Managing the Overall Balance of Operating System Threads on a MultiProcessor using Automatic Self-Allocating Threads (ASAT)

Authors: Charles Severance and Richard Enbody Department of Computer Science Michigan State University East Lansing, MI 48824 crs@msu.edu, enbody@cps.msu.edu Paul Petersen Kuck and Associates, Champaign, IL 61820, petersen@kai.com

Abstract

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. This work focuses on the area of compiler run-time environments. 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 run-time environemnt must operate within the parameters provided by the operating system. 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 performance impact of these other threads is measured. A solution called Automatic Self Allocating Threads (ASAT) is proposed as a way to balance the number of active threads across a 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. The improvements are achieved by eliminating contention for resources by jobs on the system. Furthermore, our approach uses self-scheduling so it is implemented in a run-time environment rather than in an operating system. Finally, the self-scheduling means that jobs need not all be scheduled by ASAT.

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

1 INTRODUCTION

The primary purpose of a lightweight thread library is to allow "context switches" between threads to occur in user space without operating system intervention. The overhead of involving the operating system in these lightweight thread switches would make the switch take too long. The foundation upon which a multi-threaded run-time environment is built is the thread environment 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.

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

2 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:

```
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 is 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 processor resources in order to mask latencies due to I/O from the network, disk or other sources.

3 PREVIOUS WORK

In [TukGup89] the problem of matching the overall system-wide number of threads to the number of processors was studied on an Encore Multimax. Their applications used explicit parallelism and a thread library. The thread library communicated with a central server to insure overall thread balance. They identified a number of the major problems with having too many threads including:

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

In addition, they provide an excellent survey of related work.

As the speed of the CPU's has increased, the problem of a context switch corrupting cache has become an increasing performance impact. In [MoBo90], 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 processor. 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.

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. A number of scheduling techniques have been proposed and implemented. An excellent survey of these techniques is presented in [LiuSal93]. These techniques include Pure Self Scheduling (PSS), Chunk Self Scheduling (CSS), Guided Self Scheduling (GSS) [PolKuck87], Trapezoidal Self Scheduling (TSS) [TzenNi91], and Safe Self Scheduling (SSS) [LiuSal92].

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 [CSERArch] and Autotasking on Cray Research [Cray] computers. Convex ASAP is based on hardware extensions to the architecture and requires very little run-time library support. Cray's Autotasking is a software based approach and is supported in the run-time library and the operating system scheduler.

The Cray Autotasking product is similar to our work. The primary differences between Autotasking and ASAT are: 1) ASAT is targeted toward a lower cost environment with commodity processors using a coherent cache and 2) the initial version of ASAT requires no operating system or run-time library modifications and can be used by an end user or system administrator.

A previous study of the feasibility of Automatic Self-Allocating Threads (ASAT) for the Convex Exemplar [CONEXMP] was done in [SevEn95].

4 ASAT Design

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:

- 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 serial-parallel loop structure shown earlier and only adjusts the thread count when the applications are running single threaded. Given the current state of automatic parallelizing compilers, the 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. Most applications which use compiler-generated parallelism will not exhibit this behavior.

5 ASAT IMPLEMENTATION

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 pseudo code is as 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 = max(leaving);
  passage = last_out - first_in;
  return(passage);
}

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

The interval between barrier evaluations can be adjusted. We set the ASAT software to only run the barrier test at most 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 application is attempted with one more thread. We call it "dipping your toe in the water." If the number of our application threads 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.

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++;
 }
}
```

6 Basic Performance Results

Our goal is to develop thread-scheduling techniques which work on machines that are running multiple jobs. We study three techniques and compare them under a variety of system loads. We also vary the size of the applications tested.

Experiment Description

The experiments in this paper were generated on a four-processor SGI Challenge-L [SGIArch] with MIPS-R4400 processors running at 150Mhz with a combined memory of 384MB. Each R4400 has a 32K L1 Cache (16K instruction and 16K data), and a 1MB L2 cache. In this section, three types of run-time approaches are compared: A fixed-thread scheduled (FTS) approach, a single-threaded approach, and an ASAT approach. The three "approaches" are the same code rearranged slightly for different scheduling. All three represent the same amount of work. The experiments in this paper were performed using a experimental version of the Guide parallel programing environment from Kuck & Associates, Inc. [Guide].

In an ASAT run, the job makes a call to asat_adjust_threads before each instantiation of the parallel loop. As a result, the jobs with the smaller grain-size (defined below) will have proportionally more calls to asat_adjust_threads. The ASAT_EVAL_TIME parameter is set to one second. Throughout the execution of these runs, the ASAT run-time dynamically adjusts the threads in use by the runtime system as the job progresses.

The fixed-thread scheduled (FTS) application on a 4-processor machine spawns four operating system threads so that it can use all four processors, if they are available. In an FTS run, the run-time library allocates four threads which are gang scheduled. The work is evenly divided up among the threads using equal chunks to minimize cache interference between threads within a single process. It runs well when all four processors are free. At the join point at the end of the parallel loop, the run-time library is set to release the CPU after a relatively short amount of wasted spin time. This allows the CPU to be rescheduled to a thread which is performing useful work.

The single-threaded process is run using a single thread. When a single copy of this type of job is executed, the system is only 25 percent utilized because the process cannot take advantage of the available processors with only a single thread. When multiple copies are run, one copy (one thread) is run on each processor.

This set of experiments studies a loaded system as the number of application threads varies. To keep the variability under control, we run multiple copies of the same job. For FTS jobs, the number of threads increases by four each time another job is added. The single-threaded application simply schedules a job's single thread to an otherwise unloaded processor. Running multiple copies of jobs allows us to observe both

- the performance under a variety of loads, and
- the interaction of multiple jobs with respect to the scheduling techniques.

The latter point is very important for ASAT since the stability of its decentralized control needs to be observed. The vertical axis is the overall performance of a single job. It is actually the average of the multiple running jobs, but there is essentially no variance except where noted.

In addition to varying the scheduling and system load, we also vary the working set of the application. Each experiment is run with different length parallel sections. We call these lengths the “grain size. The grain size represents both the granularity of the scheduling and the size of the working set. Table 1 summarizes the parameters which are related to grain size. The iteration time and memory size in the table are for a single copy of a single-threaded execution. The overall working set for the system is the memory size in Table 1 multiplied by the number of copies being run simultaneously. Each job does the same number of loads, adds, and stores regardless of the grain size by executing the parallel loop more times for the smaller grain sizes. In this way, the overall wall times can be compared across grain sizes. The pseudo code for the application is as follows:

```
         ITERATIONS = WORK/GRAINSIZE
         DO IGRID=1,ITERATIONS
   *     Select Scheduling Approach
             DO PARALLEL I=1,GRAINSIZE
             A(I) = B(I) + C(I)
             ENDDO
         ENDDO
```

Grain Size	Iteration Time	Memory Size
10K	0.002s	240K
100K	0.040s	2.4M
1M	0.400s	24M
4M	1.600s	96M

Table 1 - Parameters Related to Grainsize

Results for Small Threads

We begin our experiments with an examination of small threads using a grain size of 10K. As described above, 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.

Figure 1 and Figure 2 show the performance results when 1 to 4 copies of each type of job are run with small threads.

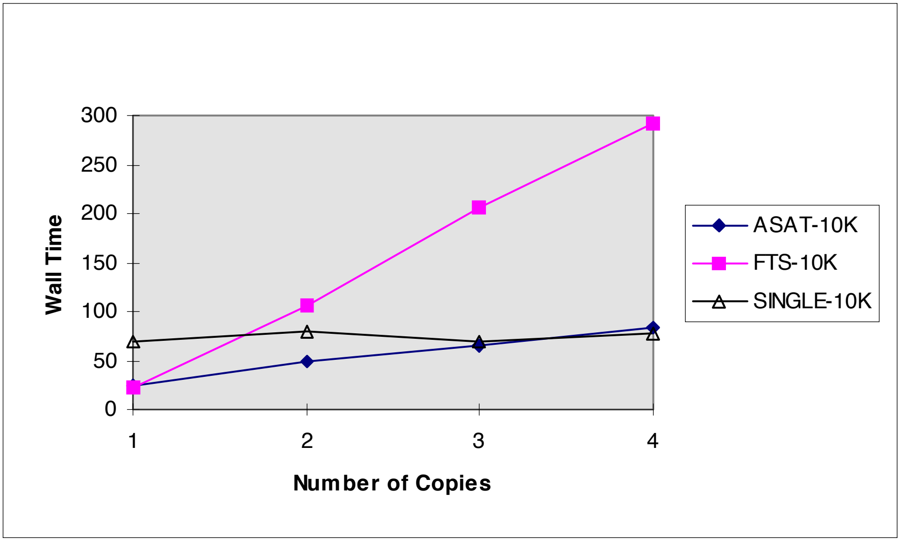

Figure 1 - Multiple Copies Using Small Threads (GS=10K)

The wall times of the single-threaded jobs (SINGLE-10K) in Figure 1 remain relatively constant as more copies are added up to 4 copies. There is no interaction between threads and there is always a processor to execute each of the threads. In the case of a context switch, the operating system generally is able to re-schedule each thread back on the same processor to maximize cache re-use.

Each copy of the FTS job is run with four threads. When only one copy is executed on the system, the additional threads are used to improve the wall time of the computation. As such the lone FTS run executes 2.7 times faster than the SINGLE (Figure 1). However, as multiple copies are added to the system, the overall number of threads managed by the operating system rises from four threads with one copy running up to sixteen threads when four copies are running. These threads are scheduled across the available four processors. As the additional copies are run, the overhead increases significantly resulting in four copies running 21 times slower than one copy of the FTS job.

When running a single copy of the ASAT job on an empty system it operates with four threads like an FTS job and executes the code in parallel. As the number of copies is increased, ASAT detects the changes in load and drops threads to generally maintain a balance of four threads across the entire system (Figure 1).

A key observation from Figure 1 is that, as copies are added, ASAT behaves like the better of the other schemes. The overall wall time of the ASAT as the copies are increased is at least as good as the best of either the FTS or SINGLE times. This is because ASAT operates like FTS when there are system resources available and like SINGLE when there are fewer system resources.

It is interesting to examine how ASAT works as the number of copies (and, hence, threads) increases. When two copies are being run, each job runs with two threads for most of the time. When three copies of the ASAT job run, an ideal static assignment of threads to processes doesn't exist. Under those circumstances the system generally runs with two jobs having one thread and the remaining job with two threads. This inherent imbalance caused the execution times among the three copies of the ASAT application to show the largest variance. A typical variance between the fastest and slowest job would be in the range of 10-25 percent.

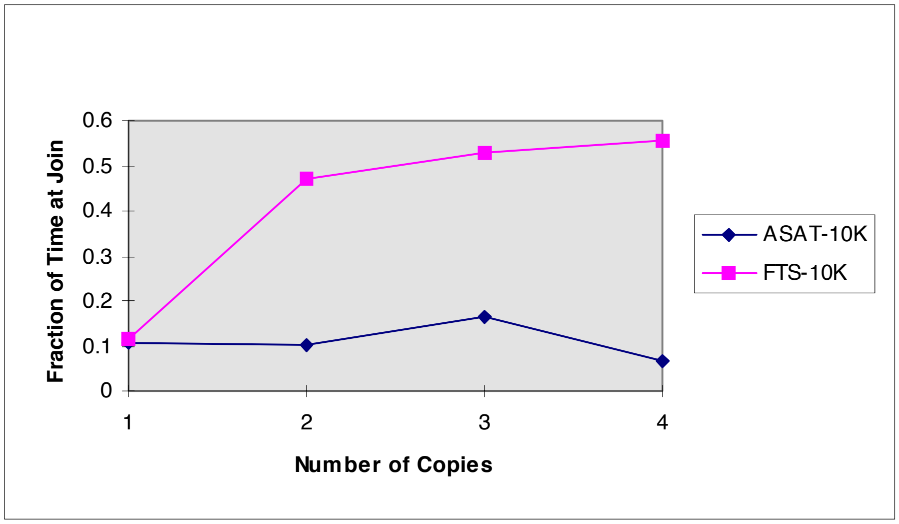

Figure 2 - Multiple Copies' Join Time, Small Threads (GS=10K)

To better understand the causes for the performance shown Figure 1, we examine the time spent a the joins. Figure 2 shows the percent of the wall time spent waiting at joins is measured across all of the processors. Interestingly, even on an empty system about ten percent of the time is spent waiting at the join for both the FTS and ASAT runs. This wastage is one of the reasons that the FTS job only runs 2.7 times faster on four processors than on one.

Note that the single-threaded job (SINGLE) is not shown because it is always zero since only one thread is executing per process so all the threads arrive at the join at the same time.

In the ASAT job, the percent time spent waiting for joins hovers around 10 percent regardless of the number of copies. The slight increase at three copies is due to the fact that there is some swapping of the extra thread among the three processes. For four copies, the time spent at joins drops to six percent because the processes are nearly always operating with one thread. The value is not zero because even in the four-processor case, ASAT may try an additional thread periodically. When that additional thread is tried, the load is unbalanced resulting in poor join waits for that process until ASAT again evaluates load and drops the thread.

It is important to note that the percent time spent at join is wall time, and that it includes both the time spent spinning and the time spent sleeping while waiting for the other threads to arrive. When the thread puts itself to sleep, the processor it is using becomes available for other runnable threads.

The time spent waiting at joins cannot alone explain the increased wall time when four copies of FTS are run. In the small grain size case, the working set of a single process can fit into the secondary cache. Once the secondary cache is warmed up, iterations proceed much more quickly. However, the cache is not large enough to hold all of the jobs' working sets simultaneously. If a round-robin scheduling were assumed, by the time the job was again scheduled its cache might be cold. However, on this machine the time slice allotted to a process is considerably larger than the iteration time for our small threads.

The result is that there is time to fill the cache and run multiple iterations with a warm cache during one time slice.

A further source of negative performance results is due to the cost of putting threads to sleep and waking them back up. Those operations incur operating system overhead with its relatively expensive context switches. In the FTS run with four copies, the sleep-wake cycle occurred ten times more often than in the ASAT run.

This combination of context switches, wasted join time, additional system calls, and increased cache misses, cause the FTS job with 4 copies to run 3.5 times slower than the either ASAT or SINGLE.

Again, the important observation is that the ASAT job adapts, and behaves like a single-threaded job when appropriate and behaves like an FTS job when appropriate.

Results for Medium-sized Threads

In Figure 3 and Figure 4 the same run is executed with medium-sized threads (grainsize=100K). With a working set size per process of over 2MB, the jobs can no longer fit in the 1MB secondary cache. A job now evicts its own cache entries as it continues to execute on a processor.

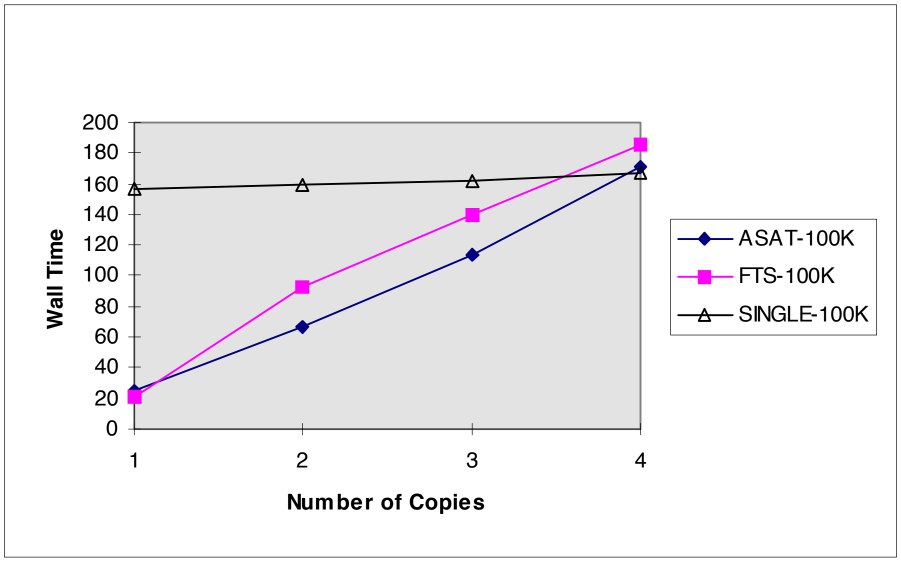

Figure 3 - Multiple Copies Using Medium-Sized Threads (GS=100K)

The first impact of this change in working set size is that the execution time for the SINGLE job increases by roughly a factor of two. The increased size of the working set is causing the job to work out of the cache so more memory references are now being filled from main memory across the system bus. On the other hand, when a single ASAT or FTS job is run, it evenly divides the work among processors resulting in an effective working set size of roughly 600K per processor which can still fit in the secondary cache. Of course, the job will stay in a cache only as long as there are no cache conflicts and the operating system is able to properly schedule the threads to maintain the warm caches. Given that the jobs using FTS and ASAT run about 8 times faster than the SINGLE job on an empty system, it would seem that the operating system is quite effective in scheduling threads on the processors with warm caches.

When we increase to four running copies, there are now four working sets to fit in the caches. As we observed in the single-threaded case an entire working set doesn't fit on one processor's cache so the combined four copies cannot fit on the combined four processors' caches. The result is that the performance of four copies is roughly the same for all of the jobs. The SINGLE and ASAT hold a slight performance advantage.

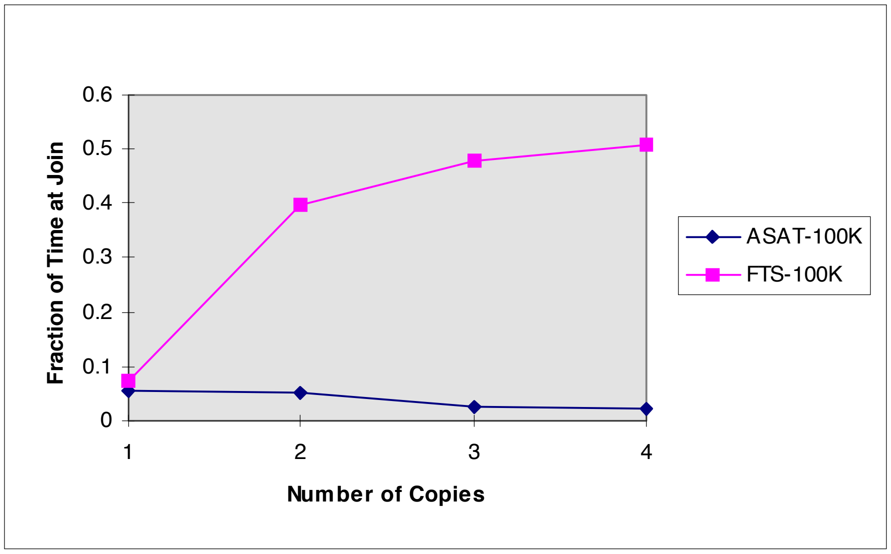

Figure 4 - Multiple Copies' Join Time, Medium-Sized Threads (GS=100K)

To better understand the performance shown in Figure 3 consider the join times of Figure 4. Observe that the time spent waiting at joins is still roughly 50 percent for the FTS job and 10 percent for the ASAT job. These times are similar to the results for the 10K case of Figure 4. In terms of how often a thread is put to sleep, the FTS job put a thread to sleep six times more often than the ASAT job. Also, because of the increased grain size, the length of the sleep time tended to be longer.

Results for Larger Threads

As we move to larger threads (grainsize=1M, Figure 5, Figure 6) and its working set size of 24M per process, we are working out of cache to the extent that cache effect is no longer a discriminating factor. Effectively, at this grain size, the application never experiences any significant *additional* data cache misses caused by context switches.

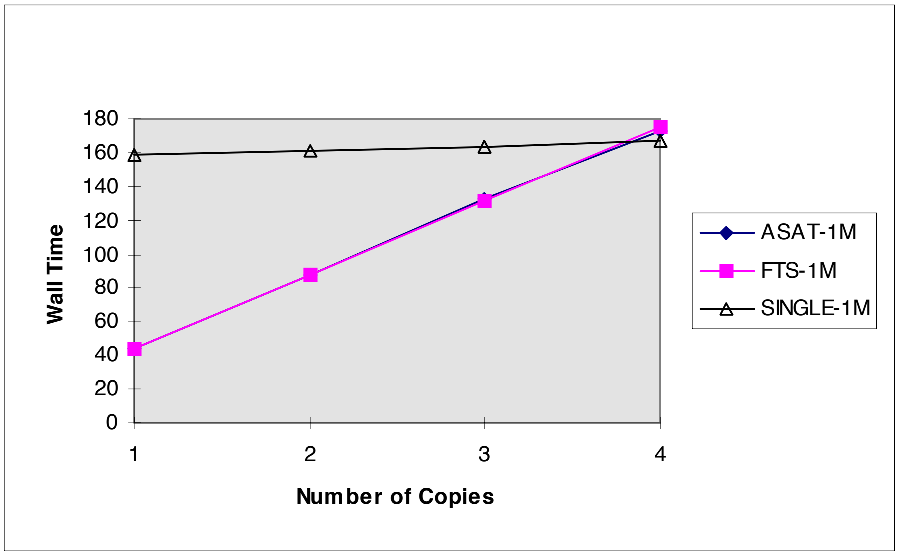

Figure 5 - Multiple Copies Using Large Threads (GS=1M)

At the grain size of 1M, the performance of the FTS and ASAT jobs is now nearly identical. There are a number of factors which cause this. (1) Data is seldom reused in cache, so there is little advantages 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. (3) The joins only happen every 0.1 to 0.4 second so the proportion of processor time wasted at joins is not significant.

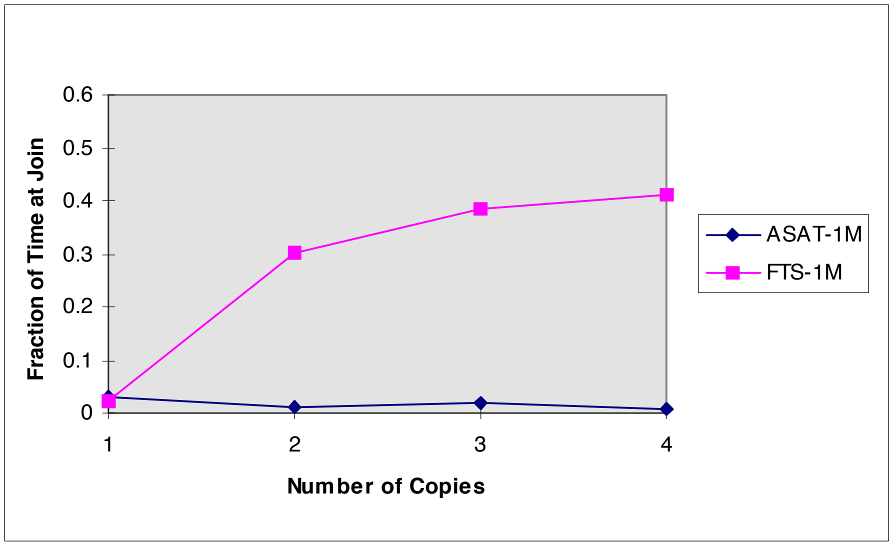

Figure 6 - Multiple Copies' Join Time Using Large Threads (GS=1M)

With large threads, the threads may still arrive at a join unevenly, but when the thread is put to sleep the processor can work on an active thread and make progress toward overall iteration completion. The operating system cost for the sleep-wake cycle is not significant when the loop execution time is on the order of a half of a second.

In Figure 7 the performance is shown for very large threads (grainsize=4M) and the working set size is 96M. When four copies are run, the system runs out of real memory and begins to page. Paging then dominates the performance of all the cases.

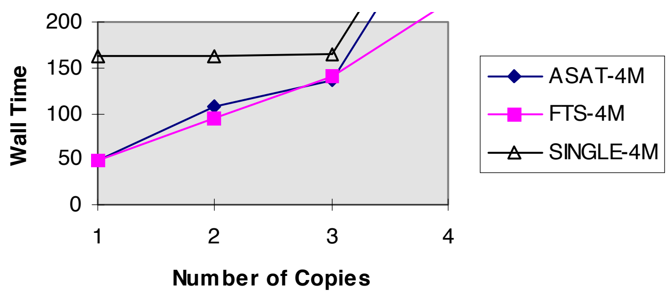

Figure 7 - Multiple Copies Using Very Large Threads (GS=4M)

Interestingly, when four copies are being run, the worst performance is the SINGLE followed by ASAT, with the best performance shown by FTS. It is the one case where the FTS approach performs better than ASAT. When the system working set is too large to fit into memory, a compute bound job becomes an I/O bound job. The excess threads are effectively reducing the apparent memory latency by causing more simultaneous page faults (waiting for disk) to be served in parallel. Furthermore, the assumptions of the timed-barrier thread imbalance test for ASAT are no longer valid and the effectiveness of the test is decreased.

7 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. The impact of context switches, unbalanced load, and cache misses was also studied.

ASAT is proposed as a technique which is easily implementable in a run-time library which effectively balances thread use across an entire system without requiring any central information. In situations where the grain size is small or the working set fits in cache, ASAT can improve performance dramatically. In situations where the working set is larger than any cache, ASAT has no negative impact. There is really no reason not to add ASAT to most loops which can be executed in parallel. The only instance where the traditional fixed thread scheduling out-performed ASAT was when the jobs would not fit into memory and were continuously paging to and from disk.

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

Bibliography

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

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

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

[Guide] Kuck & Associates Inc., "Guide(tm) Reference Manual, Version 2.0",Document #9603002, March 1996.

[LiuSal92] J. Liu, V. Saletore, T. Lewis, "Scheduling Parallel Loops with Variable Length Iteration Execution Times of Parallel Computers," *Proc. of ISMM 5th Int. Conf. on Parallel and Dist. Systems*, 1992.

[LiuSal93] J. Liu, V. Saletore, "Self Scheduling on Distributed-Memory Machines," *IEEE Supercomputing'93*, pp. 814-823, 1993.

[MoBo90] 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

[PolKuck87] C. Polychronopoulos, D. J. Kuck, "Guided Self Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers," *IEEE Transactions on Computers*, Dec. 1987.

[SevEn95] 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.

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

[TukGup89] A. Tucker and A. Gupta , "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors," { ACM SOSP Conf.}, 1989, p. 159 - 166.

[TzenNi91] T. Tzen and L. Ni, "Dynamic Loop Scheduling on Shared-Memory Multiprocessors," { Int. Conf. on Parallel Processing}, 1991, pp 247-250.

The Authors

Charles Severance is the Director of Engineering Computing Services and an instructor in the Computer Science Department at Michigan State University. He is the Vice Chair for IEEE POSIX and edits the monthly Standards Column for IEEE Computer. His research interests include parallel processing and scientific computing. He received a BS in CPS in 1985 from Michigan State University and a MS in CPS in 1990 from Michigan State University. http://www.msu.edu/user/crs/

Richard Enbody is an Associate Professor of Computer Science at Michigan State University. His main research interest is in parallel processing, especially the application of parallel processing to computational science problems. He received his BS in Mathematics in 1976 from Carleton College and his PhD in 1987 from the University of Minnesota. http://www.cps.msu.edu/~enbody/

Paul Petersen received his PhD from the University of Illinois in 1993 specializing in the field of compilers for high performance parallel machines. He is currently employed by Kuck & Associates, Inc. as a Lead Developer working on parallel language products and debugging tools.