Dynamic Processor Allocation with the Solaris Operating System

Kelvin K. Yue
Solaris OS Group
Sun Microsystems Inc.
Palo Alto, CA 94303
kyue@eng.sun.com

David J. Lilja
Dept. of Electrical and Computer Engineering
University of Minnesota
Minneapolis, MN 55455
lilja@ece.umn.edu

Abstract

The Loop-Level Process Control (LLPC) policy [9] dynamically adjusts the number of threads an application is allowed to execute based on the application's available parallelism and the overall system load. This study demonstrates the feasibility of incorporating the LLPC strategy into an existing commercial operating system and parallelizing compiler and provides further evidence of the performance improvement that is possible using this dynamic allocation strategy. In this implementation, applications are automatically parallelized and enhanced with the appropriate LLPC hooks so that each application interacts with the modified version of the Solaris operating system. The parallelism of the applications is then dynamically adjusted automatically when they are executed in a multiprogrammed environment so that all applications obtain a fair share of the total processing resources.

1. Introduction

Fairly allocating processors to the threads of parallel application programs in a multiprogrammed shared-memory multiprocessor is necessary to minimize the execution time of each simultaneously executing application while ensuring high overall system throughput. Traditional time-sharing, such as that implemented in most Unix-based operating systems, typically does not work well to share the processing resources due to high context switching overhead, poor cache utilization, inefficient locking and synchronization, and other related problems [4, 6].

Coscheduling [4] or gang scheduling can solve some of these problems by allocating processors to threads as a group rather than individually. It has the advantage that it can be implemented on top of the time-sharing used in existing operating systems [2]. Statically dividing the system into multiple partitions is another simple approach for allocating processors to parallel applications. While these approaches can improve the execution time of an individual application, they also tend to reduce the system utilization. Dynamically partitioning the system based on the system load is one of the new approaches that can improve the parallel applications' performance in a multiprogrammed environment while maintaining high system utilization [1, 3, 5, 6, 9].

This paper extends our previous work on dynamic processor allocation [9] by implementing and analyzing a strategy called *Loop-Level Process Control* (LLPC) with Sun Microsystems' Solaris operating system and related parallelizing compiler. This paper addresses the feasibility of incorporating a dynamic allocation strategy into an existing commercial operating system and compiler and discusses the related design tradeoffs and novel techniques used in the implementation. Finally, the performance of this strategy is studied using applications from the SPEC95 benchmark suite.

2 LLPC Implementation in Solaris

The fundamental idea underlying LLPC is that, by controlling the number of threads an application is allowed to create based on the system load and the application's available parallelism, the application will be able to utilize as many processors as possible without overloading the system. When the system load is high, LLPC reduces the context switching rate by allowing the applications to create only a small number of threads instead of the maximum number of threads that they may like to create. As a result, execution can still proceed for all applications while no single application monopolizes all of the processors. When the system load is light, however, a highly parallel application is allowed to utilize all of the idle processors.

2.1 Kernel Support

The original LLPC implementation on the SGI Challenge system was done entirely at the user level [9]. LLPC-enabled applications communicated their load information with each other through the use of shared memory. However, this requires all applications running in the system to be LLPC-enabled to maintain accurate system load information. One possible way to obtain accurate overall system load information without requiring all applications to be LLPC-enabled is to have a daemon process periodically check system load and update an appropriate data structure. However, this information is too coarse for LLPC. A low overhead mechanism is needed to obtain finer-grained system load information from the kernel.

Fortunately, the latest version of Solaris (version 2.6) has a new feature called *scheduling control* [7]. This feature provides an efficient mechanism for the kernel and the user-level applications to share scheduling information such as the execution state of a thread and the identification number of the last CPU on which it ran. Threads also can provide some extra information to assist the kernel in making scheduling decisions.

Scheduling control uses physical pages as the communication medium to achieve low overhead. When there is a change in the status of a thread, the kernel puts the updated information into this page with a simple store instruction. Similarly, at the user level, the thread obtains this information simply by reading the page.

While the current implementation of scheduling control provides only per-thread information, we extended it to include overall system information. When a thread makes a call to the scheduling control routine in our prototype, it obtains not only the scheduling information about itself, but also the total number of threads that are currently running on the CPUs and the total number of threads that are waiting on the run queues. This system load information is maintained with clock tick granularity, although it is updated to the page only when the scheduling control routine is invoked.

2.2 Compiler Support

In our previous work, LLPC calls were manually inserted into the applications [9]. To eliminate the manual work and to reduce their execution time overhead, LLPC routines have been incorporated into the run-time library of Sun's Fortran-77 and C parallelizing compilers.

This compiler automatically parallelizes Fortran-style DO loops. It uses many well-known techniques to determine which loops of a sequential application can be effectively parallelized and transforms them into tasks that can be executed concurrently by multiple processors. The execution of these tasks is facilitated by the *microtasking* run-time library. This library manages the threads and supports parallel loop scheduling algorithms such as guided self-scheduling and factoring. The default scheduling strategy is static scheduling, in which the loop iterations are evenly distributed to the threads.

When the application begins executing, the main thread, which is also called the *master thread*, creates a number of *slave threads*. The total number of threads (master plus slaves) is set to the number of physical processors, unless the user specifies a smaller value. When the master thread is executing a sequential section of the application, the slave threads wait idly. When a parallel section is reached, the master thread sets up the necessary data structures and releases all of the slave threads. Each thread then calculates its share of work and calls the subroutine that encapsulates the parallel loop body. When a thread finishes its share of iterations, it waits for all of the other threads to complete. At this point, the master thread continues with the execution of the subsequent sequential section while the slave threads return to the idle state.

Our prototype integrates the LLPC algorithm into the microtasking library to manage the slave threads using a call to the scheduling control routine at the beginning of the master thread routine. This call allows the master thread to obtain the latest system load information whenever a parallel loop is initiated. The master thread uses this system load information to determine how many threads should be used to execute that particular parallel loop, using the algorithm described in the next section.

2.3 Loop-Level Process Control Algorithm

The original microtasking library from Sun allows slave threads to be in one of two states, either *running* or *spinning*. In the *running* state, a thread is executing a chunk of work from a parallel loop and is thus using the CPU for useful work. When it is in the *spinning* state, it is executing an idle loop waiting for work. The master thread is always in the *running* state either executing the application or managing its slave threads.

The LLPC algorithm incorporated into the microtasking library adds a third possible state for a slave thread–*sleeping*. In this state, the slave thread consumes no CPU resources until it is awakened by the master thread. This additional state provides a mechanism for LLPC-enabled applications to reduce the CPU resources they consume to thereby adjust the load they place on the system. Note that slave threads always start in the *spinning* state when the microtasking library creates them at the beginning of the application's execution.

The reason for adding a sleeping state but not putting the slave threads into this state whenever they are not running is because of the overhead incurred when awakening them. When a parallel loop is encountered and the slave threads are spinning, the slave threads can start executing work from the parallel loop instantly. If all of the slave threads are in the sleeping state, however, they need to be awakened before the parallel execution can begin, which can add a significant amount of time to the execution of the parallel loop.

When an LLPC-enabled application begins executing a parallel loop, its master thread obtains the current load information (`current_load`), which is defined to be the number of simultaneously running threads, through the scheduling control system call. It compares this latest information with the previous system load value. If the load has not changed, the parallel loop will be executed using the same number of threads as the last parallel loop that it executed.

If the current load information obtained by the master thread is different than the previous value, the master calculates a new adjustment (`NumThreadsAdj`). First, it determines if the number of running threads in the system is higher than the number of CPUs. If it is, the system is overloaded and the master thread calculates how many of its currently spinning slave threads (`NumThreadsSpin`) should be put into the *sleeping* state to reduce the total number of active threads in the system to the number of CPUs.

Finally, if the number of running threads is lower than the number of CPUs, the CPU resources are not fully utilized. In this case, the master thread determines how many of its *sleeping* slave threads it can awaken.

To further enhance the performance of LLPC, we developed two new techniques for load adjustment. After the master thread determines the adjustment required, it must either put some slave threads to sleep or awaken some sleeping threads. Instead of applying the adjustment immediately, however, it is delayed. When the system is overloaded and spinning threads must be put to sleep, the master does not suspend the threads immediately. Instead, since the spinning slave threads are already ready to run, they are allowed to participate in the execution of the current parallel loop. A flag is set to notify the selected slave threads to go to sleep *after* completing their share of the parallel work.

On the other hand, when the system is underutilized, the master thread does not wait for sleeping threads to awaken before starting the parallel loop execution. Rather, it sends a signal to awaken the sleeping threads and then starts the parallel loop execution with the threads that are already in the spinning state. This approach masks the thread wake-up time with the execution of useful work. The newly awakened threads then go into the spinning state to participate in the next parallel loop execution.

2.4 Example Execution Trace

To provide a better idea of how the various LLPC pieces work together, an example is shown in Figure 1. This diagram shows the execution of an application on a 4-CPU system using the LLPC strategy. When the application begins its execution, a master and three slave threads are created and the slave threads are put into the spinning state. At the beginning of the first parallel loop, the master thread determines that it is the only application using the system. Consequently, it uses all of the slave threads to execute the parallel loop. After the parallel execution, the slave threads return to the *spinning* state.

When the second parallel loop is reached, the master thread finds that the system load has increased due to other applications sharing the CPUs. The system load is at five, which is one more than the number of CPUs, so it decides to put one of its slave threads to sleep. The second parallel loop is still executed by four threads (one master and three slaves) but after the loop is executed, one of the slave threads goes into the *sleeping* state. At the third parallel loop, the system load remains unchanged, so only the slave threads that are in the *spinning* state are used for this loop execution.

In the meantime, the other application terminates. When the master thread checks the load again at the beginning of the fourth loop, it decides that it can awaken one of its sleeping threads. After sending a signal to the sleeping thread, the master and the two spinning threads execute the fourth parallel loop. The sleeping thread eventually wakes up and goes into the *spinning* state. It can then participate in the execution of the next parallel loop.

3 Performance Evaluation

The system we used for the performance evaluation is a Sun Ultra Enterprise 6000 Server. This single-bus

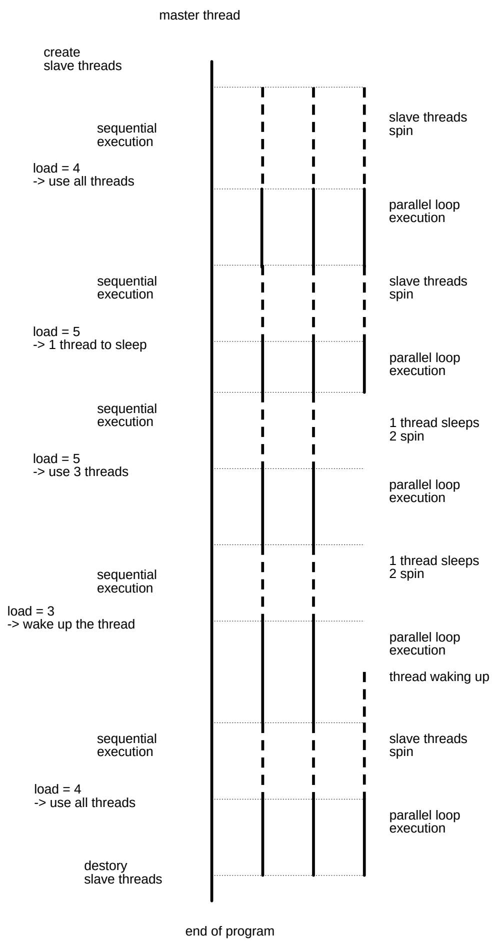

Figure 1. An example showing how LLPC changes the number of threads it uses to execute the parallel loops of an application program.

shared-memory architecture system was equipped with four UltraSPARC processors running at a clock rate of 167 MHz. Each processing unit had 16 Kbytes of on-chip data cache and 16 Kbytes of on-chip instruction cache. The secondary cache was 512 Kbytes of unified instruction and data cache. The system had a total of 256 Mbytes of physical memory.

The benchmarks we used for the performance comparison are four applications from the SPEC95 floating-point suite that have similar parallel run-times. All of the benchmark programs were parallelized using Sun's Fortran compiler and linked with our modified microtasking library.

Before studying the performance of our implementation, we determine the overhead of adding the LLPC calls to the benchmark programs. As shown in Table 1, the run-times of the LLPC-enabled benchmarks are slightly higher than the unmodified programs since, when LLPC is enabled, the applications become sensitive to the unavoidable background load caused by the various system daemon processes. However, the overhead effect is quite small. The benchmarks were able to use all of the CPUs for executing their parallel loops most of the time, as shown by the average number of threads used per loop being almost equal to the number of physical CPUs.

Table 1. The overhead due to adding the LLPC calls to the test programs.

App.	orig. run-time (sec)	LLPC-enabled run-time (sec)	aver. threads used/loop
swim	172	174.67	3.999
su2cor	166	166.33	3.999
hydro2d	241	241.0	3.996
mgrid	247	249.0	3.999

3.1 LLPC Compared to Time-Sharing

In this experiment, a synthetic sequential application was used that repeatedly executed for 18 seconds then slept for 5 seconds. This benchmark was executed simultaneously with the individual SPEC benchmarks. A corresponding *slowdown factor* for the parallel application was then calculated as follows:

$$\text{slowdown} = \frac{\text{multiprogramming run-time}}{\text{parallel run-time on a dedicated system}}.$$

A slowdown of one means that the execution time of the parallel application was not affected by the sequential application while a slowdown factor of two would mean that the parallel application executed for twice as long as it did on a dedicated system.

As shown by the slowdown factors in Table 2, the performance of the LLPC-enabled applications are not affected by the sequential load as much as those executed with time-sharing. With LLPC, the execution time of the parallel applications increases from 15% to 25% while with time-sharing, the execution time increases by at least 46%. For `su2cor`, the run-time with time-sharing is more than twice its stand-alone run-time. This table also shows that the average number of threads used per parallel loop for each benchmark with LLPC has decreased from around four threads per loop in stand-alone execution to about 3.25 threads per loop. This change clearly illustrates that LLPC adjusted its thread usage to compensate for the varying sequential background load.

Table 2. The slowdown factors of the benchmarks when using time-sharing and LLPC to execute one parallel application and one sequential application.

App.	time-sharing	LLPC	threads/loop
swim	1.46	1.20	3.26
su2cor	2.39	1.15	3.27
hydro2d	1.97	1.17	3.25
mgrid	1.73	1.25	3.27

3.2 LLPC Compared to Static Partitioning

To evaluate the effectiveness of LLPC when there are multiple parallel applications sharing the system, we compare LLPC to the Solaris version of static partitioning called *processor set (*`psrset`*)*. The processor set feature allows the system administrator to partition the processors in a system into several subsets. These subsets then run only those applications that are specifically assigned to them, although multiple applications can share a subset of processors using time-sharing.

For the processor set measurements, we partitioned the system into two sets with two CPUs in each. One of the SPEC application programs then was executed in each set. For the LLPC measurements, the SPEC applications were compiled with the modified micro-tasking library. Pairs of applications were then executed simultaneously on all four processors while they adjusted their processor requirements using the LLPC algorithm described above. Because of the dynamic nature of LLPC, a slight change in the system load can affect the number of processors used to execute a parallel loop, which then produces slightly different total execution times. Therefore, we conducted the LLPC experiment five times for each set of benchmarks and present all five execution times for comparison. We did not compare the performance with time-sharing in these experiments since time-sharing proved to be significantly slower than both LLPC and processor set.

The slowdown factors shown in Figure 2 suggest that, in almost all cases, LLPC reduces the slowdown of the parallel applications compared to the static partitioning of the processor set approach. The reductions in the slowdown factor for LLPC range from a few percent to almost 20% when `su2cor` is run with `mgrid`. Only one benchmark, `swim`, troubles the LLPC approach. This benchmark, as defined by SPEC, consists of a single application that is run twice in succession with two different input data sets. As a result, the start-up costs for LLPC are encountered twice which contributes to the higher overhead of LLPC on

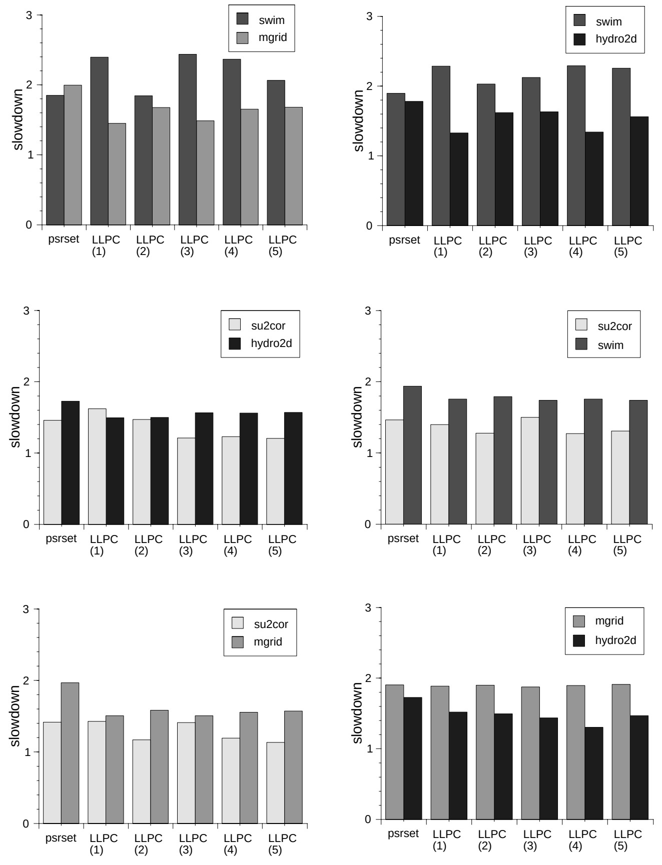

Figure 2. Comparing the slowdown factors of LLPC to the Solaris *processor set (psrset)* static partitioning mechanism when executing two parallel applications simultaneously. Five different runs of the LLPC-enabled applications are shown to demonstrate its potential variability.

this application, as shown in Table 1. Moreover, after the first run finishes and before the second run starts, the other program executing in the system is able to grab all of the processors, leaving the second run of the `swim` benchmark with only a single thread to use. This thereby puts the second run of `swim` at an initial disadvantage.

One of the main advantages of LLPC, however, is that it allows a much more flexible allocation of the processors. Instead of limiting the number of threads used by an application to 2 as in the statically partitioned case, LLPC adjusts the allocation based on the availability of the processors. For instance, when `su2cor` was running concurrently with `hydro2d`, Table 3 shows that `su2cor` used 2.30 threads per parallel loop while `hydro2d` used 2.97. This table shows that all of the applications were able to use an average of more than 2 processors to execute their parallel loops with LLPC. The static partitioning, however, sets a hard limit of at most two processors for the execution of each application's parallel loops. Thus, this flexibility in being able to use otherwise idle processor resources allows LLPC to outperform the statically partitioned processor set strategy.

Table 3. The average number of threads executed per parallel loop when using LLPC to execute two parallel applications simultaneously.

	swim	su2cor	hydro2d	mgrid
swim		2.91	2.47	2.20
su2cor	2.68		2.30	2.52
hydro2d	2.97	2.97		2.66
mgrid	2.86	3.28	2.81	

4 Conclusion

Fairly allocating the processors of a multiprogrammed shared-memory multiprocessor system is necessary to minimize the execution time of individual parallel applications while still maintaining high overall system utilization. Previous research has shown that, for loop-level parallelized applications, techniques that dynamically adjust the parallelism of the applications based on the system load can effectively achieve these contradictory goals [3, 5, 9]. This paper has demonstrated how to incorporate the LLPC dynamic processor allocation strategy [9] into the Solaris production operating system and related parallelizing compiler.

A unique feature of this implementation is the addition of a *sleeping* state for slave threads that allows parallel applications to dynamically adjust how much load they place on the system. Another unique feature is the masking of a sleeping thread's restart time with the execution of parallel loop iterations. Experiments with the SPEC95 benchmark suite show that LLPC allows simultaneously executing parallel applications to exploit more parallelism on average in each parallel loop than static partitioning or time-sharing. As a result, parallel applications executed with LLPC have shorter execution times on multiprogrammed systems that those executed using static partitioning or time-sharing.

5 Acknowledgements

We are indebted to Andrew Tucker for implementing the scheduling control feature that is currently being shipped with Solaris, including the processor set feature. We also would like to thank Michael Sebrée and Dan Stein for their help with various kernel and thread issues. Special thanks go to Vinod Grover for his assistance with the compiler.

A full version of this paper can be found in [8].

References

[1] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler activations: Effective kernel support for user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53 – 79, feb. 1992.

[2] J. Barton and N. Bitar. A scalable multi-discipline, multiple-processor scheduling framework for IRIX. In *Workshop on Job Scheduling Strategies for Parallel Processing, International Parallel Processing Symposium*, pages 24 – 40, April 1995.

[3] R. Konuru, J. Moreira, and V. Naik. Application-assisted dynamic scheduling on large-scale multicomputer systems. In *Second International Euro-Par Conference (Euro-Par'96)*, pages II:621–630, 1996.

[4] J. Ousterhout. Scheduling techniques for concurrent systems. In *Distributed Computing Systems Conference*, pages 22–30, 1982.

[5] C. Severance, R. Enbody, S. Wallach, and B. Funkhouser. Automatic self-allocating threads on the Convex Exemplar. In *International Conference on Parallel Processing*, pages I:24 – 31, 1995.

[6] A. Tucker. *Efficient Scheduling on Multiprogrammed Shared-memory Multiprocessors.* PhD thesis, Department of Computer Science, Stanford University, 1993.

[7] A. Tucker. *Scheduler Activations in Solaris.* Internal document, Sun Microsystems Inc., April 1996.

[8] K. Yue and D. Lilja. Dynamic processor allocation with the Solaris operating system. Technical Report HPPC-97-09, University of Minnesota, Minneapolis, Minnesota, September 1997. Available from http://www-mount.ee.umn.edu/~lilja/papers.sched.html.

[9] K. Yue and D. Lilja. An efficient processor allocation strategy for multiprogrammed shared-memory multiprocessors. *Transactions on Parallel and Distributed Systems*, 8(12):1246 – 1258, Dec. 1997.