the chunk size divided by two each time though the critical section. In Figure 6, one possible iteration assignment for GSS is shown.

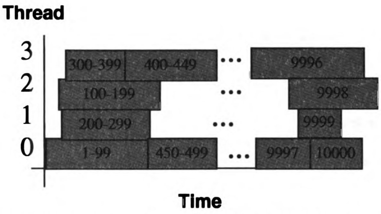

Figure 6 - Iteration Assignment for Guided Self Scheduling

Note that the first time each thread takes iterations, it takes 100 iterations and the second time through it takes 50. A slight modification to GSS called GSS(k) bounds the minimum chunk size to a fixed value (k) so GSS approaches CSS toward the end of the loop. GSS(k) is used in loops with short iteration times to keep GSS from performing one and two iterations per critical section execution during the final phase of the loop. GSS is supported in the SGI Power FORTRAN [33] compiler.

Trapezoid Self Scheduling (TSS) [39,40] improves on GSS. One problem with GSS is that the number of critical section executions required at the end of the loop becomes quite large as the chunk size approaches one. The other problem occurs when there is a widely varying execution time for the work of each iteration. A processor which grabs a large number of iterations at the beginning of the loop may get an inordinate number of iterations which take a long time causing significant imbalance. Another case in which GSS performs poorly is when the time of execution for each iteration is decreasing uniformly. The first large chunk grabbed at the beginning of the loop will have the longest execution time per iteration. TSS reduces the number of iterations in a linear fashion as the loop progresses. TSS achieves a balanced workload under a broader range of execution conditions than any of the other scheduling techniques.

In TSS, the size of the first and last chunk is specified as well as the change in chunk size. The chunk size starts out high and during each critical section, the chunk size is decremented. By reducing the chunk size in a linear fashion, TSS has less overhead than PSS and is more capable of coping with varied work loads than CSS or GSS.

An important aspect of TSS is the choice of the starting, ending, and incremental chunk size. To determine the optimum for these values, some advance knowledge of the iteration times is needed. However, there are some conservative values which can be chosen without pre-knowledge of loop execution time for the starting, ending, and incremental chunk size which yield good results for a wide range of loops. TSS has been implemented in a Beta version of a production compiler [11].

The goal of Safe Self Scheduling (SSS) [16] is to assign each processor the largest possible number of iterations as its initial chunk size and then perform the rest of the loop in a simple self scheduled mode. If the initial assignment of iterations to the processors accurately reflects the actual execution time, SSS can perform excellent load balancing.

2.2 Dynamic Load Balancing on Convex Vector/Parallel Supercomputers

Each of these iteration scheduling approaches has its strengths and weaknesses based on the cost of the critical section and loss of cache affinity. Both problems can be minimized with sufficient hardware investment. The Convex C-Series (C240, C3240, C3480, C3880, C4xx) vector/parallel supercomputers have solved both of these problems with extra hardware.

Convex computer systems are used as departmental and central site computers for many numerically-intensive computing applications. These systems have a maximum of 4 or 8 CPUs and extensive parallel processing support in hardware called Automatic Self-allocating Processors (ASAP) [7]. No cache is used and the memory subsystems have a large number of banks. With these hardware features, the Convex can use dynamic scheduling for every parallel loop. Because the Convex systems have vector registers with 128 elements, it is quite natural to schedule most parallel loops with a chunk size of 128 for maximum performance and nearly ideal load balancing.

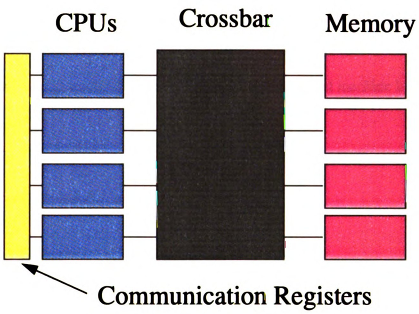

Figure 7 - Convex C-Series Architecture

The ASAP processing in the Convex C-Series systems is made possible because of an architectural feature called "Communication Registers" (Figure 7) which are shared by all of the CPUs. These communication registers allow a multi-threaded process to create, delete, or context-switch threads with minimal performance impact. Using this hardware, the compiler can parallelize loops without regard for the number of threads which will actually execute in the parallel loop. An idle CPU can dynamically create a thread and enter a parallel computation with low overhead.

This hardware support allows users to compile their applications assuming a generalized parallel environment regardless of whether or not there will be enough resources at run-time to execute with multiple CPUs. One significant benefit of ASAP is that a long running job that is compiled to run in parallel can "soak-up" idle cycles as load changes.

This flexibility allows a parallel/vector computer to be nearly 100 percent utilized over long time periods.

ASAP capabilities are accessed using special machine language extensions. In the following example, a small FORTRAN loop with its associated machine language is shown.

```
      DO I = 1,256
        A(I) = B(I) * 3.14
      ENDDO

     snd.l    s0,0x8001      ; Communication Register ****
     spawn    L3,sp          ; Post Request for Help ****
L3:  ldw      #0,a5          ; Nice constant
L5:  inc.w    0x8001,a5      ; Increment Comm Register ****
     bra.f    L5             ; Make sure we get it
     le.w     1020,a5        ; Loop Termination
     jbra.t   L6             ; Get Out
     ld.w     B(a5),s0       ; Load value
     mul.s    3.14,s0        ; Multiply
     st.w     s0,A(a5)       ; Store
     jbr      L3             ; Jump Back up
L6:  join                    ; Drop back to 1 CPU ****
```

In the above example[1], the instructions marked with asterisks (****) are the instructions which access the communication registers. The "snd.l" instruction stores a value into a communication register and the "inc.w" instruction increments the communication register. The "spawn" instruction indicates the beginning of a parallel loop and requests that any free processors create a thread and enter the loop. At run-time if there is only one CPU available, the ASAP "spawn" will be executed, but the ASAP hardware will not

[1] This example was taken from Convex training materials, dated 1988.

create any additional threads. If additional CPUs become available while the loop is being executed, new threads will be created and enter the computation at the "spawn" point. The run-time environment uses the global communications registers for iteration variables in parallel loops regardless of whether the loop is being executed on one CPU or several. Accesses to these communication registers are implicitly synchronized across processors so there is no need for a separate critical section for access to these shared values.

The "join" instruction indicates the end of the parallel loop. As each thread finishes its last iteration, it executes the "join". For all threads except the last thread, the join destroys the thread. When the last thread executes the join, the join is a no-op and the thread continues on the serial portion of the code.

Because the ASAP instructions and the communication registers are so fast, the single CPU performance of an ASAP application is not much slower than the performance of the application compiled for a single CPU. Using ASAP, the number of processors and threads assigned to the application depends on the system load. As the load changes dynamically, the number of processors and threads assigned to the parallel code in an application changes as the loop is executed.

When ASAP is used on these systems, it is very easy to keep the system 100% utilized under dynamic load conditions. Parallel applications can get the best possible time to solution (under current load) while other programs continue to use the system effectively.

When an ASAP job runs concurrently with a randomly changing load on a four processor Convex C-240, ASAP automatically adjusts the number of threads in use on the system so that there are never more than four threads overall. If there are more than four jobs running, the ASAP job only uses a single thread. ASAP works to maximize overall system load by allowing the parallel jobs to use all the available cycles which remain after yielding to the single processor jobs (assuming equal priority).

Figure 8 shows graphically how this resource sharing occurs. In this figure, the horizontal axis is time and the vertical axis is the percentage of the four CPUs in use. In the first graph, a parallel application uses all four CPUs because there is no other load on the system. The second graph of the figure shows several unrelated serial jobs executing on an otherwise empty system. Each job has a different start time and duration. While this job is running, at times there are no CPUs in use and at other times, there are 3 CPUs in use. Because of the variations in load, the system is not 100% utilized over the indicated time period. This collection of jobs is the "load" that is put on the system. In the third graph, the parallel ASAP application is run at the same time as the other serial code. The ASAP hardware automatically adjusts the usage of the parallel ASAP job to match the load on the rest of the system. The ASAP job "soaks-up" the available cycles resulting in high overall utilization under the dynamic load. In addition, the parallel job does not slow down the serial jobs.

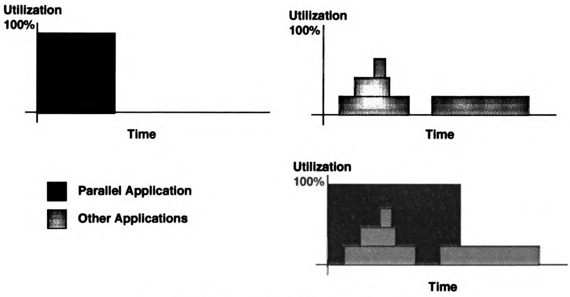

Figure 8 - The Operation of ASAP Under Load

The time to solution for the parallel application when run with the other applications is affected by the load on the system, but the total CPU use by the parallel application is approximately the same as on an unloaded system. Under light load, the parallel application grabs as many free processors as it is capable of using. It should be noted that the number of processors an application is capable of using can vary from loop to loop (see operating point later in this chapter). When there is a heavy load on the system, the scheduler does not assign more than one CPU to the ASAP application (assuming the ASAP application does not have a fixed higher priority than the other applications). In some sense, the parallel ASAP application politely allows the other applications to use the processors they need.

Hardware approaches have one advantage over ASAT and other software-based, thread-adjustment solutions because they have the ability to adjust threads *while* a parallel loop is executing. The details of this process are rather complex and won't be covered here.

However, because of this, hardware based approaches do *not* depend on the existence of the serial portion of the execution profile to accomplish dynamic load balancing.

There is a disadvantage to the hardware approaches when compared to the software solutions such as ASAT, LLPC, or Process Control. The hardware approaches are forced to use some form of dynamic iteration scheduling with a relatively small chunk size to cope with dynamic load. With small chunk sizes the overhead of the critical section can become significant. Simple scheduling, on the other hand, has no overhead *during* the processing of the loop because it completely divides the iterations among the threads *before* the loop starts. Simple iteration scheduling is ruled out for hardware approaches because the number of threads which will actually execute the loop is unknown before the loop starts, and the number may dynamically change throughout the execution of the loop. Because there is a very large number of parallel loops for which simple iteration scheduling is the proper approach, this disadvantage is not a trivial one.

2.2.1 Performance of the Convex on Parallel Jobs

To test the performance of ASAP on the Convex C-240, a simple, parallel computation will be used as the benchmark application. The kernel for these tests is as follows:

```
C$ DO_PARALLEL
        DO J=1,100000
            A(I) = B(I) + C(I) * D(I)
        ENDDO
```

Figure 9 shows the performance of the code with several compiler options and load scenarios. The first bars show the CPU and wall time for the application on a single CPU. The second set of bars shows the performance of the same application on four empty

CPUs. Since the application makes good use of the parallel resources, its CPU time is roughly the same and its wall time is reduced by 75%. The third bars, labeled “Other”, indicate the overall time for a set of single-threaded applications spaced out over time. These applications do not generate enough load to completely utilize the four processor system. These applications arrive randomly and execute for a random amount of time. The wall time and CPU time shown in the chart is the aggregate duration for all of the jobs.

The fourth bar shows the “Ideal” CPU and wall time for the combination of the parallel job and the set of single-threaded jobs running simultaneously assuming perfect load balancing on four CPUs. Note that while CPU time increases additively, the wall time does not increase from the “Other” bars to the “Ideal” bars. This lack of an increase is because on the four CPU system, there are enough “spare” cycles while the “Other” job is running for the parallel job to execute to completion. The ideal performance would only occur if the ASAP job could “soak up” the free cycles while the “Other” jobs were running without adding a great deal of overhead. The last bar shows the actual performance achieved on the Convex C-240 when the jobs are run together.

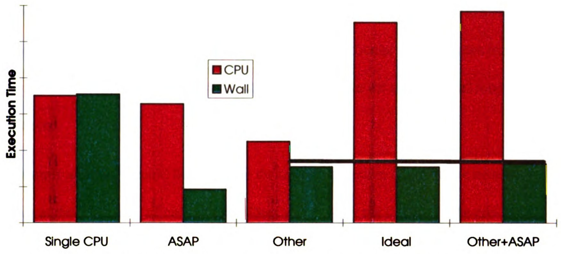

Figure 9 - Performance of the Convex Under Load

In the actual run using ASAP and the "Other" jobs, both the CPU time and the wall time are about 1.05 times longer than the ideal time. This small difference shows that ASAP is very effective in maintaining good utilization when faced with a combination of a compute job with other unrelated load. Long term experience has shown that this excellent load balancing capability with low overhead (~5%) is maintained across a wide range of load scenarios consisting of various mixes of computer and interactive processes.

2.3 FORTRAN Run-Time Thread Management

Because bus-based commodity parallel processors have no hardware support for the dynamic creation of threads, a parallel application must depend on the operating system to manage its threads. Before a loop can begin execution with a number of threads, the threads must be activated to join the parallel loop. In this section we examine how this is thread activation is accomplished at run-time. Thread management is only mildly related to iteration scheduling (Section 2.1). Iteration scheduling assumes that some externally controlled number of threads are participating in the loop and distributes the iterations among those threads. Thread management deals with the issues related to activating threads which are to participate in a parallel loop. Of particular interest is the number of threads which are made available to the iteration scheduler for each parallel loop execution.

One simple approach would be to invoke the operating system to create the necessary threads at the beginning of each parallel loop, and destroy the threads at the end of the loop much like the Convex ASAP processing. Unfortunately, calling the operating system at the beginning of each loop would incur excessive overhead. In addition to the operating system overhead, the arrival of these newly created threads to enter the parallel computation is often skewed significantly [10]. To avoid this overhead and late thread arrival, the run-time library typically creates its threads once at the beginning of execution and then manages those threads in user space.

This approach which creates a fixed number of threads at the beginning of execution and uses them throughout the duration of the application is referred to as Fixed Thread Scheduling (FTS) throughout this document. The choice of the name "Fixed" emphasizes the fact that the number of threads does not change once the application begins execution.

In this section, we will examine how these threads are managed by the run-time library and the performance of this approach in the face of a dynamic load.

2.3.1 Run-Time Thread Management Details

When a compiled parallel application is executed, the run-time environment starts up a number of threads. The number of threads is often the same as the number of installed processors. These threads are scheduled by the operating system much like UNIX processes. On some systems, these threads are scheduled by the operating system using gang scheduling. When a set of threads is gang scheduled, either all of the threads are executing or all of the threads are suspended[2]. In Figure 10 (Free Scheduling) and Figure 11 (Gang Scheduling) these approaches are graphically compared.

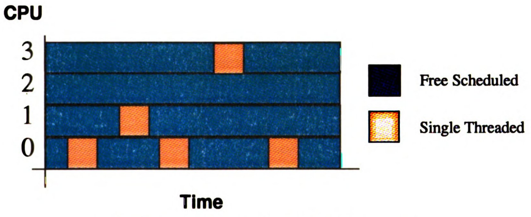

Figure 10 - Free (Non-Gang) Thread Scheduling

If gang scheduling is done in its strictest sense, a significant amount of time could be wasted when a multi-threaded application is timesharing with a single-threaded application as in Figure 11.

[2] This is a simplified definition. In actuality the operating system tries to do its best to get all threads started within a specified time period.

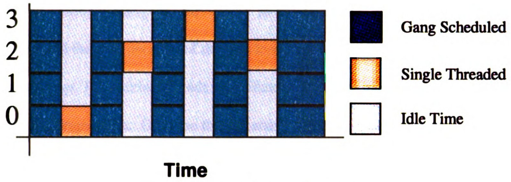

Figure 11 - Gang Thread Scheduling

Regardless of the way the operating system schedules the threads, when the program starts, one thread begins executing the user application while the other threads "spin", waiting for a parallel loop to be encountered. The code which these waiting threads execute is as follows:

```
while ( wakeup == 0 ) ;  // "Infinite loop"
goto beginning_of_loop;
```

The variable *wakeup* is initially set to zero. With a bus-based system, this approach might appear to cause a great deal of unnecessary memory traffic. Actually, each waiting processor ends up with a shared cached copy of the *wakeup* variable with the value zero. Once the caches have been filled, there is no additional memory traffic as these waiting processors execute this tight loop. Some systems [35] have added special hardware instructions to make this wakeup from a spin loop perform even more efficiently.

When a parallel loop is encountered, the variable *beginning_of_loop* is set to indicate which parallel loop is about to be executed and the variable *wakeup* is set to one to activate the waiting threads. Setting ***wakeup*** to one causes all of the cached copies to be invalidated and the next iteration of the spin loop exits as the caches reload with the new value of ***wakeup***. The waiting threads immediately notice the change, exit the spin loop and join the computation. Once the threads arrive at the loop, they each determine the appropriate iterations to process using the iteration scheduling technique chosen for this particular loop and begin processing those iterations.

When the loop completes, each executing thread enters a barrier and when all of the threads have entered the barrier one thread continues executing in serial. The remaining threads again execute the spin loop waiting for the next parallel loop. This approach results in extremely quick thread activation times.

To demonstrate the timing of these operations, the following FORTRAN loop was executed on the SGI:

```
        IN = WALLTIME()
C$DOACROSS LOCAL(J),SHARE(MIDDLE),MP_SCHEDTYPE=SIMPLE,CHUNK=1
        DO J=1,4
          MIDDLE(J) = WALLTIME()
        ENDDO
        OUT = WALLTIME()
```

The entire source code for this program is included in Appendix A. Because the scheduling type is SIMPLE, the consistent mapping of a CPU to an iteration of the J loop is forced. By checking the elapsed time between the IN and MIDDLE times, one can determine the time for a thread to arrive in the parallel section of a loop (spawn time). By checking the elapsed time between the MIDDLE and OUT, the time spent processing the loop-end barrier can be determined for each thread. Once the overhead for the WALLTIME calls is removed, this loop does no work and we should be able to measure the performance of the parallel loop startup and completion. This loop will be used to measure how fast a thread can be brought into the loop from a spinning state. Later we will measure how these timings change when a loop is executed when there are more threads than processors.

In the Figure 12, the performance of this loop is measured on an empty system. The loop was executed a number of times and the average values are reported in this figure.

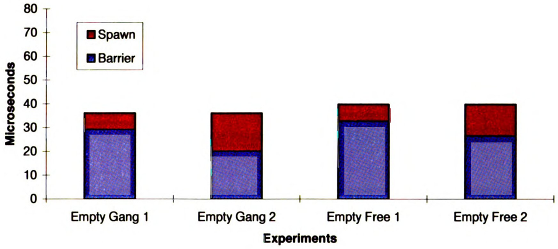

Figure 12 - Thread Timing Results

This figure shows the performance of the first and second threads (out of four threads) using free scheduling and gang scheduling. Using Gang Scheduling, the entire loop takes 36 microseconds and using free scheduling it takes 40 microseconds. Thread one is the thread which wakes the other threads up so it is the first to arrive in the body of the loop. Thread one takes 7 microseconds to go from the serial code to the body of the loop in both cases. Thread two takes 16 microseconds to arrive in the body of the loop. This difference shows how quickly the "helper" threads can arrive from their spin loops. The barrier at the end of the loop takes somewhat longer in general than the spawn process. Also, since thread one arrives quicker, it completes its work quicker and as such spends more time at the barrier waiting until thread two arrives to terminate the loop. The performance for threads three and four look identical to thread two.

While the spin-wakeup approach results in fast loop startup times, a problem is that it assumes that all the available threads are actually *executing* the spin loop at all times when they are not participating in a parallel loop. There are two reasons that a thread might not be executing the spin loop when a parallel loop is encountered. The first reason is that the thread may have decided to voluntarily put itself to sleep because it has waited too long in the spin loop code at a barrier or waiting for work. The second reason is that the operating system may have suspended a thread involuntarily because of other unrelated load.

A spinning thread waiting for work will voluntarily suspend itself to minimize the wasted CPU time in case an application is about to spend a significant time running serial code. The pseudocode for this is roughly as follows:

```
while ( wakeup == 0 ) {
  counter = 10000;
  while (counter > 0 ) {
    if ( wakeup == 0 ) break;
    counter --;
  }
  if ( counter == 0 ) release_cpu();
}
goto beginning_of_loop;
```

The *counter* value is typically controllable by the application programmer. The programmer may also be able to suppress the *release_cpu* behavior altogether.

Interestingly, by suppressing the release of the CPU, the programmer gets marginally better performance for their application. Of course, the CPU time spent spinning is not accomplishing any real work and is not available for other users. Often, benchmark runs are executed with the *release_cpu* behavior suppressed. When a thread voluntarily gives up the CPU it records this fact so when the serial thread encounters a parallel loop, it can request that the operating system reschedule the suspended thread.

With the potential that a waiting thread might suspend itself, the pseudocode for starting a parallel loop is as follows:

```
If any threads have put themselves to sleep, wake them up
Store starting location of parallel loop
Notify spinning threads via wakeup
Distribute iterations to threads
Process assigned iterations
Perform Barrier
One thread continues execution while others wait for the next parallel
loop to be encountered
```

If a spinning thread has been suspended involuntarily by the operating system, the startup latency is much larger because the serial thread is not aware (and is not even allowed to awaken) the spinning thread which has been suspended. The loop startup code simply assumes the spinning threads will join in a few microseconds. The operating system must re-schedule the suspended thread before the thread can execute, detect the changed *wakeup* variable, and join the computation. In the worst case, this latency can be on the order of an operating system time slice. Thread arrival skew can cause a non-dynamic iteration scheduling algorithm to appear to have very unbalanced load as will be shown later.

While this approach seems to waste the CPU in spin loops, a program which is well suited to parallel processing when properly tuned will typically run in parallel a large part of the time so the time that processors spend spinning at the end of parallel loops should be minimized on an unloaded system. The excellent speedup on the unloaded SGI Challenge and Convex C-240 for some unmodified end user applications is compared in Figure 13. It is interesting to note how closely the four CPU SGI Challenge with much less hardware support (and much less cost) tracks the speedup of the four-CPU Convex C-240 vector/parallel supercomputer on an empty system.

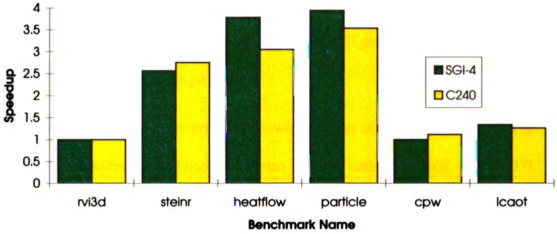

Figure 13 - Comparing Speedup on SGI Challenge and Convex C240

In Figure 13 the horizontal axis is the name of a series of benchmarks used in the procurement process at Michigan State University in 1993. None of the programs were modified, and the vendor compilers performed automatic parallelism detection. The jobs were all run on an empty system. The jobs were first run on a single CPU for both vendors and then on a four-CPU system with parallelism enabled. The single CPU runs were assigned a value of one and the speedup for the parallel runs were computed.

Interestingly three out of six of the applications saw a speedup of 3-4 on both systems and the other three experienced no speedup. The most interesting aspect of these results was that there was no particular advantage of one architecture over the other when the systems were empty.

2.4 Problems with Dynamic Load

Computer systems similar to the SGI Challenge perform very well when there is no other load on the system and each thread has exclusive access to a CPU. The excellent speedups shown in Figure 13 were achieved with otherwise empty systems. In an earlier section, we showed how well the Convex vector/parallel supercomputers handled a dynamic load. Unfortunately, the performance of the SGI gets much worse when there are more threads than available processors.

The problem of matching the overall system-wide number of threads to the number of processors was studied on an Encore Multimax [38] and later on the SGI 4D/340 [37]. They identified and measured a number of the major problems with having more threads than processors including:

- Preemption during spin-lock critical section,
- Preemption of the wrong thread in a producer-consumer relationship,
- Unnecessary context switch overhead,
- Corruption of caches due to context switches, and
- Operating point effect.

While most of these are well understood, "operating point effect" is somewhat more subtle. Some applications can make better use of extra processors than others. Highly parallel applications such as "particle" in Figure 13 can make use of additional processors at nearly 100% efficiency. That is, for each processor added, the time-to-solution is reduced linearly. Other applications such as "lcaot" in Figure 13 with poor speedup make inefficient use of additional processors. When an application is executing which makes inefficient use of additional processors, and there is no other load, there is no harm in utilizing the otherwise idle processors as long as there is some marginal benefit. However, if the program is operating on a system with other multiprogramming and/or multi-threaded load which can make more efficient use of the processors, overall system efficiency is improved if the program with poor speedup operates with fewer threads. When an application is using more threads than it can use efficiently, it is above its "operating point" [37].

As the speed of the CPU's has increased and the increasing reliance on data resident in cache, the problem of a context switch corrupting cache has become an increasing performance impact. In [21], when a compute-bound process was context switched on a cache-based system, the performance of the application was significantly impacted for the next 100,000 cycles after the process regained the CPU. The context switch still had a small negative impact on performance up to 400,000 cycles after the context switch. In many situations, the cache impact dominated the overall cost of a context switch.

2.4.1 Loop Performance Under Load

In this section, the performance of the simple loop is studied when a single CPU-bound job is added to the system. The same experiments are run as shown in Figure 12 except with the addition of load. These experiments are run with the standard SGI compiler options which use spin without *release_cpu* at the end of a loop.

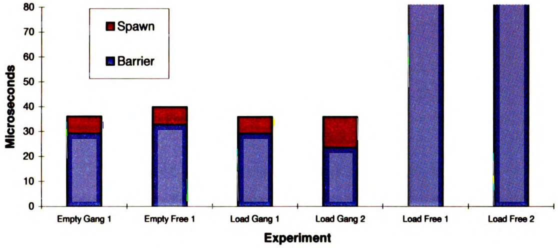

Figure 14 - Loop Timing Under Load

In Figure 14, the spawn and barrier times are shown for a gang scheduled and a free scheduled[3] application. All the experiments use Fixed-thread scheduling in that four threads are used for every parallel loop. Both the first and second thread are shown in the above figure. The performance for the unloaded system is included for reference. When gang scheduling is used, the change in performance is effectively unmeasurable when load is added. However, when load is added to the free scheduled program, the performance suffers dramatically. The following figure changes the y-axis to a

[3] "Free-Scheduling" indicates the lack of gang-scheduling. This term will be used throughout the remainder the document to refer to applications which use free scheduling with a fixed number of threads.

logarithmic scale so that the performance of the loop under free scheduling can be observed.

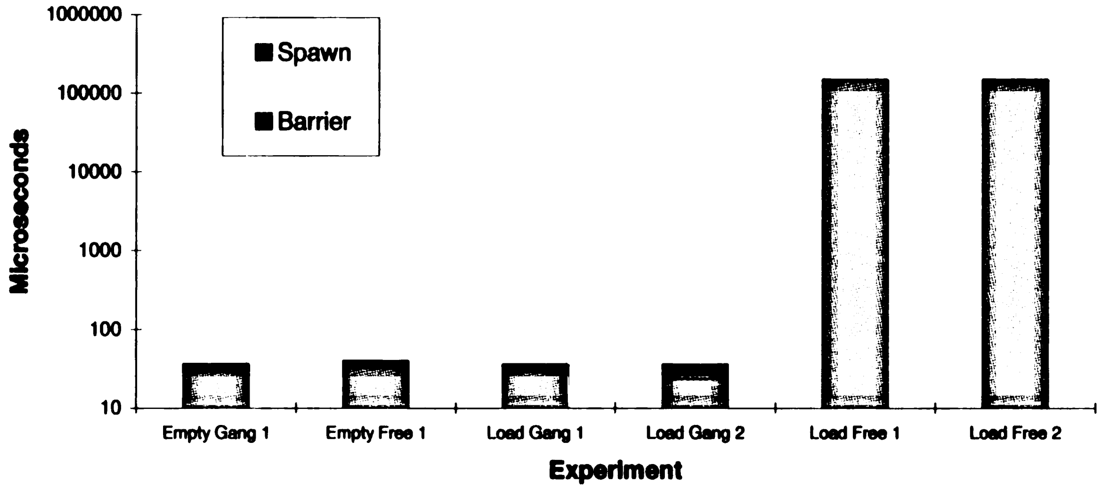

Figure 15 - Loop Timing Under Load (Log Scale)

In Figure 15, the free thread scheduled application performs over three orders of magnitude slower. Interestingly, thread one still enters the loop in 14 microseconds although at the increased scale, it can no longer be seen. The second thread averages 10583 microseconds (or 0.1 seconds) for its arrival. The bulk of the time is spent in the barrier for all threads with each thread displacing the other spinning threads to spin. Further, because there are four CPUs and only one single load thread, three threads are usually active in the application. These threads quickly go to the barrier and spin waiting for the arrival of the fourth thread. Once the fourth thread arrives, it quickly completes its 15 microseconds of work and goes to the barrier.

To further understand the values which make up this average, the following figure shows the performance of thread two (the last bar above) for a selected number of individual iterations. The vertical axis is again log scaled.

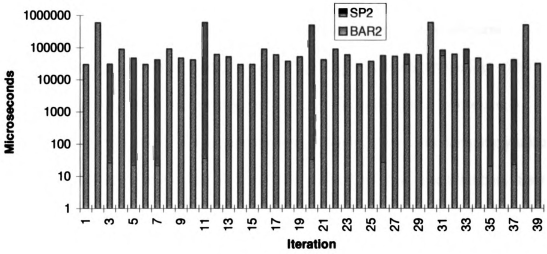

Figure 16 - Individual Iteration Timing - Thread 2/Free (Log Scale)

In Figure 16, the cost for the spawn and the barrier are shown for the second thread with free scheduling on a further expanded scale. The barrier contributes the majority of the performance impact. In some iterations the spawn time (or thread arrival time) is significant and in a few iterations there is a negative performance impact from both the thread arrival and barrier time.

2.4.2 Parallel Applications and Load on the SGI

The above tests focus on the potential negative impact when gang scheduling is not used. However, in a real application on the SGI, gang scheduling is available and loops execute longer in parallel which reduces the impact of loop startup and termination performance. In my example, cache effect, context switch overhead and other factors impact performance. To measure these negative performance impacts on a typical application, the multiple job experiment performed earlier on the Convex in Figure 9 is also performed on a 4-CPU SGI Challenge system. This experiment is run using the SGI compiler with gang scheduling turned on. Figure 17 shows the results of that experiment. As on the Convex, the first set of bars show that the application code parallelizes automatically without any user modifications using simple iteration scheduling. The second bars represent set of single-threaded applications with random arrival and duration. These applications do not generate enough aggregate load to completely utilize the system which is why the wall time is not 25% of the CPU time. The third bars show the CPU and wall time for the ideal combination of the two codes assuming perfect load balancing on four CPUs. As with Figure 9, the wall time does not increase from the "Other" bar to the "Ideal" bar because there are enough "spare" cycles while the load job is running for the parallel job to execute to completion. The next-to-last bar, "Other+Simple", shows the actual performance achieved on the SGI Challenge when the jobs are run together. However, unlike the Convex, the system performs much worse than ideal when both jobs are run simultaneously. The wall time for the combination job is 1.68 times longer than ideal, and the CPU time of the combination job is 1.76 times longer than the ideal CPU time. In fact, with the two jobs running simultaneously, the SGI performs *worse* than if you ran the jobs sequentially using a batch queue as shown by the last set of bars in the figure.

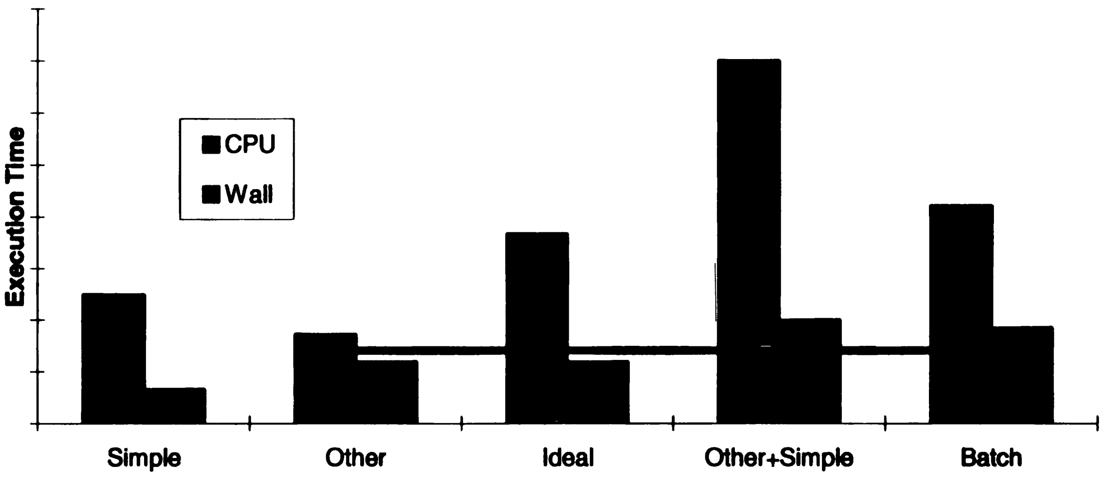

Figure 17 - SGI Performance with Load

When both the parallel and serial jobs are running simultaneously, the parallel application experiences poor performance and also slows down the non parallel applications as well. The general problem which causes the poor performance for both jobs when they are run simultaneously can be explained using Figure 18. When the SGI system is otherwise empty, either the parallel job or the serial jobs will make good use of the resources. However, when all of the jobs are run at the same time, at times there are more active threads than processors. All the threads must be timeshared across the available processors. This sharing has a significant negative impact on both the parallel and serial jobs. The parallel job experiences a wide range of effects as described earlier. The serial job slows due to context switches, cache loss and the simple loss of CPU due to timesharing. The worst part of this situation is that the parallel work delays the serial work from getting completed, extending the length of time the system is operating inefficiently.

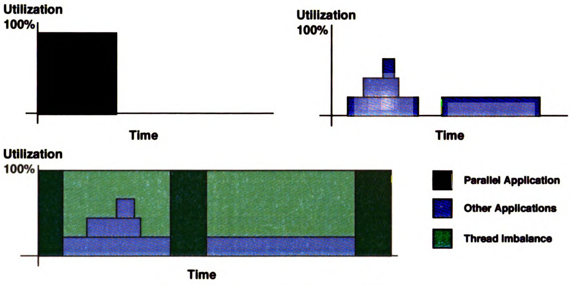

Figure 18 - Impact of Thread Imbalance

The only reasonable choice on such as system (other than the dynamic thread adjustment described herein) is to run parallel compute jobs on an otherwise empty system. These incompatible workloads can be separated using a batch queue, a predetermined schedule for usage during different parts of the day, or some other external management policy.

2.5 Summary

There are many techniques to perform dynamic load balancing within a single application using iteration scheduling on an otherwise empty SMP parallel processor. These techniques assume that every thread will have a dedicated processor. There is an excellent system-wide dynamic load balancing solution available on expensive parallel/vector supercomputers which allows those systems to maintain 100% utilization over the long term. Unfortunately, there is no production quality solution for low-cost SMP systems which provide overall dynamic load balancing.

CHAPTER

3.

AUTOMATIC SELF-ALLOCATING THREADS

In this chapter we explore the mechanisms which Automatic Self-Allocating Threads (ASAT) uses to maintain the overall balance of threads across an entire system.

3.1 ASAT Design

There are a number of goals of the ASAT design:

- Must be easily integrated with existing compilers and run-time libraries
- Must not require operating system modifications
- Must not use high-cost system calls
- Must have minimal performance overhead (should compare favorably to the 1-4% cost of Convex ASAP hardware).

The general goal of our ASAT is to eliminate thread imbalance by detecting excess threads, and then dynamically reducing the number of active threads to achieve balanced execution over the long term. In this way, multi-threaded ASAT applications will experience thread imbalance only during a small percentage of the execution time of the application. To implement ASAT on a parallel processing system, there are a number of problems which must be solved in the ASAT run-time library. The most important are:

- Detecting if too many active threads exist.
- Detecting if too few active threads exist.
- Adjusting the number of threads.

ASAT takes advantage of the basic loop structure shown earlier in this document. Under Fixed Thread Scheduling (FTS) the beginning of the parallel loop activates the same number of threads each time it is executed over the duration of an application. When ASAT is used, the run-time library will activate the appropriate number of threads based on the overall load on the system. The goal is to execute with the number of threads which match the available processors.

ASAT only adjusts the thread count when the applications are running single-threaded. Given the current state of automatic parallelizing compilers, this parallel segment time duration tends to be shorter rather than longer. As compilers improve and applications are re-written, the length of the parallel segments should increase.

It is acknowledged that some programs spawn once and run in parallel for very long periods of time. These applications will need to be modified to spawn more often to participate in ASAT. Such applications are often hand coded with explicit parallelism and as such, a modification to support ASAT may not be a great burden. Most applications which use compiler-generated parallelism will not exhibit this behavior.

There were two versions of ASAT which were developed. The first version [26] was developed using a dynamic scheduled loop. The second version [29] was initially developed on the Convex Exemplar and later ported back to the SGI. The second version is the basis for the ASAT built into the Kuck and Associates Guide compiler.

3.2 ASAT Version 1 Implementation

The first version of ASAT was developed on the SGI and described in [26]. This approach took advantage of an expected increase in thread arrival skew at loop startup under loaded conditions. It was assumed[4] that any operating system call would have excessive overhead and so the idea was to completely determine the relative thread imbalance in user space.

To determine if there were excess threads, the following loop was executed as the ASAT evaluation:

```
C$DOACROSS LOCAL(I),SHARED(WHICH),CHUNK=1,MP_SCHEDTYPE=DYNAMIC
       DO I=1,1000
         WHICH(I) = MP_MY_THREADNUM() + 1
       ENDDO

       DO J=1,4
         THREAD(J) = 0
       ENDDO
       DO I=1,1000
          THREAD(WHICH(I)) = THREAD(WHICH(I)) + 1
       ENDDO
```

The iterations in the parallel loop were scheduled using the DYNAMIC approach so the assignment of iterations to threads as recorded in the *WHICH* array depended on the order of the threads reaching the critical section to update the iteration variable *I*. After this loop, the maximum and minimum of the four values in the *THREAD* array were compared.

[4] Mistakenly as will be shown later.

To make ASAT very sensitive to changes in the other load on the system, gang scheduling was turned off and the priority of the ASAT jobs was reduced below that of the other load. The fundamental assumption in ASAT is that the ASAT job is soaking up only excess cycles. If an ASAT process detects excess threads, it assumes that it is the process which should give up a thread. By turning off Gang Scheduling and lowering priority, if there are too many threads, the ASAT job will have a lower probability of getting "clean" cycles on all of its threads. The reduced priority and free scheduling increases the probability that the parallel loop will execute unevenly. Once ASAT has dropped sufficient threads, the remaining threads each have a dedicated CPU and the slightly lower priority[5] of ASAT processes in the steady state does not affect the actual processor allocation and the ASAT job ends up with clean cycles.

To test the ability of the above loop to detect load, the loop is run on an empty system and on a system with one other process and the thread differential is measured. The following figure shows the distribution of the iteration differential for 500 executions of the loop with and without load. Since the loop has 1000 iterations, the iteration differential ranges from 0-1000. An iteration differential of 1000 means that one thread processed all the iterations in the loop and the other three threads processed no iterations at all. An iteration differential of zero indicates that each thread processed exactly 250 iterations.

[5] On the SGI under IRIX there are three priority classes and within each class there are relative priorities. On the SGI, ASAT operates at the lower end of the middle priority class. If the lowest priority class were used (non-decreasing) the performance would be very poor even when there is no other load on the system.

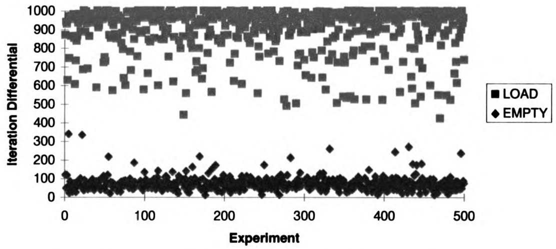

Figure 19 - Iteration Differential In Loaded and Unloaded Conditions

In Figure 19 when there is no load, all four threads process roughly 250 iterations and the difference between the maximum and minimum is relatively small. The average thread differential was 70 iterations with a standard deviation of 30. During thread imbalanced conditions, the thread startup skew would be so severe that it was not uncommon to see all of the iterations processed by a single thread. The average thread differential under load was 900 with a standard deviation of 125.

The largest problem with this approach was that the 1000 iteration loop took an average of 6000 microseconds or 0.006 seconds under ideal conditions. This time could be a significant overhead given that we showed earlier that the shortest loop took roughly 40 microseconds. Interestingly the 1000 iteration loop took an average of 0.15 seconds under loaded conditions.

With this relatively large overhead, the key issue for Version 1 of ASAT was choosing the appropriate serial portion of code in which to insert the ASAT evaluation. One did not want the evaluation to be performed too often.

The other challenge with the Version 1 approach was deciding "when to increase the threads?". The problem is that when you mistakenly add a thread, performance was so poor for the next parallel loop that significant time was wasted. A crude solution was to consult the UNIX "uptime" command every 30 seconds.

Even with its technical limitations, properly used, ASAT Version 1 work showed remarkable load balancing ability as is shown in [26]. Some of the performance results using Version 1 are presented in sections 4.2 and 4.3.

3.3 Timed Barrier Performance Study

When ASAT was ported to the Convex Exemplar [8],we wanted to take advantage of the high-resolution real-time clock mapped into user space. This clock allowed time to be determined without a system call. Furthermore, we observed that what the loop in Version 1 of ASAT was actually measuring was thread arrival skew. In unloaded conditions one thread would arrive and process lots of iterations long before any other thread would arrive.

We looked at a barrier as a simpler measure of thread arrival skew. The idea was to time the first thread arriving at a barrier to the last thread leaving the barrier. It was hoped that the same thread skew would manifest itself as it did in the Version 1 loop.

The following experiments time a barrier and measure the "worst case passage time", that is, the elapsed time from when the first thread arrives to when the last thread leaves.

In the following figure, the barrier passage on an empty system is shown for gang scheduled and non-gang scheduled situations on the SGI Challenge using the SGI compiler and the MP_BARRIER call:

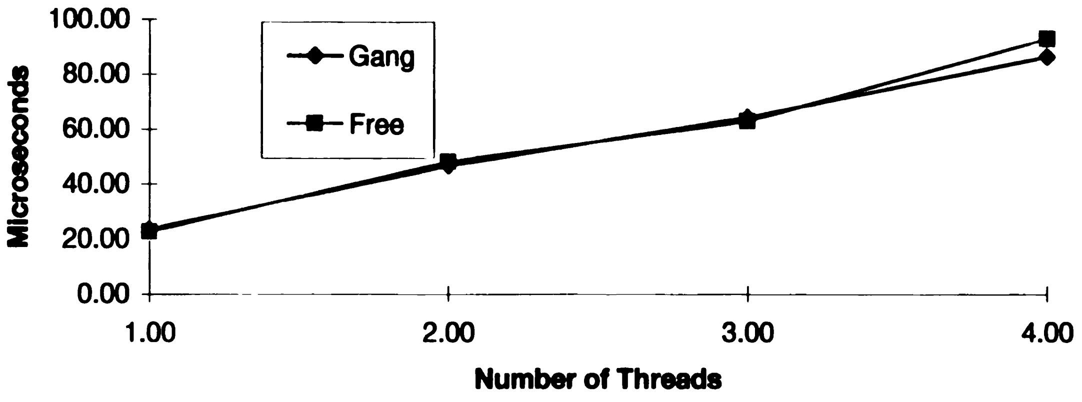

Figure 20 - Barrier Passage Time (scaling)

In Figure 20, the barrier passage time is shown as the number of threads is increased. The SGI Challenge provides a hardware instruction to improve the performance of this type of operation so the performance is quite good ranging from 20 to 90 microseconds for a barrier passage. The different types of scheduling have little impact on the performance of a barrier passage. Also, the cost of a barrier appears to scale at least *O(n)* where *n* is the number of threads.

The key question regarding the timed barrier is whether or not it can detect when there are excess threads. In the next experiment, a barrier test is run under four conditions: (1) gang scheduled on an empty system, (2) free scheduled on an empty system, (3) gang scheduled on a loaded system, and (4) free scheduled with lower priority on a loaded system. The graph below summarizes the results of those experiments. Note that the y-axis is a logarithmic scale.

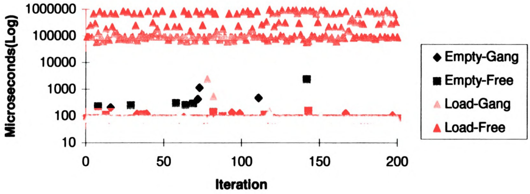

Figure 21 - Timed Barrier Passage With and Without Load

In Figure 21, all of the experiments except free with load maintain approximately 90 microseconds for their barrier passage during the 200 tries. There are some anomalies in all of the first three experiments due to interrupts or context switches. However, when gang scheduling is turned off and the priority is lowered, the average barrier passage time dramatically increases to 0.3 seconds. Furthermore, in none of the 200 iterations in the Load-Free case is the barrier passage less than 10000 microseconds. There is roughly three orders of magnitude between a loaded and unloaded barrier passage.

These measurements indicated that the timed barrier passage would be a reliable indicator of load as long as gang scheduling was turned off and the priority was reduced.

In further testing on the SGI, it was determined that the operating system can be called to read the real time clock in roughly 14 microseconds. Once the timed barrier test was proven to be useful on the Convex Exemplar, and the SGI IRIX operating system could quickly provide the time, the timed barrier approach was integrated into the SGI Version 2 of ASAT.

3.4 ASAT Version 2 Implementation

The Version 2 implementation of ASAT on all platforms uses a timed barrier test to detect thread imbalance on the system. Before the barrier test is performed, the clock is called to determine the amount of time since the last timed barrier test was run. This elapsed time value can be tuned by the user. It is a good approach because the clock overhead (14 microseconds) is lower than a barrier test (20-90 microseconds) in all cases and a clock call in single threaded code is constant as the number of processors are increased.

The pseudo code for the timed barrier is a follows:

```
static double entering[MAX_THREADS];
static double leaving[MAX_THREADS];

double timed_barrier_test(int THREADS) {
  spawn(THREADS);
  barrier_code();
  first_in = min(entering);
  last_out = min(leaving);
  passage = last_out - first_in;
  return(passage);
}

barrier_code() {
  entering[MY_THREAD] = real_time();
  execute_barrier();
  leaving[MY_THREAD] = real_time();
}
```

The interval between barrier evaluations can be adjusted. The ASAT software is set to only run the barrier test once every 0.5 seconds of elapsed time by default. The ASAT routine could then be called thousands of times per second, but most of the calls would return immediately because the specified time between ASAT barrier tests had not yet expired.

The number of spawned threads is decreased when the barrier transit time indicates a thread imbalance. ASAT has tunable values which determine the values for what is a “bad” transit time and the number of “bad” transit times necessary to trigger a drop in threads.

To determine whether or not to increase the number of threads, the ASAT barrier test is executed with one additional thread, and the barrier transit time is measured. If the barrier transit time indicates that one more thread would execute effectively, the computation is attempted with one more thread. One can think of this as “dipping your toe in the water.” If the number of threads in use has been working smoothly for a while, test with more threads for a single barrier. If this barrier runs well, dive in and run the whole application with more threads. Of course, if the increase in threads results in an imbalance, ASAT will drop the thread count at the next parallel section.

The pseudo code for the ASAT thread adjustment heuristic is as follows:

```
/* The current number of threads */
static int ASAT_THREADS;

asat_adjust_threads() {
 Check Time
 if ASAT_EVAL_TIME has not expired, return
 BAR_TIME = timed_barrier_test(ASAT_THREADS)
 IF (BAR_TIME > ASAT_BAD_TIME) {
   ASAT_BAD_COUNT++;  ASAT_GOOD_COUNT = 0;
 } else {
   ASAT_GOOD_COUNT++;  ASAT_BAD_COUNT = 0;
 }
 IF (ASAT_BAD_COUNT >= ASAT_BAD_TRIG
     and ASAT_THREADS > 1) ASAT_THREADS--;
 IF (ASAT_GOOD_COUNT >= ASAT_GOOD_TRIG
     and ASAT_THREADS < MAX) {
   BAR_TIME =
        timed_barrier_test(ASAT_THREADS+1)
   IF (BAR_TIME < ASAT_EVAL_TIME)
     ASAT_THREADS++;
 }
}
```

The ASAT heuristic is more aggressive about decreasing threads than increasing threads. This is done for 2 reasons: (1) running with too few threads slows you down by a linear factor whereas running with too many threads can cause extremely poor performance and (2) to create a natural hysteresis loop to keep ASAT from repeatedly adding and dropping a thread.

3.5 ASAT Version 2 Tunables

ASAT has a number of tunable parameters which can be used to adjust its performance.

- ASAT_FLAG – This flag disables the operation of ASAT. ASAT can be disabled using an environment variable or by a subroutine call added to the user code. ASAT can be turned on and off by the application during the execution of the application. The default is "YES". If the environment variable is not set, ASAT is not turned on.

- ASAT_EVAL_TIME – This flag specifies the number of seconds to wait between ASAT evaluations. The default is 0.5 seconds.

- ASAT_BAD_TIME – This value is used to determine if barrier passage time is "bad". If the barrier time is less than this value, it is a "good" barrier, and if it takes longer than this value, it is a "bad" barrier. The default is 0.001 seconds. A well-balanced barrier passage is much faster than 0.001 seconds, and an unbalanced barrier passage is much slower than 0.001 seconds.

- ASAT_BAD_TRIG – This flag specifies the number of successive "bad" barrier times before the number of active threads is reduced. The default is 2.

- ASAT_GOOD_TRIG – This flag specifies the number of successive "good" barrier times before ASAT attempts to increase the number of active threads. If the number of threads is the same as the number of processors, ASAT will not increase the number of threads. The default is 15.

3.6 Summary

The core concepts and resulting code for ASAT are relatively simple, and it is ready to be added to any modern automatic parallizing compiler. As will be shown in the next section, the overhead of ASAT is small enough that it can be effectively added to every loop worth parallizing. Furthermore, while ASAT does not require gang scheduling it performs as well as gang scheduling in most cases. ASAT performs as well as or better than Fixed/Free scheduling in all cases.

CHAPTER 4. PERFORMANCE RESULTS

In this chapter, the performance results for ASAT are summarized on two computer architectures. The overall effect of ASAT is explored. Then the effect of the loop iteration count (grain size) and the effect of multiple jobs are studied.

4.1 Measuring ASAT Overhead and Benefits

In this section we quantify the performance benefits and impact of ASAT using our microsecond timed loop as described in Section 2.3.

The first four bars of the following figure (EG1 - Empty System Gang Scheduled Thread 1, EF1 - Empty System Free Scheduled Thread 1, LG1 - Loaded System Gang Scheduled Thread 1, LF1 - Loaded System Free Scheduled Thread 1) are simply the previous results from Figure 14 included for reference. As before, LF1 has a value of 150,000 microseconds and is way above the top of the graph scaled at 80 microseconds. The remaining bars are the performance of the ASAT job on the empty (EA1 and EA2) and loaded (LA1, LA2, and LA3) system.

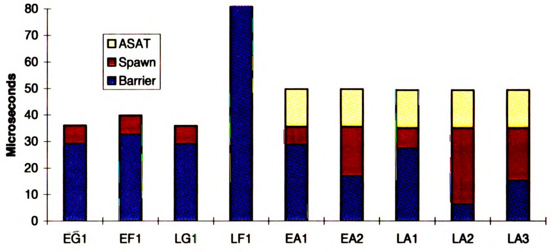

Figure 22 - Loop Iteration Timing with ASAT

In Figure 22, ASAT is first run on an empty system in order to determine its impact. Threads one and two (EA1 and EA2) are reported. As before threads three and four are like thread two. The additional overhead of ASAT increases the overall loop time from 36 microseconds to 51 microseconds. This is exactly the cost (15 microseconds) of a time call. Because the ASAT_EVAL_TIME is set to 0.1 seconds, the barrier (which is only 90 micro seconds) is only run every 2000 iterations, and, as such, contributes little to the average overhead. The thread one/thread two pattern remains in the empty ASAT runs in that the first thread gets into the loop more quickly, but waits longer at the end of the loop because each thread processes one iteration. When load is added, the ASAT loop actually executes like the Load Fixed (LF1) for several iterations (see Figure 23), and then drops to three threads reacting to the load. The performance reported in Figure 22 is the average steady state once ASAT has dropped to three threads. The overall loop time is unchanged from the unloaded system. The first thread performance is identical to the unloaded system.

The second "thread" profile is somewhat different under ASAT with load than under an empty system. This anomaly is due to simple iteration scheduling with four iterations and three threads. The loop actually is coded as I=1,4 using simple iteration scheduling. When there are four threads, each iteration is processed by a thread. However, when there are only three threads, one thread must perform two iterations. In simple scheduling this is done statically, and, in this case, thread one gets both iterations one and two. So LA1 and LA2 are actually processed by thread one and LA3 is processed by thread two. This can be easily seen as the LA3 results are exactly the same as the EA2 results.

To understand the LA1 and LA2 results we will walk through thread one to see roughly how it executes. At 0 microseconds it checks to see if it is time for a barrier test as part of the ASAT evaluation. At 15 microseconds, it notifies the other threads that it has encountered a parallel section. At 22 microseconds it makes its I=1 mid-loop timer call (LA1) which takes 15 microseconds. At 37 microseconds it prepares to processes iteration I=2. It takes 6 microseconds to get back to the top of the loop. At 43 microseconds it makes the I=2 timer call (15 microseconds). However, because the overhead of the time call is subtracted after the loop completes, this timer call "appears" to be instantaneous so "time" stays at 43 microseconds. At 43 (adjusted) microseconds it arrives at the barrier (now in LA2) and since the rest of the threads (LA3, LA4) have been there since 35 (adjusted) microseconds, thread one (LA1 and LA2) passes through the barrier in about 6 microseconds for a total iteration time of 49 (adjusted) microseconds. In summary, because the first thread gets started so quickly, it is able to slip the I=2 iteration into the time it would have otherwise wasted waiting for the other threads due to their longer arrival skew.

This problem of unbalanced distribution of iterations to threads in simple scheduling becomes less of a problem as the number of iterations increases and it not unique to ASAT. It occurs anytime the number of iterations is not an even multiple of the number of threads. For an I=1,1000 loop executing with three threads, the threads will process 334, 333, and 333 threads respectively.

While the above figure describes the steady state average ASAT performance, the following figure shows how ASAT reacts to load over time. The following figure has three experiments, all performed with load on the system. The application is scheduled using (1) gang scheduling, (2) free scheduling, and (3) ASAT scheduling. The vertical axis is a log scale and is the wall time for each iteration. The horizontal axis is the iteration number. Each iteration from one to 200 was timed separately.

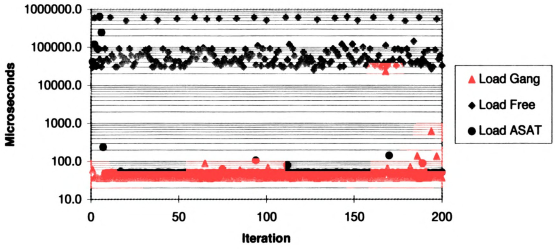

Figure 23 - ASAT Performance Over Time

In Figure 23, when the free scheduled application is executing the loop takes an average of 0.15 seconds per iteration. At the other end of the spectrum, the gang scheduled application averages 36 microseconds per iteration. For ASAT, it initially starts with four free scheduled threads and has six very poor iterations and then automatically drops a thread to execute with three free scheduled threads. At that point and for the remainder of the experiment ASAT executes the loop in 49 microseconds.

In summary, ASAT has an overhead of about 15 microseconds per parallel loop. In the shortest possible parallel loop[6] this overhead is 30%. In a slightly longer loop such as the trivial 1000 iteration loop which takes 6000 microseconds, this overhead drops to 0.3%. In a parallel loop which lasts 2 seconds, such as the *particle* benchmark in Figure 13, the overhead of ASAT drops to 0.001%. Since compilers will already suppress automatic parallization for loops which are too small, in general it is safe to say that if there is benefit to parallizing a loop, ASAT will benefit the loop under load and add less than 1% overhead for that loop.

4.2 Single Job Performance with ASAT

The goal for this section is to compare the simple application executed with ASAT on the SGI and Convex Exemplar with the execution on the Convex C-240 using ASAP.

4.2.1 ASAT Version 1 on the SGI

The first test is to duplicate the experiment which was performed for using ASAT to adjust the number of threads in the application code. Simple scheduling was used along with ASAT. This experiment uses ASAT Version 1 and is from [26].

[6] This is not the shortest practical parallel loop.

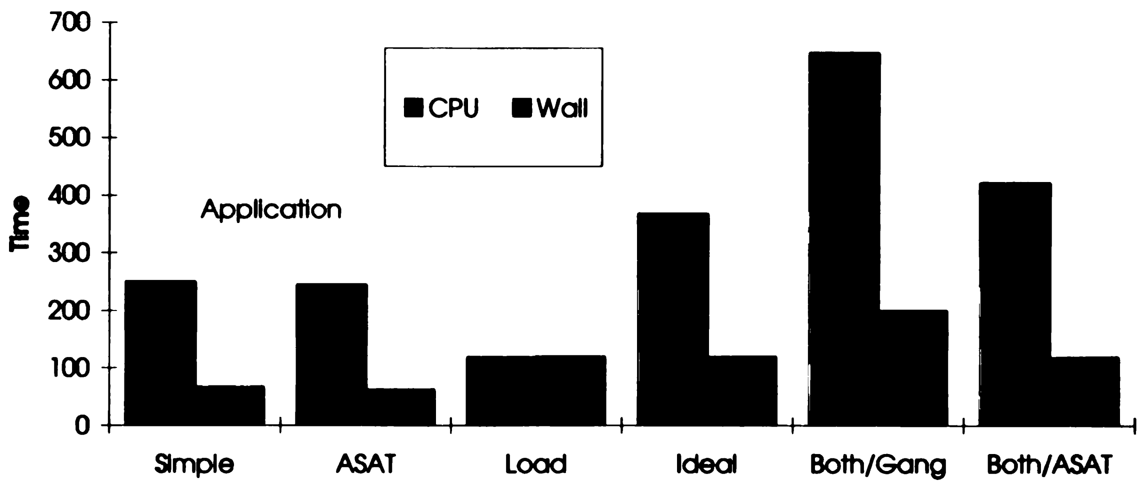

Figure 24 - ASAT Performance on the SGI

There are several observations about Figure 24. Running the application with ASAT enabled on an empty system did not change the performance of the program significantly (1-2 percent). The performance of the system with both the application and load running simultaneously is very close to ideal. Wall time for Both/ASAT was the same as ideal because the ASAT application ran to completion using the spare cycles before the load completed. The ASAT job runs at a lower priority than the load job so the load job effectively received 100 percent of the CPU for the duration of its run. The CPU time for Both/ASAT was 1.14 times the ideal CPU time. Recall that both the CPU and wall time were 1.05 times ideal for the ASAP on the Convex in Figure 9. Also the wall time for gang scheduling is 1.68 times longer than ideal and the CPU time for gang scheduling is 1.76 times longer than the ideal CPU time.

These results were made possible because the loop length was long enough that the 6000 microsecond ASAT Version 1 overhead was balanced by a longer application loop. Also some tuning on the part of the programmer was required to select an appropriate loop for the ASAT evaluation because it was executed before *every* selected parallel loop since no timer was used.

4.2.2 ASAT On the Convex Exemplar

In this section a similar experiment is run on a four-CPU Convex Exemplar. This experiment is the first test of ASAT Version 2 with the timed barrier and is from [29]. The "LOAD" job in this experiment is again a randomly arriving set of jobs with random duration, but the specific pattern and total usage are different from the earlier SGI and Convex C-240 experiments.

Experiment Details (the numbers in parenthesis indicate the seven bar graph pairs of Figure 25):

- A parallel application run on an empty system (1)
- The application with ASAT run on an empty system (2)
- A "load" job with several non-parallel jobs (simulating random activity of varying lengths) (3)
- A combination run with the non-ASAT application and the load jobs run together (4,5)
- A combination run with the ASAT application and the load jobs run together (6,7)

All runs were on a 4-CPU Exemplar with gang scheduling turned off and the ASAT job executed at lower priority.

The results from this experiment are shown below:

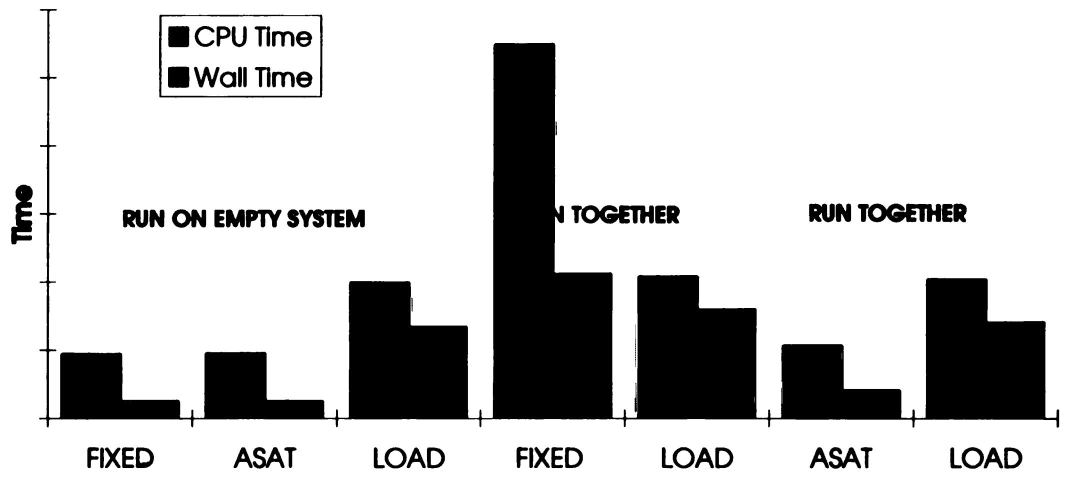

Figure 25 - ASAT on the Convex Exemplar

In Figure 25, the overhead of ASAT is measured, and the positive impact of ASAT on the overall performance is shown on the Convex Exemplar. This graph shows the results of three runs on an empty system and two runs together. The leftmost pair of bars represent the performance of the individual jobs run on an empty system. First a parallel application which did not use ASAT was executed (FIXED). Then the same application was run with ASAT added (ASAT). Both show a 4X speedup of wall time over CPU time. The ASAT overhead was effectively unmeasurable between the ASAT and FIXED job. Again, the LOAD job consists of a set of single-threaded jobs which arrive randomly with random duration. As before, if a parallel job is efficient, enough idle cycles exist during the elapsed wall time of the LOAD job, that the parallel can run to completion on the idle cycles only.

The fourth and fifth pairs of bars represent a run with the FIXED and LOAD jobs run together. The LOAD job did not experience much of a negative performance impact. However, the wall time of the FIXED job increased by a factor of 10. Effectively on the

Convex Exemplar, the FIXED application makes almost no progress at all during thread imbalance periods. These experiments were run on the SPP-1000 version of the Convex Exemplar. The SPP-1000 memory subsystem was very sensitive to increases in cache misses. Subsequent versions of this architecture (SPP-1200, SPP-1600, and SPP-2000) have each improved the memory subsystem performance and this extreme performance may not appear on those systems. I expect that on the SPP-1600 the effect will be roughly the same as on the SGI.

The last two pairs of bars represent the ASAT and LOAD jobs executing together. The LOAD job executed as if it were on an empty system. The ASAT job experienced almost no increase in CPU time, but did experience an increase in wall time as it only used the excess cycles.

In summary, the overhead for Version 2 of ASAT is effectively unmeasurable for even moderately short loops. This compares VERY favorably with the 5% overhead for the hardware based ASAP solution.

4.3 ASAT Response to Dynamic Load

To test ASAT under more varied load patterns, two time-oriented tests were performed. Again, these simple tests were performed with the Version 1 ASAT on the SGI. The first time-oriented test measured the ASAT response to rapidly changing load patterns. In the rapidly changing load scenario, the varying load conditions consisted of:

- One job that averaged 5 minutes CPU time and arrived approximately every 15 minutes
- Three jobs that averaged 1 minute of CPU time and arrived approximately every 4 minutes

These load jobs were all sequential and had higher priority than the ASAT application.

In Figure 26, the combination job finishes 4 minutes (11 percent) earlier when using ASAT scheduling. In addition, because ASAT processes run at lower priority, the time that the random load (simulating other users) completed was only 1 minute (4 percent) later than when the load completed on an empty system. Using gang scheduling, the simulated random load completed 7 minutes (20 percent) later than it would have completed with no competition for resources. In essence, the ASAT process "soaked-up" the idle cycles of the system with little or no impact on the rest of the load on the system. Because the ASAT process maintained a balanced number of threads it executed more efficiently and terminated faster than the gang scheduled process which also had a significant negative impact on the other jobs.

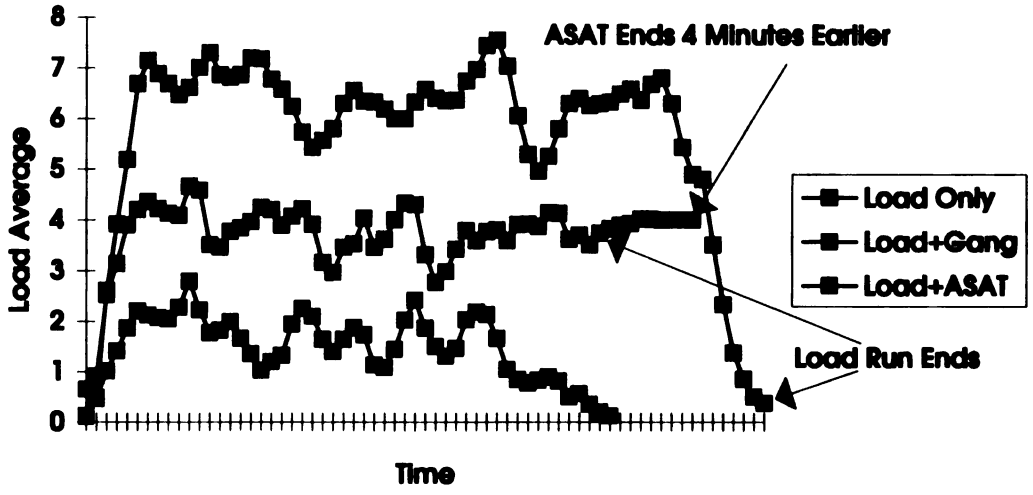

Figure 26 - ASAT Response to Rapidly Changing Load

The second time-oriented test is exactly the same as the previous test except that the load is more regular. At 2.5 minute intervals, the load is increased from 1 to 4 and then back down to zero. The same ASAT process was run with this new load profile. In Figure 27 the ASAT performance for this more slowly changing load in shown. In addition, the number of ASAT threads is shown. As the load is increased over the time of the run, ASAT quickly adjusts the number of its threads, maintaining system balance. As resources free up, the number of threads is increased to take advantage of the idle resources. The dynamic adjustment of threads results in complete and efficient utilization of the resources while providing priority to the short term load on the system.

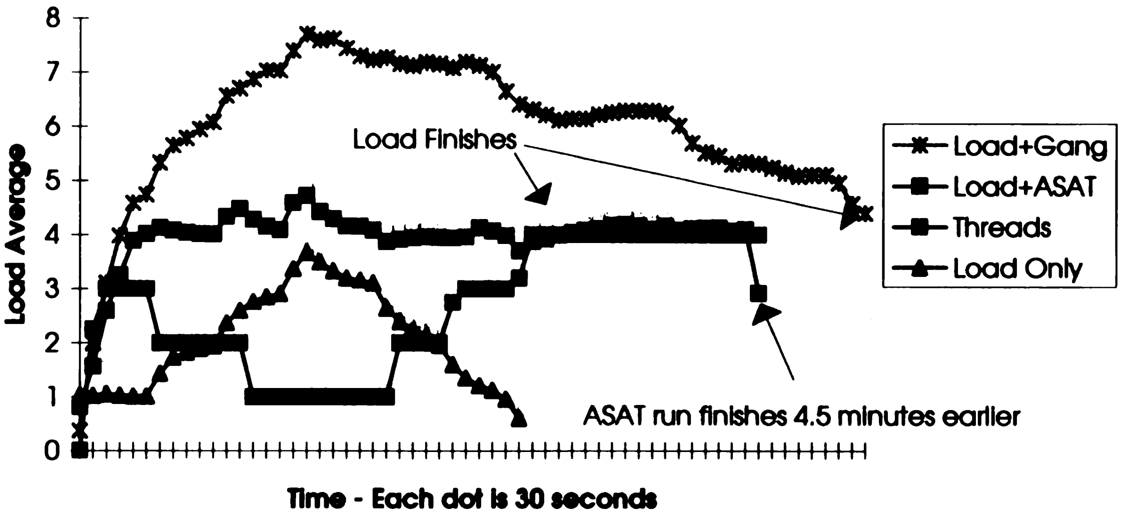

Figure 27 - ASAT Responses to Slow Changes in Load

Note that in Figure 27 the load job does not finish much later when run with the ASAT job than when it is run on the empty system. Further the overall number of threads when the Load+ASAT are running is maintained at four threads. As would be expected, the gang scheduled job severely impacts the serial job and negatively impacts its own performance as both jobs finish much later.

In this section, ASAT is shown to have performance benefits for an application which uses ASAT and the other applications which use the system. This win-win situation effectively gives all the users additional compute resources which would have otherwise been wasted without ASAT.

4.4 Focused ASAT Performance Tests

In the remaining section a series of focused experiments are performed which demonstrate the effectiveness of ASAT across a wide range of loop sizes and run-time settings. We have already established that the overhead of ASAT is so small that there is no harm in adding it to a loop worth parallizing. We now establish the situations in which ASAT has the greatest benefit. We also examine how the "load" or single-threaded jobs are effected by the scheduling choices used by the parallel jobs.

4.4.1 Run Time Scheduling Options

In this and the following section, a highly parallel application is used for all the experiments. This application is compiled and executed under a wide range of run-time scheduling options. The possible scheduling choices which can be made with this application are as follows:

- The entire computation can be executed in parallel or serial
- Gang Scheduling can be turned on or off.
- ASAT thread adjustment can be turned on or off
- Loop termination strategy can either be a hard-spin or with the ***release_cpu*** behavior (as described in Section 2.3.1) enabled

The following table summarizes the option settings for the various runs:

Title	Number of	Gang	Thread	Loop
Single	1	N/A	N/A	N/A
ASAT	4	No	ASAT	Hard_spin
ASAT-	4	No	ASAT	Release_cpu
Gang	4	Yes	Fixed	Hard_spin
Gang-R	4	Yes	Fixed	Release_cpu
Free	4	No	Fixed	Hard_spin
Free-R	4	No	Fixed	Release_cpu

Table 1 - Types of Run-Time Choices

The "-R" in the above indicates "*release_cpu*". The idea is that with hard-spin turned off, the performance of the parallel application might suffer, but less CPU time would be wasted by the loop-end spinning resulting in better overall system utilization. These titles will be used throughout the remainder of this section and the following section to label the graphs.

4.4.2 Code Structure

The basic structure of the code is a parallel inner loop with a serial outer loop. The code uses the ANSI X3H3 parallel FORTRAN directive format as supported by the Kuck and Associates Guide compiler:

```
      DO I = 1,EXCOUNT

C Perform ASAT adjustment if appropriate

C$PAR   PARALLEL
C$PAR& SHARED(A,B,C) LOCAL (J)
C$PAR PDO
        DO J=1,GRAINSIZE
          A(J) = B(J) + C(J)
        ENDDO
C$PAR END PARALLEL
      ENDDO
```

In order to test the effect on programs with different memory access patterns and loop duration times, the inner loop length (*GRAINSIZE*) is varied. This inner loop length is called the "grain size" as it affects the granularity of the parallel sections. The number of iterations of the inner parallel loop can be adjusted from 1K to 4M. The size of the data structure used in the loop is also adjusted. Varying the data structure size will affect how much of the data accessed by the application will actually reside in the cache of the system. In order to processes the same "work", the number of outer loop executions (*EXCOUNT*) is decreased as the inner loop iteration length (grain size) is increased. The following table relates the grain size to execution count, the wall time used per iteration on a single unloaded CPU, and the size of the data structure accessed by the system.

Grain	Execution	Iteration Time	Data Accessed
2K	200,000	0.00035s	48K
10K	40,000	0.0022s	240K
100K	4000	0.035s	2.4M
1M	400	0.35s	24M
4M	100	1.4 s	96M

Table 2 - Parameters Relative to Grain size

4.4.3 Execution Environment

The compiler used for these tests is a Beta version of the Kuck and Associates Guide compiler with the Flow(ASAT) run-time extensions. The version of the compiler used for these results is: "Guide 2.00 k270721 960606". The system used for these tests is an SGI Challenge with the following attributes:

```
4 150 MHZ IP19 Processors
CPU: MIPS R4400 Processor Chip Revision: 5.0
FPU: MIPS R4000 Floating Point Coprocessor Revision: 0.0
Secondary unified instruction/data cache size: 1 Mbyte
Data cache size: 16 Kbytes
Instruction cache size: 16 Kbytes
Main memory size: 384 Mbytes, 2-way interleaved
Operating system: IRIX 6.2
```

For the ASAT jobs, the following settings were used:

ASAT Parameter	Value
ASAT_EVAL_TIME	1.00
ASAT_ BAD_TRIG (KMP_ASAT_DEC)	1
ASAT_ GOOD_TRIG (KMP_ASAT_INC)	10

Table 3 - ASAT Run-Time Settings

See the previous chapter for the definitions of these variables.

4.5 Running Jobs on an Empty System

The following figures show the performance of the different jobs on an empty system for various grain sizes. The green lines are the ***release_cpu*** lines and the blue lines are the hard-spin lines.

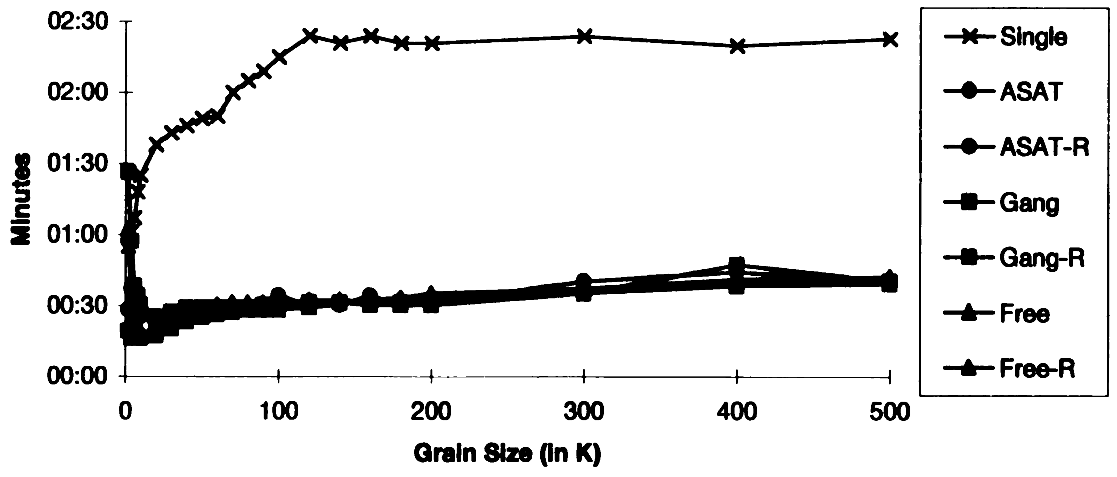

Figure 28 - Runs on Empty System

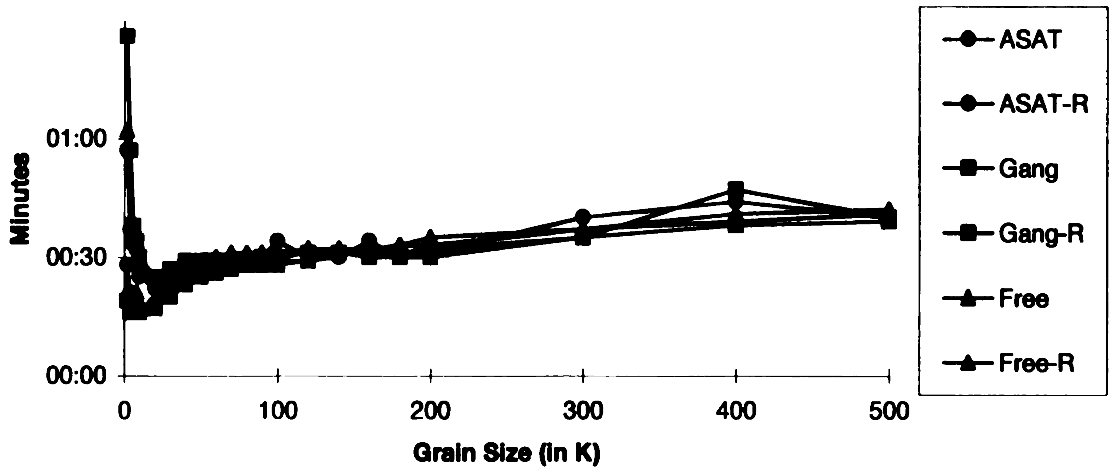

Figure 29 - Runs on Empty System (Expanded Vertical Axis)

As expected, in Figure 28 and Figure 29 the parallel jobs on an empty system have essentially the same running time regardless of basic scheduling choice (ASAT, Fixed, or Gang) as long as hard-spin is enabled. When the *release_cpu* behavior ("-R" runs) is enabled performance suffers across all loop sizes with a more significant impact with grain sizes of 50K or less. Interestingly for small grain sizes (< 10K) Gang Scheduling has slightly better performance among the six experiments shown when hard-spins are used (Gang) and the worst performance of the six experiments when ***release_cpu*** (Gang-R) is used. In general, the parallel jobs execution time is considerably faster than the single-threaded execution.

One can see the effect of the first and second levels of cache as jumps in the graph of the single threaded run. While even the smallest loop at 2K (48K working set size) will not completely fit into the 16K L1 cache, it fits in the L2 cache and the L1 cache can hold much of the data. Between 50K and 100K in the single threaded run, the data structure can fit in the 1MB L2 cache. Above 200 K, none of the data structure fits in any of the caches from iteration to iteration and the application executes at main-memory speeds. To see the speedup of the parallel application over the serial application more clearly and factor out some of the cache effect, in the following figure the vertical axis indicates the performance as a ratio relative to the single threaded application execution time on an empty system.

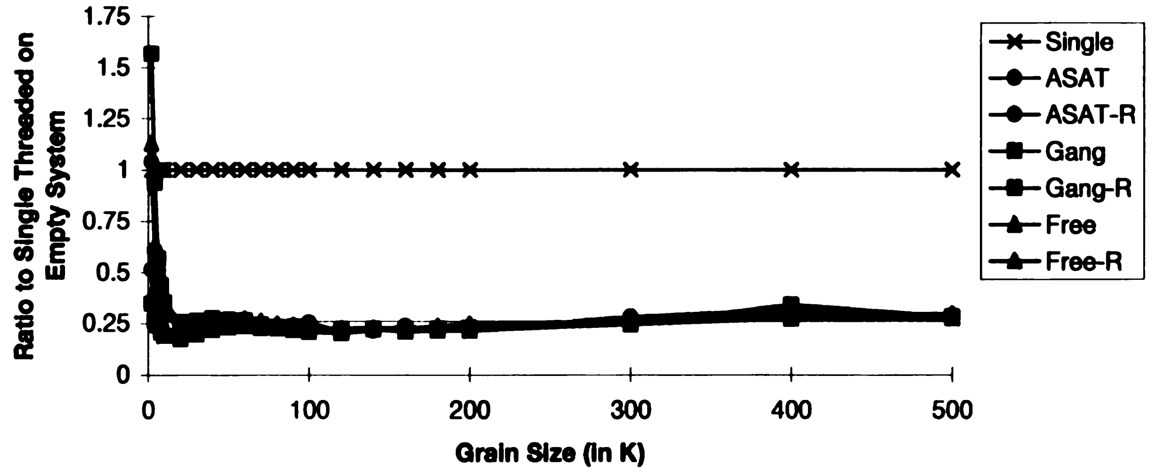

Figure 30 - Speedup for Parallel Jobs on Empty System

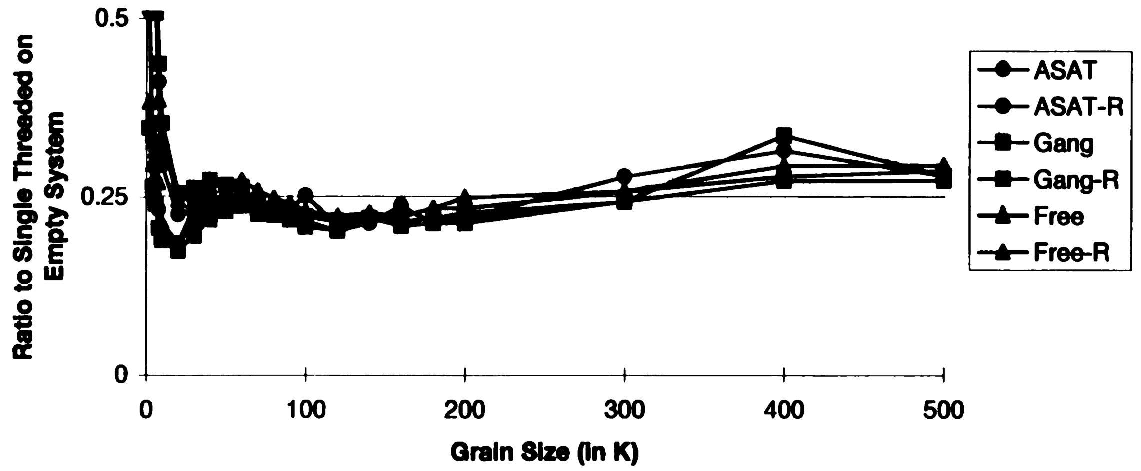

Figure 31 - Speedup for Parallel Jobs on Empty System (Expanded Vertical Axis)

In Figure 30 and Figure 31, the benefits and effects of parallelism on this application are shown. The first observation is that the performance of ASAT tracks the performance of gang scheduling very closely. Gang Scheduling only has a benefit over ASAT for very small loops (<=4K) . On these small loop sizes (<=4K), no scheduling setting achieves linear speedup due primarily to the overhead of the parallel loop. A line representing the performance for linear speedup is drawn at 0.25. Between iteration sizes of 8K and 300K, the application experiences super-linear speedup. The first dip in the graph represents the parallel application taking advantage of an effective 64K of L1 cache. The second dip represents the advantage of the effective 4M L2 cache due to the four processors cooperating. At 300K the speedup is linear and above 300K the speedup is close to linear as both the serial and parallel applications are executing out of main memory.

While all these cache effects are interesting, the strong result from these two figures is that ASAT is not a significant negative performance impact across a wide range of iteration sizes. Further, gang scheduling is not required to achieve excellent performance on an empty system. Also, the *release_cpu* option has a slight negative impact on the performance of parallel applications. In the next section, we examine the effect of adding a single threaded load application to these parallel jobs.

4.6 Running Combinations of Serial and Parallel Jobs

In this series of experiments, the parallel applications are run simultaneously with a serial application. The memory reference patterns and iteration sizes are identical for each trial. When the parallel application has a grain size of 2K, the corresponding serial application also has a grain size of 2K. For each experiment the pair of jobs were run twice. First the serial job was run to completion and timed while the parallel job executed in the background continuously. Then the parallel job was run to completion and timed while the serial job executed in the background continuously. In this way, we see the steady-state impact of each job on the other. In the following figure, the performance of each of the ASAT and Gang combination runs is shown. The line color is changed to red to indicate that the hard-spin jobs were run under loaded conditions. The purple lines are the loaded conditions with the *release_cpu* behavior enabled. In these graphs, a label such as "Single/ASAT" is used for the performance of the single threaded job when the ASAT job was running in the background. A label such as "Gang/Single" is used when the performance of the gang scheduled job is measured while the single threaded job runs in the background. The solid symbols are the parallel jobs executing with the single threaded job and the outline symbols are the single threaded jobs executing with parallel jobs. The symbol shape indicates which type of scheduling was used in the parallel job (Circle - ASAT, Square - Gang, and Triangle - Free).

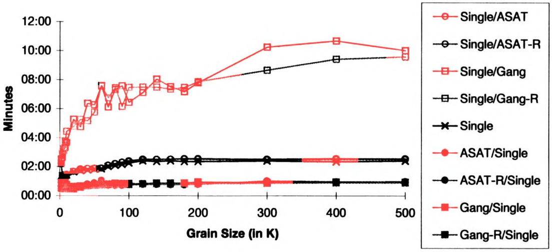

Figure 32 - Performance of ASAT and Gang Combination Runs

In Figure 32, the most dramatic result is the significant slowdown shown by the single threaded application when executed simultaneously with the Gang Scheduled application with either loop termination option. In comparison, the slowdown experienced by the single threaded application when executing with the ASAT job is nearly imperceptible. This result is similar to the earlier single job results which shows that ASAT can use up "excess" cycles from a serial job without harming the performance of the serial job. The performance of the Single/ASAT and Single/ASAT-R are so close that they appear as a single line slightly above the Single (on an empty system) line in the center of the graph.

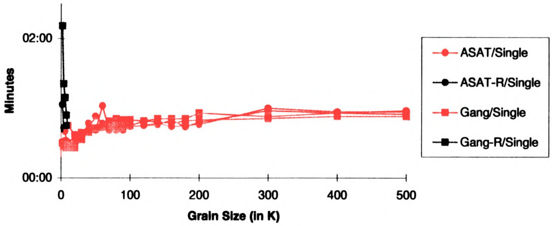

Figure 33 - Performance of ASAT and Gang Combination Runs (Expanded Vertical Axis)

In Figure 33, the addition of the *release_cpu* behavior in the parallel job does *not* significantly improve the performance of the serial application under load for ASAT or Gang Scheduling. Releasing the CPU (purple lines) can have a negative performance impact on the parallel application of up to 400% at small grain sizes (<10K) and little or no negative effect at the larger grain sizes (>100K). Releasing the CPU in a Gang Scheduled application (Gang-R) seems to confuse the operating system somewhat in that the application has requested that all of its threads execute simultaneously and now it is releasing one of its threads. There seems to be little benefit to the serial application when the gang scheduled application releases the CPU at the end of the loops.

The following figure shows the relative performance of these jobs compared with the single threaded application. As before a relative performance plot helps eliminate the effect of cache.

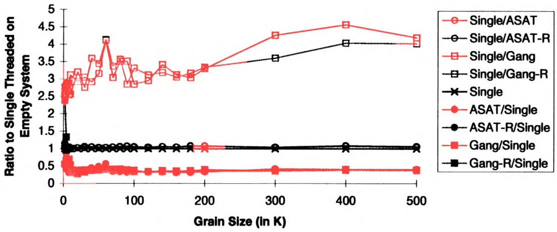

Figure 34 - Relative Performance of ASAT and Gang Compared to Single Threaded Job on an Empty System

In Figure 34, the slowdown when the single threaded job executes with the gang scheduled job is roughly 3.5 times for grain sizes that fit in L2 cache. Once the data structure no longer fits in the L2 cache (>200K), the performance impact is a factor of 4.5. To understand the smaller details on the parallel job performance curves, this figure is shown again below with an expanded vertical axis scale.

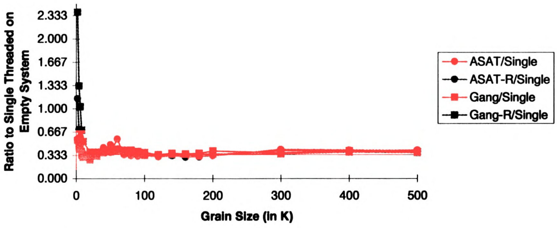

Figure 35 - Relative Performance (ASAT and Gang) compared to a Single threaded Job on an Empty System (Scale)

In Figure 35, a line is drawn at the "ideal" speed -up ratio of 0.33 (assuming one processor is dedicated to the single threaded CPU on the average). Also, the super-linear speedup only occurs when the parallel application can best take advantage of its aggregate 48K L1 cache. However, the effect of the L2 cache can still be seen on the graph.

The following figure shows the performance of the applications which use neither Gang Scheduling nor ASAT scheduling. We term this "Free[7] scheduling" because the operating system is free to schedule any thread without regard to its relationship to the other threads. When parallel jobs are run with serial jobs in this manner, the threads are all executed in a round robin fashion. In a sense, the parallel job is not "demanding" special scheduling as in Gang Scheduling. Furthermore, the number of threads is fixed at four throughout the entire duration of the application execution regardless of the load on the system. The use of Free scheduling is one possible way to allow a parallel job to "soak-up" excess cycles without negatively impacting the serial job.

The following figure shows the performance of the Free jobs on an empty system and when run with serial code. Both the hard-spin and *release_cpu* graphs are shown.

[7] ASAT uses Free scheduling for its threads, but actively adjusts the number of threads in use to avoid the problems seen in the Free/Fixed experiments.

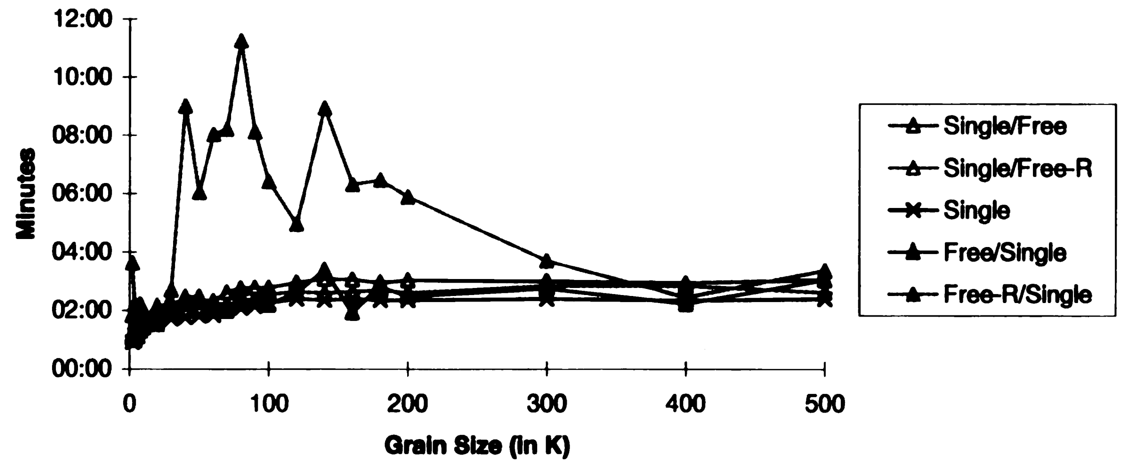

Figure 36 - Performance of Free Scheduled Jobs With Fixed Threads

In Figure 36, the serial jobs experience only a slight negative performance impact when executed with either Free scheduled job. When the Free scheduled parallel job releases the CPU, the serial job gets slightly better performance, but the performance of the parallel job is very poor. The best parallel performance occurs when hard-spins are used, but even when using all four CPUs, the Free scheduled, hard-spin job runs *more slowly* on four processors than on one processor. If the Free scheduled parallel job releases the CPU, its performance suffers dramatically at some grain sizes. Free scheduling appears to be a lose-lose situation, when jobs are run in combination, both the serial and parallel jobs run slower than optimal.

In the following figure, the relative performance of all the single threaded jobs under the different load scenarios is shown. That is, the various single threaded plots from the previous figures are combined into one graph.

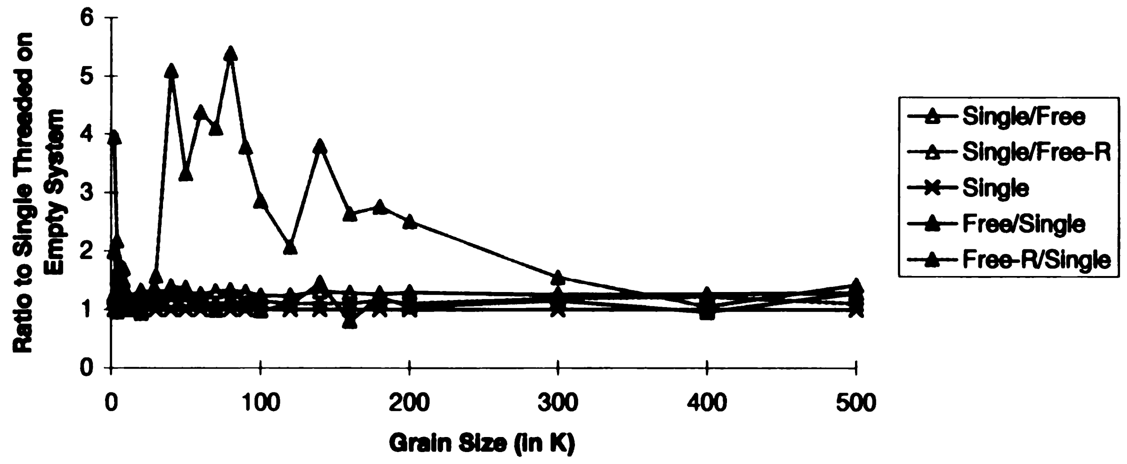

Figure 37 - Performance of the Single Threaded Job With Other Jobs Executing

In Figure 37, the best performance occurs when the job is run on an empty system followed closely by the performance when run with ASAT. The best non-ASAT choice with respect to single threaded performance is Free scheduling with ***release_cpu*** enabled followed by Free scheduling with hard-spin. Gang, and Gang-R scheduling have a large negative performance impact on the serial application. Neither Gang Scheduling nor Free scheduling are sufficient to efficiently "soak-up" excess cycles like the ASAT scheduling approach. ASAT has the strengths of both Gang and Free with none of the weaknesses.

In the following figure, the performance of the parallel jobs is shown with only the hard-spin versions included. The empty single threaded performance and empty system parallel gang scheduled performance is included for reference.

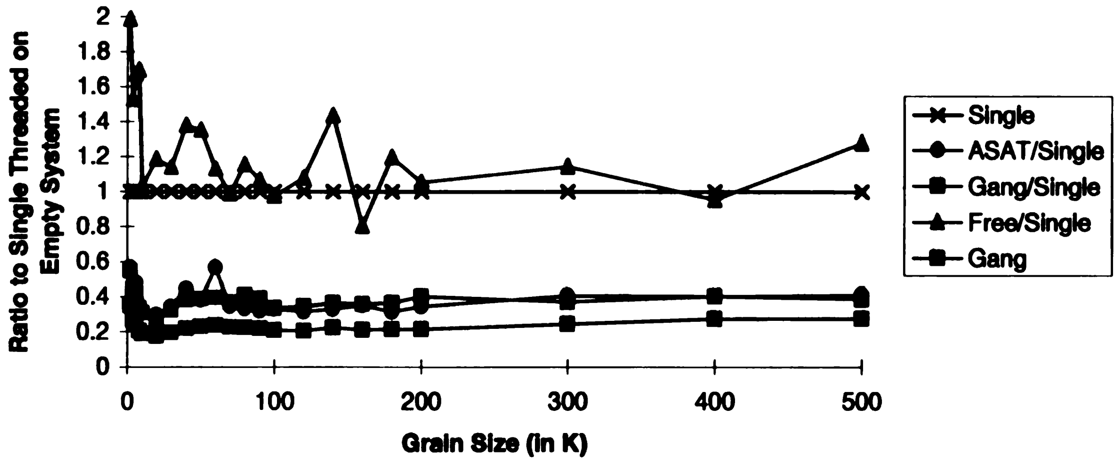

Figure 38 - Performance of Various Parallel Jobs with a Single Threaded Job Running

In Figure 38, there is no particular advantage of Gang Scheduling over ASAT when load is present on the system. Fixed Scheduling performs very badly in the presence of load.

The experiments described in this section were run in the intended environment for ASAT, and demonstrate its ability to "soak-up" free cycles without impact on the "foreground" work. ASAT uses its lack of gang scheduling and lower relative priority to ensure that it always has the precise number of threads. In the next section, we will explore what happens when multiple ASAT jobs are running at the same time with no other load on the system. ASAT will no longer have the advantage of its relatively lower priority to detect system load. In a sense, an ASAT process assumes that any excess threads are its responsibility. When multiple ASAT jobs are run, they all attempt to be polite and drop threads. Further because all of the other processes are polite ASAT jobs may decide to periodically add a thread, causing imbalance. The purpose of this section is to determine how significant a problem this is for ASAT.

4.7 Running Multiple Parallel Jobs

This section poses a set of experiments which test the stability of ASAT with multiple parallel jobs on the system. The primary challenge for ASAT in this situation is that when multiple jobs run on the system with no other load, all of their priorities are equal. As such the ASAT evaluation cannot take advantage of relative priority as in the ASAT/Single scenario. It should be noted that this is not the anticipated design target for ASAT. The assumption is that if there were long-running parallel batch jobs, each capable of consuming the entire system efficiently, they would be best executed one at a time from a queue soaking up the all the available cycles from the non-batch system load.

The experiment details are the same as in the previous sections. Five representative grain sizes will be tested (2K, 10K, 100K, 1M, and 4M). Three of these correspond to sizes used in the previous section and the 1M and 4M are larger than the previous experiments. In all cases one to four identical jobs are executed. They all use the same grain size and run-time scheduling option settings.

4.7.1 Multiple Grain Size=2K Jobs

The first experiments examine the performance for small threads using grain size = 10K. As described earlier, the effect of running multiple copies of different types of jobs is studied. In each case, one to four copies of each type of job are run on an otherwise empty system. For each run, the overall wall time is measured for the completion of all jobs. The gang scheduled jobs each use four threads so the total number of threads across the entire system increases from four to sixteen as the number of copies goes from one to four. In the following figure, red lines are used for the hard spin behavior and purple lines indicate that ***release_cpu*** is used..

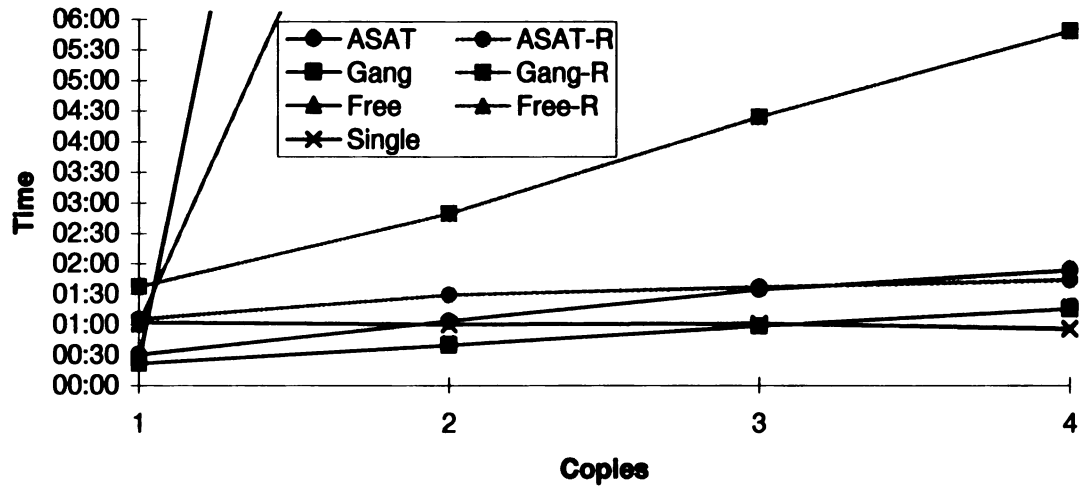

Figure 39 - Multiple Jobs with Grain Size 2K

In Figure 39, a number of effects can be observed. When looking at the hard-spin jobs, Gang starts out with a small advantage over ASAT which increases as the number of copies are increased. This advantage is roughly 20%. At four copies, single threaded is the fastest. When looking at the ***release_cpu*** graphs, the ASAT-R job performs poorly when there is only a single copy on an otherwise empty system. The problem is that with such a small grain size, any thread skew at the end-loop barrier will result in the release of the CPU. Then the released thread must be reactivated at the top of the loop which causes relatively large thread skew at the top of the loop. The net result is that the ASAT run-time effectively perceives its own thrashing as load on the system, and drops to one thread resulting in performance that is about the same on the empty system as the single threaded job.

Interestingly, as the number of copies of the ASAT-R job are increased, its relative performance improves to the point that at four copies it out-performs the ASAT without *release_cpu*. While this may seem curious, there is a simple explanation. The ASAT-R job is beyond "polite", it has become "paranoid". That is, all of its metrics are indicating very heavy load, and it quickly drops to one thread and stays at one thread. With the normal ASAT, at four copies each job "tries" to see if there are available cycles and at times, the two-threaded barrier runs successfully and ASAT adds a thread. Usually, a short time (1-2 seconds) later ASAT notices its mistake and drops back to a single thread. The result is that when four copies of the normal ASAT job are running (with the same low priority), the load metric used to decide to increase threads provides incorrect information. The ideal performance of ASAT with four copies should be the same as the Single-threaded application with a very slight overhead added. As such, the cost of this ASAT behavior can be easily observed from this that the following graphs. In this situation because the grain size is so small, the cost is roughly 100%.

The Gang Scheduled parallel job with *release_cpu* enabled (Gang-R) performs poorly because it does the same thrashing that ASAT with *release_cpu* experiences except that it never drops to one thread. The net result is poor performance which gets worse as more copies are added.

The Free scheduled jobs perform the same as the ASAT jobs on the empty system, but as soon as two copies are executed, the performance becomes very poor. Four copies without *release_cpu* take over seven hours to complete and with *release_cpu*, takes nearly three hours to complete.

4.7.2 Multiple Grain Size=10K Jobs

In the 10K grain size experiments, cache effect is still significant so a scheduling technique will be rewarded for effective use of the cache. Further, the L2 caches are sufficient to hold all four applications' data simultaneously so most memory references will be to L2 cache. Since the loop iteration time is on the order of 2000 microseconds, several loop executions may be processed during a single time slice. The graph of the results of running one to four copies with grain size of 10K is as follows:

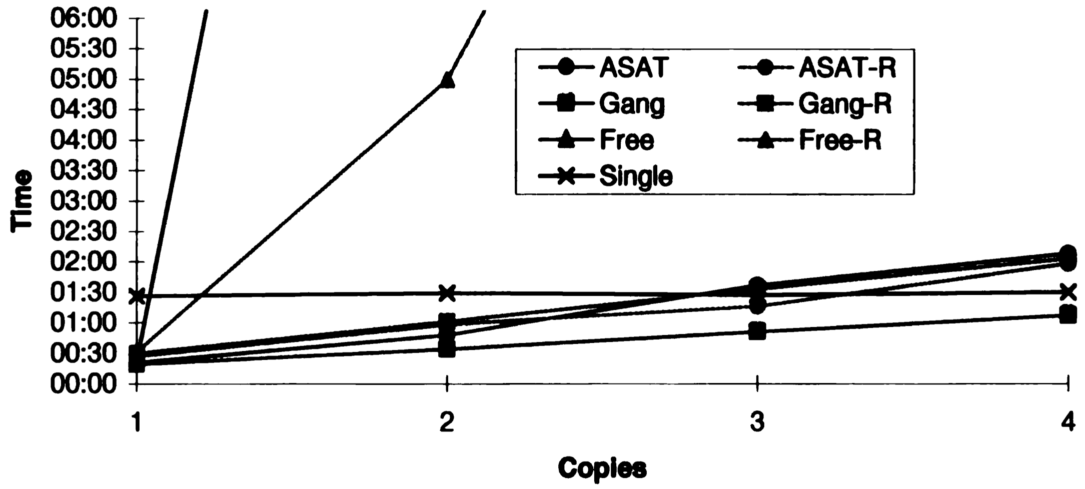

Figure 40 - Multiple Jobs with Grain Size 10K

In Figure 40, the choice of scheduling technique has less of an impact. ASAT and ASAT with *release_cpu* enabled have nearly identical performance except that ASAT has a slight advantage below two copies and ASAT-R has a slight advantage for three or four copies. Again this is because ASAT-R is more conservative about adding threads and tends to run less often in sub-optimal configurations with four copies. The *release_cpu* option has a more significant negative impact for gang scheduling. This is due to two reasons: (1) the application is ideally load balanced so under gang scheduling the loop-end thread skew is very small and releasing the CPU is foolish, and (2) the loop executes quickly enough that it can be executed several times during a time slice and keep a warm cache under gang scheduling. Gang Scheduling with hard-spins at loop termination is the fastest approach. It has super-linear speed-up even when there are four copies executing. This superlinearity is a combination of the aggregate 64K L1 caches and the ability to execute more than one iteration during an operating system time slice. The fact that the SGI IRIX operating system can time slice the CPU cleanly and evenly between four gang scheduled applications is a compliment to their scheduling implementation.

The ASAT jobs continue to be mislead about the overall load at their thread increment time resulting in a 42% cost when compared to four copies of the single threaded job.

Again the Free/Fixed jobs experience very bad performance when more than one copy is executed. At four copies, the Free jobs execute in over five hours and the Free-R jobs execute in over an hour.

4.7.3 Multiple Grain Size=100K Jobs

As the grain size is increased to 100K, the effect of the L1 caches is less noticeable. Also with a loop iteration time of roughly 0.035s, there is a reasonable chance that a parallel loop execution will experience an operating system induced context switch so that an excess context switch caused by the ***release_cpu*** option is less noticeable.

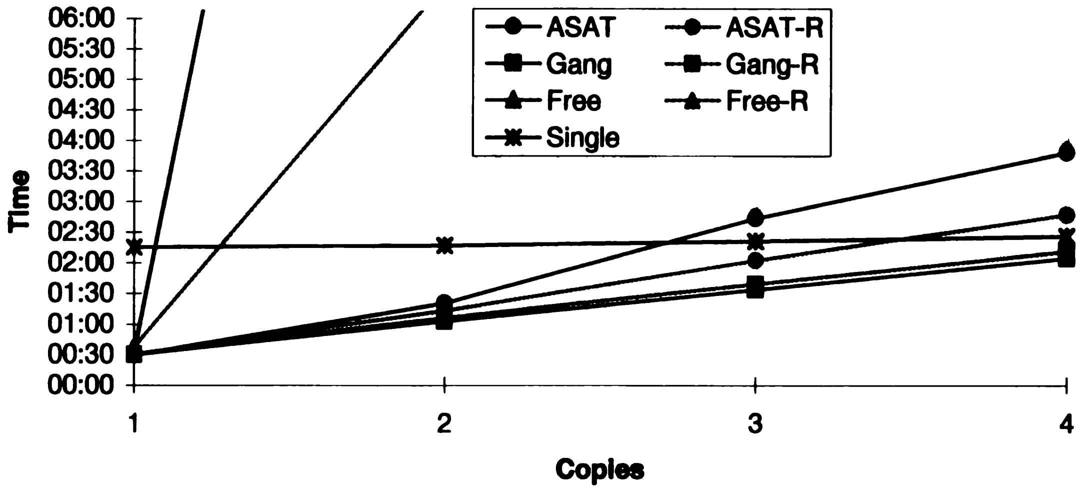

Figure 41 - Multiple Copies with Grain Size 100K

In Figure 41, the Gang and Gang-R have nearly the same performance because the overhead caused by releasing and requiring the CPU is small compared to the loop execution time. Both Gang Scheduled approaches still exhibit super-linear performance at four copies because of the aggregate cache effect. The negative performance impact of ASAT is also decreasing as the loop length increases. ASAT has an 45 percent overhead and the more conservative ASAT-R has dropped to 15 percent.

Again the Free and Free-R jobs perform poorly with multiple copies. At four copies they take 1.5 hours and 45 minutes respectively.

4.7.4 Multiple Grain Size=1M and 4M Jobs

At larger grain sizes, there is no remaining cache effect and all memory operations are done to main memory. These working set sizes eliminate any remaining super-linear speedup.

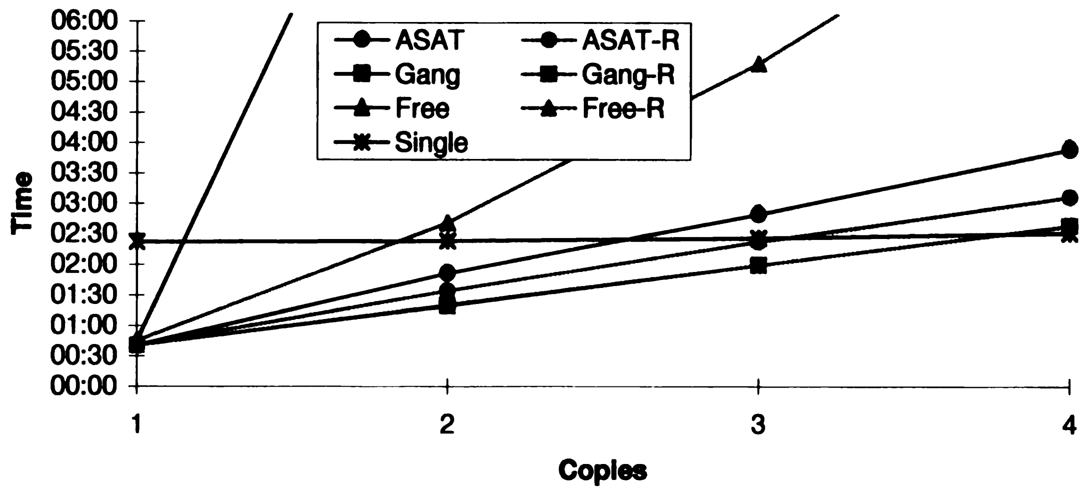

Figure 42 - Multiple Jobs with Grain Size 1M

At the grain size of 1M, in Figure 42 all of the curves are approaching the single threaded performance at four CPUs. There are a number of factors which cause this. (1) Data is seldom reused in cache, so there is little advantage to being scheduled on a processor with a "warm" cache. (2) When a context switch does occur there is plenty of effective work to do in either type of job so progress to the solution continues. The ASAT job continues to have a large (52%) performance loss when compared to the single threaded job because of its use of too many threads due to its inability to judge when to add a thread. Further, the lack of deterministic scheduling leads to wasting time spinning at the bottom of the loops in join barriers. The time spent at join barriers is not used toward productive work. With such a large grain size, releasing the CPU at a join barrier improves performance 20% for ASAT.

When examining the Free and Free-R performance, while the Free performance is very poor (4 copies take over 20 minutes) the Free-R jobs complete at about 9 minutes. This is down to a factor of 3X the performance of the other techniques at four copies. With a loop iteration time of 0.35s, the operating system is probably involved in every loop termination in Free-R but some progress does occur.

In the following figure, the grain size is set to 4M with a data set size of 96MB for each process. The SGI Challenge used in this experiment has a total of 384 MB of memory so three jobs can fit into memory while four jobs cannot fit. The performance for this set of experiments follows:

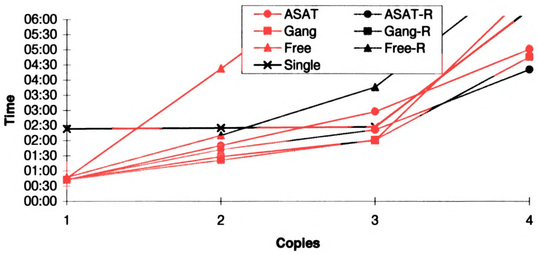

Figure 43 - Multiple Jobs with Grain Size 4M

In Figure 43, the performance trends are very similar for one and two copies. At three copies some paging begins, but by four copies, there is very little data resident in main memory for each iteration. Context switches are insignificant given the large number of page faults. Releasing the CPU and reactivating a thread also does not show up significantly. However, there are some interesting results when four copies are run. ASAT with *release_cpu* performs the best followed by ASAT, and Gang with *release_cpu*. Fixed-R is only 20 percent slower than Gang when four copies are run.

Fixed with hard spin is the worst performer at 17 minutes. Interestingly Gang and Single have nearly identical results when four copies are run.

The problem with exceeding real memory is that memory operations become I/O operations. In this situation, more threads are superior so multiple I/O operations can be performed in parallel. Single has the worst performance because at most four page faults can be outstanding at any one time. In Gang Scheduling, the operating system keeps the threads together so the additional threads are often spinning and not free to generate additional page faults. In any of the *release_cpu* enabled runs, threads do not spin and are released to generate more page faults on some other thread. ASAT with hard-spin loses some performance spinning when it could have released the CPU.

In summary, several conclusions can be drawn regarding the experiments in this section:

- Gang Scheduling on the SGI Challenge under IRIX 6.1 is excellent at time slicing between multiple parallel processes with the same number of threads in each process. The context switching is so effective that superlinear speedup often occurs as the effective cache size increases.

- ASAT does not do as well as Gang Scheduling with multiple parallel jobs on an otherwise empty system. The gang scheduling advantage is greater the more copies are executed.

The problem with ASAT is two-fold. First, it does not get the benefit of the larger aggregate caches which occur for gang scheduling. The more disturbing result is that four ASAT jobs do not perform as well as four single threaded jobs. This indicates that the

ASAT evaluation is thrashing in its thread determination. For example when four copies of ASAT jobs are running, they should each move toward one thread and use one thread for the remainder of the run. Any negative performance should only occur in the first few seconds of the run. While the ASAT thread reduction mechanism does quickly move the processes down to one thread each, they remain forever optimistic and continually try additional threads. Because all processes are running at the same low priority, ASAT often believes that load has changed and increases the number of threads, only to drop the thread later. During the time an excess thread exists, the system is operating poorly as in a Free scheduled situation. ASAT quickly drops back to one thread but the damage has been done.

There are two possible solutions to the problem. The simple solution is to set the ASAT_GOOD_TRIG to a relatively large number such as 60 seconds. In this way, even if multiple jobs are running, they will run poorly about 1/60 of the time. Given the relatively slow pace of load change and the fact that ASAT will still react quickly to reduce threads, this is a quite acceptable solution. Another solution may be to consult the operating system (as in SGI Version 1) every 30 to 60 seconds for a picture of the overall load. This would only be done at the thread increase decision point. Another solution is to simply run all the parallel jobs capable of consuming the entire system efficiently one at a time from a batch queue. This single ASAT job would be quite compatible with combinations of serial and gang-scheduled jobs which used less than the entire system.

4.8 Summary

In this section, the performance of ASAT was tested under a wide variety of load conditions. On an unloaded system, ASAT, Free Scheduling, and Gang Scheduling have nearly identical performance. When a single-theaded job is executed in combination with a parallel job, Gang Scheduling maintains good parallel performance at the expense of the single-threaded job. Fixed Scheduling maintains good performance for the single-threaded job at the expense of the parallel job. Only ASAT provides ideal performance for both the single-threaded and parallel job.

The performance is also studied when multiple (1-4) copies of the parallel application are run on an otherwise empty system with different scheduling options. In this scenario, Gang Scheduling outperforms ASAT by up to 50%. ASAT does significantly better than Fixed Scheduling in this situation. The causes of the performance disadvantage when comparing ASAT to Gang are explored and potential solutions are proposed.

CHAPTER 5. CONCLUSION AND FUTURE WORK

5.1 Conclusion

The ability to dynamically adjust a parallel application to the amount of available resources is an important tool which allows parallel processors to be used more efficiently and applications to complete more quickly. In this paper, the negative performance impact of having a system with an unbalanced number of threads was investigated and reported. ASAT is proposed as an efficient technique which is easily implementable in a run-time library which effectively balances thread use across an entire system without requiring any central information. The overhead of ASAT as implemented on the four CPU SGI Challenge is precisely measured to be 15 microseconds per ASAT evaluation.

The performance of ASAT is compared to the existing state of the art in hardware, compiler and operating system provided load balancing solutions. ASAT is shown to be superior to all other scheduling approaches when a single ASAT job is executing on the system with other non-parallel applications. Because ASAT is completely decentralized, it depends on a lower priority and free scheduling to get an accurate idea of the overall load on the system. As such, ASAT has some trouble when there are multiple equal-priority jobs running on the same system simultaneously. In this case, ASAT is far superior to Free scheduling, but Gang scheduling has a performance advantage over ASAT when there is a job mix of multiple competing parallel jobs.

If there are serial jobs running with the gang scheduled jobs, the serial job performance is negatively impacted. In a sense Gang scheduling is a win-lose situation, the parallel applications win and the serial applications lose. ASAT is a win-win situation in that a parallel job wins and the serial jobs have nearly empty-system performance.

The goal of ASAT was to emulate Convex's Automatic Self-Allocating Processors (ASAP) hardware solution to dynamic load balancing. In the case of a parallel job running to soak up excess cycles under a wide variety of load conditions, ASAT has met and exceeded that goal.

ASAT is a production-ready technique which has already been integrated into one commercial compiler. As more and more users deploy commodity processor based parallel processing systems, they will find that ASAT allows them the flexibility for these systems to be used by a wide range of users with a minimum of manual system management.

5.2 Future Work

Further study is needed to determine how to best implement ASAT using further compiler and operating system modifications. ASAT, as currently implemented, does not make or require any operating system changes. One operating system change which would be helpful to ASAT is to assign a lower priority to processes with more active threads. This modification would naturally encourage processes with the largest number of threads to give up their threads and balance overall usage in the long run. Such an approach would also penalize non-ASAT processes which make irresponsible use of system resources. Also, the operating system needs to provide a non-adjusting priority that does not have a negative impact on the performance of a parallel application.

Another area of work is to do a long-term study of the overall effect of ASAT. This work would allow one to study the average time spent in a parallel section across a wide variety of applications.

Another possible outcome of this work is to develop the thread-skew measurement program into a benchmark and collect and publish parallel performance data on a wide range of computer systems and run-time environments.

Another area of work in ASAT is to develop a version of ASAT which consults some external information before increasing the threads to solve the only remaining advantage of Gang Scheduling over ASAT. This is the only remaining "production" feature of ASAT.

APPENDIX A - SOURCE CODE FOR LOOP LATENCY TEST

This is the source test for the loop-skew testing program.

```
      IMPLICIT REAL(A-Z)
      INTEGER I,J,K
      LOGICAL CHECK_ASAT
      REAL*8 MSF77W
      REAL*8 COST,BARMAX,BARMIN
      REAL*8 LASTTIM
      PARAMETER(TSIZE=200)
      REAL*8 PRE(tsize),IN(tsize),XIN(tsize,4)
      REAL*8 OUT(tsize)
      REAL*8 ASAT(tsize),LOOP(tsize),SPAWN(tsize,4),BAREND(tsize,4)
      CHARACTER ENVFLAG*10
      PARAMETER(GOODTRIG=0.001)
*
* Determine the time overhead while you have a good timeslice
*

      PRINT *,'Determining clock Overhead ...',MSF77W(0.0)
      CALL TIMEOVER(COST)

      PRINT *,'Cost per time call  ',COST

* If ASAT is off due to environment variable, this call does nothing
* If ASAT is on, it reduces priority and turns off gang

      CALL ASAT_INIT()

      CALL GETENV('CHECK_ASAT',ENVFLAG)
      IF ( ENVFLAG(1:1) .EQ. 'Y' .OR. ENVFLAG(1:1) .EQ. 'y' ) THEN
        PRINT *,'ASAT Evaluation will actually be done'
        CHECK_ASAT = .TRUE.
      ELSE
        PRINT *,' To cause the ASAT_EVAL  setenv CHECK_ASAT y '
        CHECK_ASAT = .FALSE.
      ENDIF

      OPEN(UNIT=10,name='testskew.csv')

      PRINT *,'Wasting a timeslice in parallel ...',MSF77W(0.0)
      CALL KILLTIME()

      PRINT *,'Detail timing the trivial loop ', MSF77W(0.0)

      LASTTIM = MSF77W(0.0)
      DO I=1,TSIZE
        PRE(I) = MSF77W(0.0)

* Do the time check by hand
        IF ( CHECK_ASAT ) THEN
          THISTIM = MSF77W(0.0)
          IF ( (THISTIM - LASTTIM) .GT. 0.1 ) THEN
            CALL ASAT_EVAL()
            LASTTIM = THISTIM
          ENDIF
        ENDIF
        IN(I) = MSF77W(0.0)


C$DOACROSS LOCAL(J),SHARE(XIN,I),MP_SCHEDTYPE=SIMPLE,CHUNK=1
        DO J=1,4
          XIN(I,J) = MSF77W(0.0)
        ENDDO
        OUT(I) = MSF77W(0.0)
      ENDDO

      DO I=1,TSIZE
        ASAT(I) = IN(I) - PRE(I) - (COST)
        LOOP(I) = OUT(I) - PRE(I) - (3 * COST)
        IF ( I .LT. 5 ) print *,PRE(I),IN(I),OUT(I),ASAT(I),LOOP(I)
        DO J=1,4
          SPAWN(I,J) = XIN(I,J) - IN(I) - (COST)
          BAREND(I,J) = OUT(I) - XIN(I,J) - (COST)
        ENDDO
      ENDDO

      PRINT *,'Writing Results ...',MSF77W(0.0)

* End-run Print outs

      WRITE(10,999)
999   FORMAT(1X,'LOOP,ASAT,SP1,SP2,SP3,SP4,BAR1,BAR2,BAR3,BAR4')
      DO I=1,TSIZE
        WRITE(10,1000)LOOP(I)*1000000,ASAT(I)*1000000,
     +   (SPAWN(I,J)*1000000,J=1,4),(BAREND(I,J)*1000000,J=1,4)
1000    FORMAT(1X,E13.7,11(',',E13.7))
      ENDDO

      END
      SUBROUTINE NOTHING()
      RETURN
      END
```

BIBLIOGRAPHY

[1] Anderson T, Bershad B, Lazowska E, Levy H, "Scheduler Activations: Effective Kernel Support for the User-level Management of Parallelism", Proceedings of the 13th ACM Symposium on Operating System Principles, pages 95-109.

[2] Baden, S "Programming Abstractions for Dynamically Partitioning and Coordinating Localized Scientific Calculations running on Multiprocessors", SIAM Jan 1991, pages 145-157.

[3] Beazley D, Lomdahl P, "Message-Passing Multi-Cell Molecular Dynamics on the Connection Machine 5", To appear in Parallel Computing 1993. Also available on line as: http://www.acl.lanl.gov/Lomdahl/t.ps

[4] Bhatt S, Chen M, Cowie J, Lin C, Liu P, "Object-Oriented Support for Adaptive Methods on Parallel Machines", Scientific Programming, V 2 n 4 Winter 1993, pages 179-192.

[5] Casavant TL Kuhl JG, "A Formal Model of Distributed Decision Making and its Application to Distributed Load Balancing", Proc 6th International conference Distributed Computer Systems, IEEE, pages 232- 239,1986.

[6] Casavant TL Kuhl JG, "Analysis of Three Dynamic Distributed Load-Balancing Strategies with Varying Global Information Requirements", Proc Int Conference Distributed Computing Systems 1987 pages 185-191

[7] Convex Computer Corporation, "Convex Architecture Reference Manual (C-Series)", Document DHW-300, April 1992.

[8] Convex Computer Corporation, "Convex Architecture Reference Manual (Exemplar)", Document DHW-940, March 1993..

[9] Autotasking User's Guide, CRAY Research Inc., SN-2088, 1988. http://www.epcc.ed.ac.uk/t3d/documents/cray/cf77optim.60/

[10] Elesser G, Ngo V, Sourav B, Wei-Tek T, "Processor Preallocation and Load Balancing of DOALL Loops", The Journal of Supercomputing, 8 135-161 (1994).

[11] Kuck & Associates Inc., "Guide™ Reference Manual, Version 2.0", Document #9603002, March 1996.

[12] Kennedy K, Koelbel C, et. al. "High Performance Fortran Forum (HPFF)", Scientific Programming, Vol. 2, no. 1-2, pp. 1-170, John Wiley and Sons, On-line information on HPF is available on http://www.erc.msstate.edu/hpff/home.html

[13] High Performance Fortran Forum, "HPF-2 Scope of Activities and Motivating Applications" Version 0.8, November 1994 ftp://hpsl.cs.umd.edu/pub/hpf_bench/hpf2/hpf2.html

[14] Krueger P, Chawla R, "The Stealth Distributed Scheduler", Proceedings International Conference on Distributed Computer Systems, 1991, pages 336-343.

[15] Litzkow M, Livney M, Mutka M,, "Condor - A Hunter of Idle Workstations", Proceedings Eighth International Conference on Distributed Computing, 1988, pages 104-111.

[16] Liu J, Saletore V, Lewis T, "Scheduling Parallel Loops with Variable Length Iteration Execution Times of Parallel Computers," Proc. of ISMM 5th Int. Conference. on Parallel and Dist. Systems, 1992.

[17] Liu J, Saletore V, "Self Scheduling on Distributed-Memory Machines," IEEE Supercomputing '93, pp. 814-823, 1993.

[18] Liu J, Saletore V, Lewis T, "Safe Self Scheduling: A Parallel Loop Scheduling Scheme for Shared-Memory Multiprocessors", International Journal of Parallel Programming, Vol 22, No 6, 1994, pages 589-614.

[19] Load Sharing Facility, Platform Computing, http://www.platform.com/

[20] Mansour N, Fox G, "Parallel Physical Optimization Algorithms for Allocating Data to Multicomputer Nodes", Journal of Supercomputing V 8 n 1 March 1994, pages 53-80.

[21] J. C. Mogul and A. Borg, The Effect of Context Switches on Cache Performance, DEC Western Research Laboratory TN-16, Dec., 1990. http://www.research.digital.com/wrl/techreports/ /abstracts/TN-16.html

[22] Nichol, Salz "An Analysis of Scatter Decomposition", IEEE Transactions on Computers, November 1990, pages 1153-1161.

[23] Ousterhout J, "Scheduling Techniques for Concurrent Systems", Distributed Computing Systems Conference, pages 22-30, 1982.

[24] Polychronopoulos C, Kuck D, "Guided Self Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," IEEE Transactions on Computers, Dec. 1987.

[25] Polychronopoulos C, "Parallel Programming and Compilers", Kluwe Academic, Boston.

[26] Severance C, Enbody R, Software-Based, Automatic, Self-Adjusting Threads (ASAT) for Parallel Supercomputers, Submitted to : Supercomputing '94 April 1, 1994, Also available as: Michigan State University Computer Science Department Technical Report CPS-94-17. http://clunix.msu.edu/~crs/papers/asat_sgi

[27] Severance C, Enbody R, "Automatic Self-allocating Threads (ASAT) on the Convex Exemplar", June 1994, Revised September 1994, http://www.egr.msu.edu/~crs/papers/asat_ex/index-old.html

[28] Severance C, Enbody R, "Software-Based, Automatic, Self-Adjusting Threads (ASAT) for Parallel Supercomputers", Poster Session, Supercomputing 1994, http://www.egr.msu.edu/~crs/papers/super_94/asat.html

[29] Severance C, Enbody R, Wallach S, Funkhouser B, "Automatic Self-allocating Threads (ASAT) on the Convex Exemplar" Proceedings 1995 International Conference on Parallel Processing (ICPP95), August 1995, pages I-24 - I-31.

[30] Severance C, Enbody R, Petersen P, "Managing the Overall Balance of Operating System Threads on a Multiprocessor using Automatic Self-Allocating Threads (ASAT), to appear in Journal of Parallel and Distributed Computing (JPDC) Special Issue on Multithreading for Multiprocessors.

[31] Severance C, Enbody R, , "Automatic Self-allocating Threads (ASAT) on the SGI Challenge", to appear in " Proceedings 1996 International Conference on Parallel Processing (ICPP96), August 1996.

[32] Severance C, Enbody R, "A Possible Addition to HPF 3.0 - Real Valued Indexed (RVI) Arrays", Proceedings High Performance Computing 1996, Society for Computer Simulation, Pages 248-253.

[33] Silicon Graphics, Inc., "Power FORTRAN Accelerator User's Guide," Document 007-0715-040, 1993.

[34] Silicon Graphics, Inc., "FORTRAN77 Programmer's Guide," Document 007-0711-030, 1993.

[35] Silicon Graphics, Inc., "Symmetric Multiprocessing Systems," Technical Report, 1993.

[36] Stumm M, "The Design and Implementation of a Decentralized Scheduling Facility for a Workstation Cluster", Proceedings of the Second Conference Computer Workstations, 1988, pages 12-22.

[37] Tucker A, "Efficient Scheduling on Multiprogrammed Shared Memory Multiprocessors", Phd. Thesis, Stanford University, December 1993. CSL-TR-94-601 from http://elib.stanford.edu/

[38] Tucker A, Gupta A, "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors," ACM SOSP Conf., 1989, p. 159 - 166.

[39] Tzen T. and Ni L., "Dynamic Loop Scheduling on Shared-Memory Multiprocessors", in Proc. of Int. Conf. on Parallel Processing, 1991, pp 247-250.

[40] Tzen T. and Ni L., "Trapezoidal Self-Scheduling: A Practical Scheduling Scheme for Parallel Compilers", IEEE Transactions on Parallel and Distributed Systems, 4(1):87-98 (1993).

[41] Yue K, Lilja D "Loop-Level Process Control: An Effective Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors," Workshop on Job Scheduling Strategies for Parallel Processing, International Parallel Processing Symposium, Springer-Verlag Lecture Notes in Computer Science, Vol. 949, April 1995, pp. 182-199.

[42] Yue K, Lilja D, "Parallel Loop Scheduling for High-Performance Computers," High Performance Computing: Technology, Methods, and Applications, Elsevier Publishing Company, Amsterdam, September 1995, pp.243-264.

[43] Yue K, Lilja D , "Efficient Execution of Parallel Applications in Multiprogrammed and Multiprocessor Systems", International Parallel Processing Symposium, April 1996, pp. 448-456.