AUTOMATIC SELF-ALLOCATING THREADS (ASAT) ON AN SGI CHALLENGE

Charles Severance and Richard Enbody*
Department of Computer Science
Michigan State University
East Lansing, MI 48824-1027
crs@msu.edu, enbody@cps.msu.edu

Automatic Self Allocating Threads (ASAT) is proposed as a way to balance the number of active threads across a shared-memory multiprocessing system. Our approach 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 the run times of all jobs improve when it is in use. In this paper we consider the application of ASAT for improving the scheduling of threads on an SGI Challenge. We demonstrate how the number of threads of an ASAT job adjusts to the overall system load to maintain thread balance and improve system throughput.

INTRODUCTION

A multi-threaded runtime environment which supports lightweight threads can be used to support many aspects of parallel processing including: virtual processors, concurrent objects, and compiler run-time environments. However, such a library must depend on the underlying thread mechanism provided by the operating system. Threads working on compute intensive tasks work best when there is one thread performing real work on each processor. Matching the number of running threads to the number of processors can yield both good wall-clock run time and good overall machine utilization. The challenge is to schedule threads to maintain one running thread per processor by dynamically adjusting the number of threads as the load on the machine changes. It is generally not efficient to involve the operating system during a thread switch between lightweight threads. As such, a lightweight thread must operate within the parameters provided by the operating system.

If an application runs on a dedicated system with a known number of available processors, a multi-threaded run-time environment can utilize a known number of operating system threads and assume that each operating system thread will have relatively uninterrupted access to CPU resources. However, it is much more common to operate in an environment in which resources are shared by a number of multi-threaded applications running on the same multiprocessing system. This work is directed at implementing efficient multi-threaded runtime environments in such a shared environment. This work identifies the situations on a multiprocessing system when the operation of a lightweight thread environment might be negatively impacted by other threads running on the system.

The paper consists of several parts. (1) A proposed mechanism (ASAT) which allows processes to adjust their thread usage to maximize overall system utilization, (2) A characterization of the performance impact of having the improper number of threads on a multiprocessing system, and (3) an experiment using this technique in a multi-threaded compiler run-time environment.

EXECUTION MODEL

This work focuses on an execution model in which a serial portion of the code is periodically executed between the parallel sections of the code.

In a procedural-language environment such as FORTRAN, a loop similar to the following will generate that pattern:

*This work is based on work supported by the National Science Foundation under Grant No MIP-0209402.

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

The parallel portions of the code may be executed by any number of operating system threads. This work focuses on how to insure that the right number of operating system threads are used each time the parallel code is executed.

Our approach does not necessarily apply to all multi-threaded environments. Database or network server environments may want to have significantly more operating system threads than available CPU resources in order to mask latencies due to I/O from the network, disk or other sources.

1 PREVIOUS WORK

In [11] the problem of matching the overall system-wide number of threads to the number of processors was studied on an Encore Multimax. They identified a number of the major problems with having too many threads including:

1. Preemption during spin-lock critical section,
2. Preemption of the wrong thread in a producer-consumer relationship,
3. Unnecessary context switch overhead, and
4. Corruption of caches due to context switches (also see [4]).

The general topic of scheduling for parallel loops is one that is well studied. 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. An excellent survey of these techniques is presented in [3].

The implementation of these techniques on most shared-memory parallel processors works with a fixed number of threads determined when the program is initially started. For the purpose of this paper, we call this technique Fixed Thread Scheduling (FTS). The FTS approach is reasonable for many of the existing parallel processing systems as long as each application has dedicated resources. As we point out in this paper, not having a dedicated system can seriously degrade the effectiveness of the FTS approach.

Other dynamic, run-time, thread management techniques which are geared toward compiler detected parallelism include: Automatic Self-Adjusting Processors (ASAP) from Convex [1] and Autotasking on Cray Research [2] computers.

A previous study of the benefits of Automatic Self-Allocating Threads (ASAT) for the Convex Exemplar was done in [6], details on multiple ASAT jobs appears in [7].

ASAT

The general goal of our Automatic Self-Allocating Threads (ASAT) is to eliminate thread imbalance by detecting thrashing and then dynamically reducing the number of active threads to achieve balanced execution over the long term. In this way, multi-threaded 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. The most important are:

1. Detecting if too many active threads exist.
2. Detecting if too few active threads exist.
3. Adjusting the number of threads.

ASAT takes advantage of the basic parallel loop structure shown earlier. 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 create the precise number of threads which match the available processors.

A critical concept of ASAT is that a job will examine the availability of system resources with respect to current system load. The process is accurate, efficient and completely decentralized. The thread imbalance detected is for all threads currently on the system, not simply for this job's threads. Whether other jobs are scheduled using ASAT doesn't matter. However, the stability of multiple ASAT jobs is an important question we examine later in the paper.

ASAT uses a timed barrier test to detect thread imbalance on the system. A special barrier routine is inserted to test the system while executing as a

single thread. Using the clock, the elapsed time between the first thread entering the barrier and the last thread leaving the barrier is measured. There is a three-orders of magnitude difference between barrier passage times under thread-balanced and thread-imbalanced conditions. That difference is significant enough to make the barrier a good test for load imbalance.

The interval between barrier evaluations can be adjusted. We set the ASAT software to only run the barrier test once every 1 second 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 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. We call it "dipping your toe in the water." If the number of threads we are using has been working smoothly for a while, we test with more threads for a single barrier. If this barrier runs well, we 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 spawn opportunity.

ASAT IN A COMPILER RUN-TIME ENVIRONMENT

The basic goal of ASAT is to allow a multithreaded run-time environment to operate most efficiently in an environment where the overall load on a system changes dynamically.

The first multi-threaded runtime environment which we have investigated is a compiler run-time environment. For this study, ASAT was implemented without modifications to the actual compiler library. Because it is not implemented inside the compiler library, the calls to ASAT must be explicitly added to the application. The two routines are ASAT_INIT and ASAT_ADJUST. ASAT_INIT is called at the beginning of the program before any parallel loops have executed and ASAT_ADJUST is called periodically outside of a parallel loop. A highly stylized example is as follows:

```
      CALL ASAT_INIT()
      DO ITIME=1,INFINITY
      CALL ASAT_ADJUST()
C$DOACROSS LOCAL(I),SHARE(PARTICLE),SCHED(GSS)
        DO IPART=1,10000
           Work..
        ENDDO
      ENDDO
      END
```

Once ASAT is supported directly by the compiler, its use can be controlled using a directive.

```
C$DOACROSS LOCAL(I),SHARE(PARTICLE),
C$         SCHED(GSS),THREADS(ASAT)

C$DOACROSS LOCAL(I),SHARE(PARTICLE),
C$         SCHED(GSS),THREADS(FTS)
```

It it important to separate the thread management aspects from the chunking and work distribution issues. Work distribution techniques such as Guided Self Scheduling (GSS) depend of the variation of the length of each iteration. Thread management simply controls the number of threads which are used to process the work. Most compiler run-time libraries are designed to check the number of threads at the beginning of each parallel section.

AN EXISTING MECHANISM

An good example of dynamic thread balancing is the mechanism available on the Convex C-Series (C-240, C-3X00, C4XXX) supercomputers is called Automatic Self Allocating Processing (ASAP) [1]. We use ASAP as a model for comparison.

The ASAP processing in the Convex C-Series systems is made possible because of an architectural feature called "Communication Registers" 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 parallize loops without regard for the number of threads which will actually execute in the parallel loop. An idle CPU can dynamically create thread and "join" a parallel computation with a very small overhead.

This hardware support allows users to compile their applications assuming a generalized parallel environment regardless of whether or not there will be

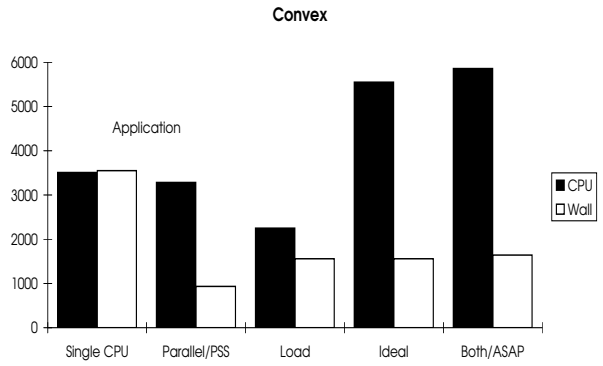

Figure 1: Performance of the Convex on Parallel Jobs

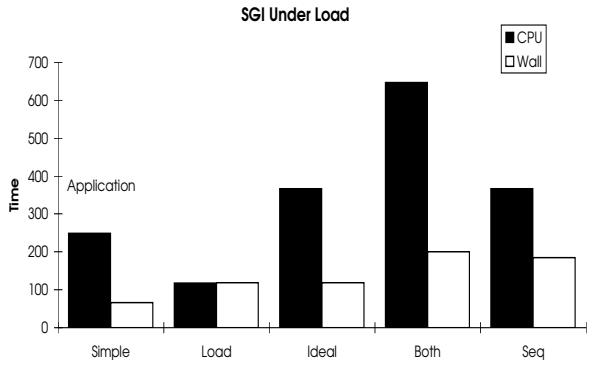

Figure 2: Performance of the SGI Under Load

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.

Throughout this section, a simple, very parallel computation will be used as the benchmark application. The kernel for these tests is as follows:

```
C$ DO_PARALLEL
        DO J=1,100000
            // 3Flops, 5 Memory references,
            // no data dependencies
        ENDDO
```

Figure 1 shows the performance of the code with several compiler options and load scenarios. The first pair of bars shows the CPU time (dark) and wall time (white) for the application on a single CPU. The second pair of bars shows the performance of the same application on four CPUs. The third pair of bars is another application which is single-threaded and cannot run in parallel. The fourth pair of bars shows the CPU and wall time for the ideal combination of the two codes assuming perfect load balancing on four CPUs. In this case, the ideal CPU time is the sum of the individual times and the wall time is the maximum of the individual wall times. The last pair of bars shows the actual performance achieved on the Convex C-240 when the jobs are run together. In the actual run using ASAP both the CPU time and the wall time are essentially the same as the ideal times (approximately 1.05 times longer).

PARALLEL APPLICATIONS AND LOAD ON THE SGI

When multiple jobs are run on a less tightly coupled parallel machine the competing jobs can show dramatic interference with each other. Figure 2 shows what happens when the experiment performed on the Convex (Figure 1) is performed on a loaded and unloaded 4-CPU SGI Challenge system.

As on the Convex, the application code parallelizes automatically without any user modifications. Like the Convex, the load application only runs on a single CPU. However, unlike the Convex, the system performs much worse than ideal when both codes 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 (i.e. submitted the jobs to a batch queue).

COMPILER OPTIONS ON THE SGI

The SGI has several compiler options for load loop scheduling provided as part of its parallel FORTRAN compiler [8] [9]. Similar options are typically available on most parallel FORTRAN compilers. Are these compiler options sufficient to solve the unbalanced threads problem? The scheduling options for a parallel loop on the SGI include:

Simple At the beginning of a parallel loop each thread takes a fixed number of iterations of the loop.

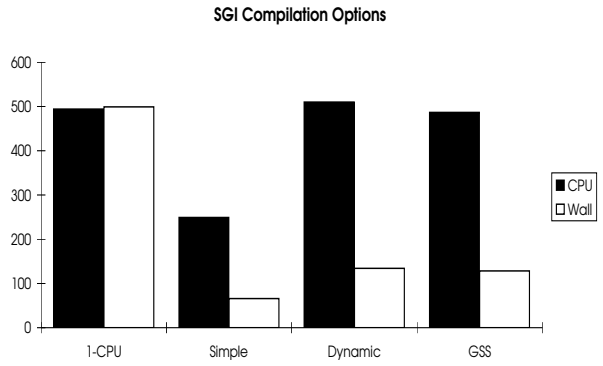

Figure 3: SGI Compiler Options

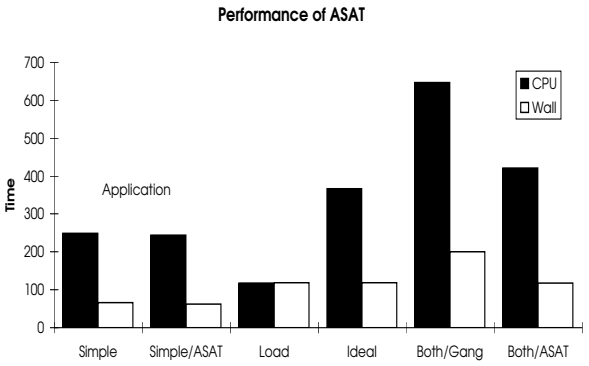

Figure 5: Performance of ASAT on the SGI

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 based on the application.

Guided Self Scheduled This is essentially a modification of Dynamic scheduling except that large "chunks" are taken during the first few iterations, and the "chunksize" is reduced as the loop nears completion. GSS is designed to even out wide variations in the execution times of the iterations of the parallel loop. GSS is described in [5].

Figure 3 shows parallel performance of the simple application on an unloaded 4-CPU SGI with various compiler options:

The Dynamic and GSS options add overhead to the loops. Unlike the Convex, this overhead is in software and has a greater impact on the performance of the application. These options do not affect the allocation of threads so they only partially solve the the problem of having too many threads in a loaded system.

PERFORMANCE OF ASAT

In this section we show that adding ASAT to the SGI allows it to run with a balanced number of threads. In addition, we show how competing jobs interact with each other. Figure 4 shows how ASAT generally operates when working on a system with variable load. In this figure, an application using ASAT is executing while other users are using the system. As the load average increases due to other users, the ASAT application releases threads to maintain its 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.

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

The first test is to duplicate the experiment which was performed for Figures 1 and 2 using ASAT to schedule the threads in the application code. Simple scheduling was used along with ASAT.

There are several observations about Figure 5. 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 while the load was running. The ASAT job runs at a lower priority than the load job so the load job got 100 percent of the CPU for the duration of its run. 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 1. Also from Figure 5, 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.

To test ASAT under more varied load patterns, two time-oriented tests were performed. 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:

1. One job that averaged 5 minutes CPU time and arrived approximately every 15 minutes

2. Three jobs that averaged 1 minute of CPU time and arrived approximately every 4 minutes

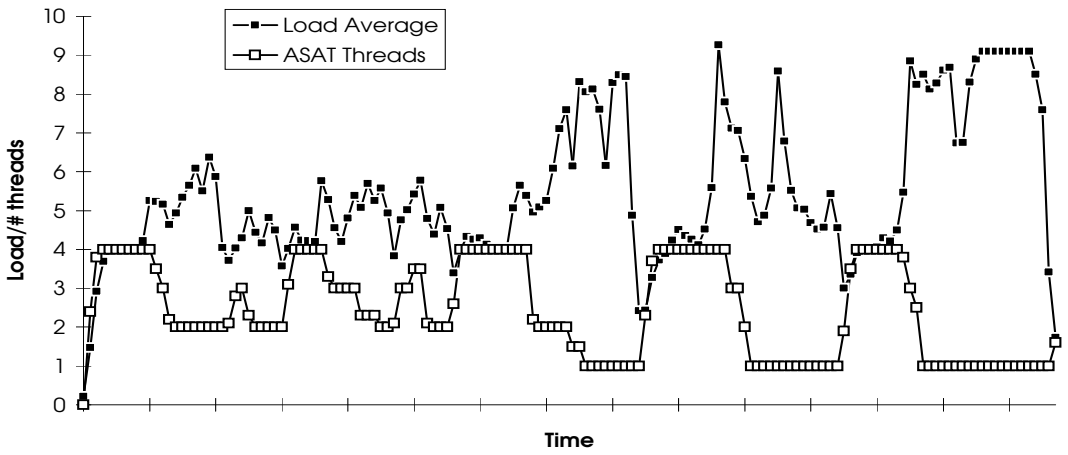

Figure 4: Example operation of ASAT

These load jobs were all sequential and were given higher priority than the ASAT application. The system load for the combination of "load" jobs is shown as the bottom plot in Figure 6.

Gang scheduling and ASAT are compared in Figure 6. In the figure, the combination "load" 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 load. 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 had a significant negative impact on the other jobs.

The second time-oriented test is exactly the same as the previous test except that the applied load is more regular. In 2.5 minute intervals, the load is increased from 1 to 4 and then back down to zero. This applied load is shown in Figure 7 as an inverted "V" representing the increase in threads followed by a decrease. The same ASAT and gang processes were each run together with this new load profile.

Figure 7 again compares gang vs. ASAT—the former is the top line and the latter is the second line. The figure also shows the load by itself (inverted "V") and the number of ASAT threads. As the load is increased over the time of the run, ASAT quickly adjusts the number of threads, maintaining system balance. When the load goes up, the number of ASAT threads goes down. As resources free up, the number of ASAT 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.

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 performance impact of having a system with an unbalanced number of threads was investigated.

ASAT is proposed as a technique which is easily implementable in a run-time library and effectively balances thread use across an entire system. As load increased on the whole system as ASAT job dynamically reduced its threads. When system load decreased the ASAT job dynamically increased its threads to soak up available cycles.

ASAT is examined in the context of a FORTRAN run-time thread management environment. The performance of ASAT is shown to be superior to the existing compiler-provided scheduling mechanisms in SGI Power FORTRAN. ASAT performs nearly as well in diverse load situations as the hardware approach used by Convex ASAP.

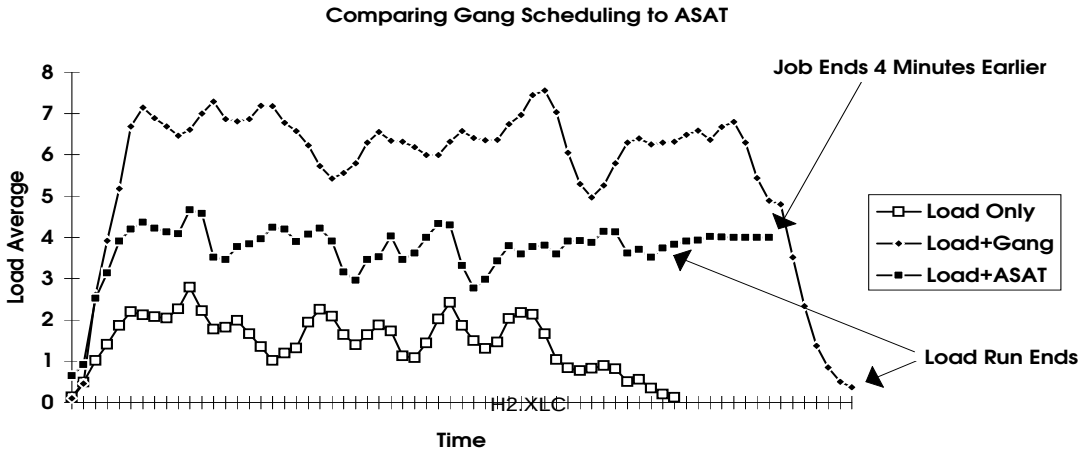

Figure 6: ASAT Response to Rapidly Changing Load

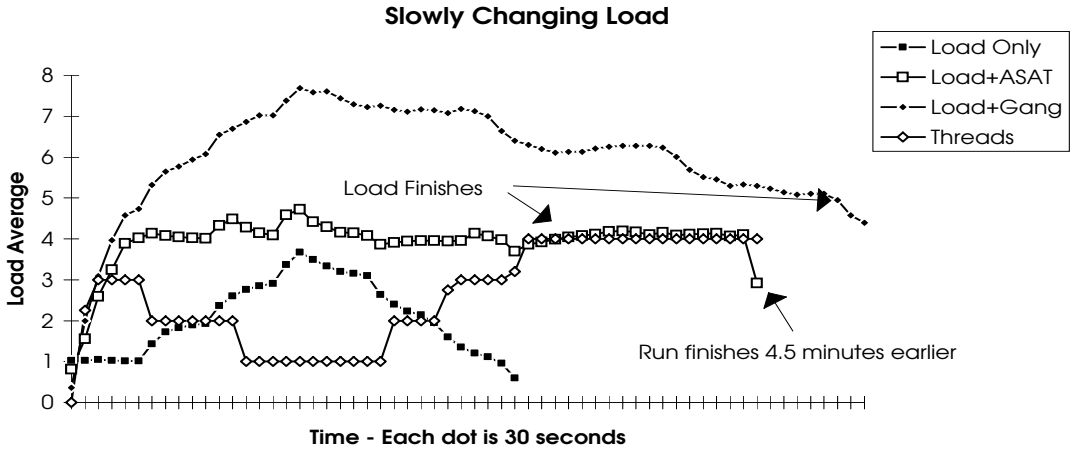

Figure 7: ASAT and Slow changes in Load

FUTURE WORK

We need to further study how to best implement ASAT using more compiler and operating system modifications. ASAT, as currently implemented, does not make or require any operating system changes. One operating system change we believe 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.

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. We hope to have a version of ASAT available via anonymous FTP. Please check the URL http://clunix.msu.edu/~crs/projects/asat for details on the availability of ASAT.

Thanks to: David Kuck and Paul Petersen, Kuck and Associates; Jerry McAllister, Michigan State University; Dave McWilliams, National Center for Supercomputing Applications and Lisa Krause, Cray Research.

References

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

[2] Cray Research, *CF77 Compiling System, Volume 4: Parallel Processing Guide.*

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

[4] 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

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

[6] Severance C, Enbody R, Wallach S, Funkhouser B, "Automatic Self Allocating Threads (ASAT) on the Convex Exemplar" Proceedings 1995 International Conference on Parallel Processing (ICPPP95), August 1995, pages I-24 - I-31.

[7] Severance C, Enbody R, Peterson P, "Managing the Overall Balance of Operating System Threads on a MultiProcessor using Automatic Self-Allocating Threads (ASAT)," *Journal of Parallel and Distributed Computing* Special Issue on Multithreading on Multiprocessors, to appear.

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

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

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

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