Loop-Level Process Control: An Effective Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors

Technical Report No: HPPC-95-02 January 1995

Kelvin K. Yue
David J. Lilja

UNIVERSITY OF MINNESOTA
High-Performance Parallel Computing Research Group
Department of Electrical Engineering • Department of Computer Science • Minneapolis • Minnesota • 55455 • USA

Workshop on Job Scheduling Strategies for Parallel Processing, IPPS, April, 1995.

Loop-Level Process Control: An Effective Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors

Kelvin K. Yue[†] and David J. Lilja[‡]
yue@cs.umn.edu lilja@ee.umn.edu
[†]Department of Computer Science, [‡]Department of Electrical Engineering
University of Minnesota, Minneapolis, MN 55455

Abstract *Processor time-sharing is the most common way to increase the overall system utilization for shared-memory multiprocessor systems. However, the performance of individual applications might be sacrificed due to the high overhead of context switching, due to the processing power wasted by busy-waiting synchronization and locking operations, and due to poor cache memory utilization. In this paper, we propose a simple and effective processor allocation scheme, called* Loop-Level Process Control (LLPC)*, for multiprogrammed multiprocessors. At the beginning of each parallel section of each application program, LLPC uses the current system load to determine an upper limit on the number of processes the application can create for that parallel section. Preliminary simulation results using the Perfect Club Fortran benchmarks show that this loop-level process control scheme can produce a high system utilization while maintaining high performance for the individual applications. Another advantage of this strategy is that it is transparent to the programmer and does not require any modifications to the operating system. Consequently, the application can remain portable and compatible.*

1 Introduction

Many of today's multiprocessor machines are *multiprogrammed* in which several applications execute simultaneously while sharing the processors and other system resources. These multiprogrammed multiprocessor systems usually use a *time-sharing* policy to multiplex the processors among processes from several different applications. Often implemented as an extension of the traditional Unix operating system, this time-sharing processor allocation policy provides easy portability and compatibility for the applications from their uniprocessor counterparts. Examples of this type of multiprocessor system include the Alliant FX [1], Silicon Graphics Onyx [15], and Cray C90 [4]. However, multiplexing the processors can severely degrade the performance of the individual applications. For instance, it has been reported that 10 to 23% of the processing power in an Alliant FX/8 system is spent on multiprogrammed overhead [5].

This work was supported in part by the National Science Foundation under grant no. MIP-9221900.

Several solutions to the multiprogrammed multiprocessor scheduling problem have been proposed, such as *space sharing* or *space partitioning* the processors, *gang scheduling* the related tasks [13], dynamic process control [11, 16], non-blocking synchronization [8], and policies that prevent tasks that hold a lock from being swapped out [18]. However, these solutions either need to sacrifice the overall system utilization for the improvement of an individual application's performance, or they require modifications to the existing operating system combined with a special programming model.

This paper proposes a new policy for processor allocation that maintains a high system utilization with only a small degradation in all individual applications' performance. Our strategy is designed with the following goals in mind:

- It is transparent to the programmer and requires no special programming language or parallelization model, instead working with the traditional *fork-join* parallel programming model.
- It is implemented at the user level to maintain high portability and compatibility with no modifications to the operating system.
- The possibility of a scheduling bottleneck is minimized by using a distributed scheme with no centralized agent or server.
- It is fair to all applications in the system.

- It is simple and straightforward, so that it can be easily implemented and optimized.

The remainder of the paper is organized as follows: Section 2 provides further background information on the multiprogrammed multiprocessor scheduling problem and describes the existing solutions. Our programming model and system architecture are also described. Section 3 presents our proposed processor allocation scheme, while Section 4 presents our simulation methodology. Preliminary results are discussed in Section 5. The final section concludes the paper and discusses future work.

2 Background

This section discusses the targeted system architecture and the programming model. It also provides additional details on the performance issues in multiprogrammed multiprocessor systems and reviews existing strategies. Although this discussion can be applied to other types of architectures, our focus is mainly on shared-memory multiprocessor systems.

To clarify the terminology, an *application* is an executing user program, either sequential or parallel. It is also known as a *job*. During its execution, an application (or a job) creates parallel *sub-tasks*, simply called *tasks*, which are then assigned to the processors for execution. We use the terms *task*, *process*, and *thread* interchangeably.

2.1 System Architecture and Programming Model

Because the memory of a shared-memory multiprocessor is equally accessible by all processors, these systems have a programming model that is similar to traditional uniprocessor systems. Consequently, shared-memory multiprocessors have been widely commercialized and are often used as a high-performance general purpose machine. Figure 1 shows a diagram of a shared-memory multiprocessor system in which a number of processors are connected together with the main memory through a high-speed network. In this system, all processors run at the same speed and each of the processors executes its instructions independently of the others. Some shared-memory systems also have a local cache memory in the processor module to improve the average memory access time.

One of the common programming models for shared-memory multiprocessor systems is the *fork-join* model. A programmer or compiler parallelizes an application with this model by recognizing the independent sections of the application and partitioning them into parallel tasks. For example, each iteration of a parallel loop can be partitioned into an independent task and executed concurrently with other iterations [10, 17].

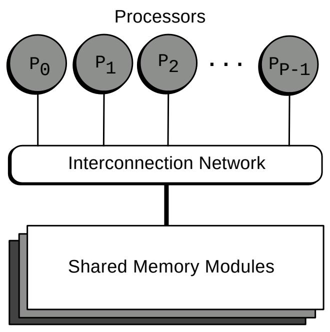

Figure 1: Shared memory multiprocessor architecture.

Figure 2 shows the task graph of a typical *fork-join* parallel application. At the start of a parallel section of the program, sub-processes will be spawned out, similar to creating child processes using the *fork* command in the Unix operating system. Since the sub-processes are independent, they can be executed concurrently. At the end of a parallel section, all of the sub-processes will be synchronized. Some systems or programming language implementations destroy the sub-processes after the synchronization completes, similar to the Unix *join* command. New processes are then created when the next parallel section begins executing. An alternative strategy is to put the processes to sleep until they are needed for the next parallel section. Depending on the characteris-

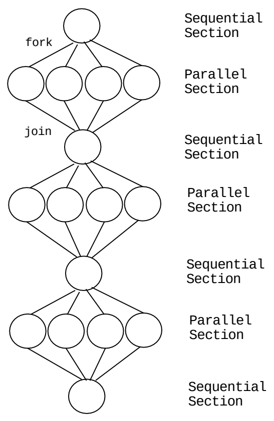

Figure 2: Task graph of a typical *fork-join* program.

tics of an application, the number of processes required for each parallel section might be different. Lock and unlock operations are used to coordinate accesses to shared variables, and to maintain the correctness of the program execution.

To share the processors efficiently, some scheduling strategy is needed to allocate the tasks to the processors for execution. The most commonly used strategy is the traditional single work queue time-sharing approach, which is also used in the Unix operating system for uniprocessor systems. With this strategy, tasks that are ready for execution are put into the *ready queue*. Whenever a processor is idle, it removes the first task from the ready queue and begins executing it. If the task is not completed before the *time slice* expires, the task is returned to the end of the ready queue. The processor then begins executing the next task from the beginning of the queue.

2.2 Performance Issues

The time-sharing processor allocation policy has been used in uniprocessor systems for decades and it has proven to be a fair and efficient system for sharing the single processor among several applications. Although this policy can improve the system utilization in multiprogrammed multiprocessor systems, it can significantly degrade the performance of individual applications.

One problem with the time-sharing policy occurs when the total number of application program processes is greater than the total number of physical processors. Since the processors must multiplex among several processes, the context switching overhead can be very high, which will degrade the performance of all of the applications. Another problem is that data cache locality cannot be efficiently utilized since, at the end of the time slice, the current task will be switched out and a new task will be switched in. The new task must reload its working set into the cache, limiting the performance advantage of the cache. The problem of high context switching overhead and poor cache utilization also occurs in time-sharing uniprocessor systems.

Multiplexing the processors among several processes not only creates context switching overhead and degrades the cache utilization, but it may also waste processing power due to busy-waiting. When a parallel application executes on a shared-memory multiprocessor system, it requires a lock operation to obtain exclusive access to shared variables. While a process holds a lock, other processes which need to access the same data must wait. In a multiprogrammed environment, processes will be swapped out when their time quantum expires. If a process that holds a lock is context-switched out, other processes that are waiting for the lock will not be able to proceed until that process again has its turn for another processor allocation. Those processes that are waiting for the lock will be idle busy-waiting, thereby wasting the system's processing power [18]. A similar scenario called *process thrashing* may also occur because of the active processes being context-switched [13].

There are several existing solutions that can be used to prevent the above situations from occurring. The simplest solution to prevent active processes from idly waiting for some inactive processes is to prevent the scheduler from deallocating a process that holds an active lock [9]. Also, processes that are waiting for a lock can be put into a *sleep* state and their unused time slice can be allocated to other processes to perform some useful work [8, 12]. Similar techniques for synchronization operations have also been proposed to eliminate the problem of process thrashing. However, these solutions have no impact on the context switching overhead, and they might be unfair to the waiting processes.

To decrease the context switching overhead, *space sharing*, also called *space partitioning*, can be used instead of time-sharing. This policy evenly divides the number of processors among the applications and limits the total number of processes that the applications can create to be less than or equal to the total number of physical processors. As a result, context switching is eliminated. However, the system might not be fully utilized since the number of processors needed by an application varies over the course of its execution. Limiting the number of processes an application can create will limit the parallelism that can be exploited by the application even though there may be idle processors available in other partitions.

Coscheduling has been proposed to eliminate the problem of process thrashing by allocating processors to all of the related tasks of an application all at the same time [13]. If a parallel application has a set of processes that are closely interacting during execution, they should be scheduled for execution simultaneously. When an application is ready for execution, its processes are appended to the system work queue. Scheduling is performed by moving a window, whose length is equal to the number of processors in the system, over the queue. Tasks or processes that are in the queue and within the window are removed from the queue and a processor is allocated to each task. If a task is not runnable, because it is related to tasks that are still in the queue, for instance, it will not be scheduled and the available processor will be allocated to the next runnable task. At the end of the time quantum, all the allocated tasks will be put at the end of the queue, and the processors will be scheduled to run the next window of runnable tasks. A similar strategy called *gang scheduling* has been implemented on some existing operating systems, such as the Silicon Graphics IRIX operating system [2].

Coscheduling avoids process thrashing since all the processes of a job are scheduled at the same time. Processor fragmentation can occur, however, when an application does not require all of the processors in the system, and it does not leave enough processors for another application's processes [8]. It has been shown that this processor fragmentation might cause as much as a 25% loss in the computing resources [6]. Although gang scheduling can improve the performance for fine-grained synchronization [7], it might not reduce the number of context switches, and thus can still cause poor cache performance [8].

The goal of *dynamic partitioning* is to minimize context switching by controlling the total number of processes in the system, which in turn preserves the processor caches [11, 16]. Each job in the system is allocated an equal fraction of the processors unless a job has a smaller amount of available parallelism. For instance, if there are three applications competing for 20 processors in a system, and the applications request 4, 10, and 20 processors each, the first job will be allocated 4 processors while the other two will be allocated 8 processors each. If another application requiring 4 processors arrives, 4 processes from the executing applications are suspended and their processors are released and reassigned to the new application. When an application terminates, its processors will be allocated to the other applications' suspended processes.

Although dynamically adjusting the processor allocation can improve system utilization and can reduce the context switching rate, the scheduler and the applications require precise coordination. In addition, a special programming model may be needed to accommodate this coordination. Moreover, since processor reassignment requires suspension of some executing processes, some critical processes may be suspended, which could affect other dependent processes. Finally, dynamically suspending and reallocating processors can increase the system overhead, which can degrade the performance of the applications. A similar policy called *adaptive partitioning* has also been proposed [11].

3 Loop Level Process Control

Dynamic partitioning [11], process control [16], and adaptive partitioning [11] have been shown to be effective methods for minimizing multiprogramming overhead and thereby improving the performance of the individual application programs. However, these methods require modifications to the programming model and to the operating system. Thus, they might not be readily adopted in existing systems. In this section, we present a new allocation policy called *Loop-Level Process Control* (LLPC) which is an extension of previous process control policies that works with existing operating systems and the traditional *fork-join* programming model.

With a parallelizing compiler, the loop-level parallelism of many existing application programs can be exploited without rewriting the sequential code [10, 17]. Although this approach might not be able to exploit the maximum inherent parallelism in an application, it can increase the portability of the code and can lessen the programmer's burden in developing new parallel algorithms. Our loop-level process control strategy targets this type of programming model to help improve the performance of these applications in a time-shared, multiprogrammed multiprocessor environment.

The philosophy behind the loop-level process control strategy is to utilize as many processors as possible without overloading the system. When the system load is high, LLPC tries to minimize the context switching rate by allowing the applications to create only a small number of processes instead of the maximum number of processes 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.

The loop-level process control strategy is outlined in the following steps:

1. At the beginning of the execution of a parallel section, check the system load.
2. Based on the system load, determine the number of processors, `no_processes`, available for the execution of this particular parallel section.
3. Create `no_processes` processes and append them to the system work queue.
4. When a processor is allocated to a process, the process executes its share of the parallel work.
5. When all parallel tasks have completed, synchronize the processes and release them to the system.

```
no_processes = max[1, min(no_needed,
       total_physical_P - system_load)];
```

Figure 3: LLPC heuristic for determining the number of processes an application is allowed to create when executing a parallel section of code, such as a parallel loop.

6. Continue the execution of the next sequential portion of the application using only a single processor.

The above steps are executed for each parallel section. These steps can be directly inserted into the generated object code by the parallelizing compiler when a parallel section is detected in the code.

There are several issues that need to be considered when implementing LLPC. The most important issue is how to determine the current system load with minimum overhead. We define the system load to be the number of busy processors plus the total number of processes waiting in the work queue. Fortunately, this information is already available from the Unix operating system making this information easily accessible to the applications.

The second issue is determining how many processes an application should create to prevent degrading the performance of the system. We use the heuristic shown in Figure 3 where variable `total_physical_P` is the total number of physical processors in the system, `system_load` is the load of the system at that moment, `no_processes` is the number of processes an application is allowed to create for the execution of that particular loop, and `no_needed` is the the number of processes that loop requires to attain maximum parallelism. This heuristic allows the parallel loop to create as many processes as there are available processors. If the system is completely busy, LLPC allows an application to create at least one process. Although this approach introduces some context switching, it still allows the application to make some progress in its execution.

A third important issue is the fairness of the policy. LLPC seems to favor applications that are ready to run first allowing them to create as many processes as they require, while late arrivals might not be allowed to create enough processes to achieve their best performance. Moreover, LLPC does not suspend any process once it has been created which may cause one application to monopolize the entire system. However, the parallelism within an application varies during its execution. So, although it might not be allowed to create as many processes as it desires for this parallel loop, an application program may be allowed to create more processes when it begins executing its next parallel loop since the system load most likely will have changed by then. Therefore, the overall performance of an application is not dependent on its order of arrival.

Unlike other dynamic process control schemes [11, 16], LLPC does not suspend any active processes even when the system load is high. Schemes that dynamically suspend processes usually require a scheduler to check if the target process is *safe* to be suspended with no dependence between the target process and other active processes. Although LLPC might not be able to adapt to changes in the system load instantaneously, processes are automatically suspended at the end of each parallel loop, and the processor allocation is revised at each loop entry. As a result, this strategy will adjust to the system load within a period time. Moreover, not allowing dynamic suspension simplifies the implementation of the policy and does not require any changes to the operating system scheduler, which is an important advantage for existing systems.

The following sections discuss the methodology we used to evaluate the performance of our strategy and how our strategy performs compared to others.

4 Simulation Methodology

Our simulation consists of two modules: a trace generator and a multiprogrammed multiprocessor simulator. The trace generator is used to generate memory reference traces from the benchmark programs while the simulator simulates the performance of the different schemes based on these traces. The memory traces characterize the benchmark programs and, by creating and suspending processes based on these traces, these traces mimic a dynamic workload. This section discusses the methodology in detail.

4.1 Trace Generation

The application programs we used for this study are from the Perfect Club Fortran benchmarks [3]. A modified version of the Parafrase-2 parallelizing compiler [14] is used to parallelize the benchmark programs. In addition to parallelizing the application, the compiler inserts memory trace generation calls into the Fortran source code to annotate each memory access with the type of reference (a read or a write), a unique identifier for each parallel loop, and the iteration index if the access is inside a parallel loop. An estimate of the execution time required for each section of an application is determined by counting the number of memory references within the section, where we assume that each memory access requires one cycle. Although this approximation does not give us the *exact* execution time for each section, it does gives a reasonable ratio of the execution times between different sections and between applications. At the beginning of a parallel section, a marker with the number of iterations of the loop is inserted to notify the simulator to create sub-processes. Also, at the end of a parallel section, a marker is inserted to notify the simulator to synchronize the processes.

Because of the length of the traces, we used only four representative applications from the benchmarks in this preliminary study. ADM and TRACK are used as applications with a small degree of parallelism, while DYFESM and TRFD are used as applications with a high degree of parallelism. These applications are modified to reduce their execution time and the size of the memory traces. We reduce only the number of time steps of the programs so that the memory access behavior remains approximately the same as the original programs. The outer loop is reduced from 720 to 2 for ADM; from 62 to 11 for TRACK; from 40 to 15 for TRFD; and from 1000 to 4 for DYFESM.

Table 1 shows the characteristics of the traces from the benchmark programs. The maximum possible speedup values are obtained by calculating the best possible parallel execution time for the benchmark executing on a dedicated 16-processor system.

Table 1: Characteristics of the benchmark programs.

Benchmark	Parallel loops executed	No. of iterations within a loop: mean	No. of iterations within a loop: std dev.	Max. possible speedup
adm (apsi)	2155	30.3	49.8	1.104
dyfesm (sdsi)	13385	33.4	29.6	2.134
track (mtsi)	4374	16.1	19.2	1.102
trfd (tisi)	83352	8.9	4.7	4.545

4.2 Multiprogrammed Multiprocessor Simulator

This module simulates a 16-processor multiprogrammed shared-memory system at the clock level. A processor can be in either a *busy* state or an *idle* state. The status of the processors is represented by an array structure and, at each cycle, the status of each processor is updated accordingly. Multiple traces are read into the simulator *concurrently* to mimic multiprogramming. The simulator generates tasks based on these traces. A first-come-first-serve work queue is used to for these ready tasks. This approach mimics the implementation of many existing systems.

When a processor is idle, it removes the first process from the system work queue and begins executing. Only one processor can access the work queue at a time. Other idle processors must wait. When a processor finishes its assigned task, it obtains another process from the head of the queue. If the time quantum expires, however, the processor stops executing the current task and returns it to the end of the queue. A time penalty is added to that processor's overhead time to account for the context switching overhead. A new task is then obtained from the beginning of the queue. In this study, we set the time quantum to 100000 cycles. An overhead penalty of 100 cycles is incurred for swapping out a process.

When an active process must create sub-processes to execute the parallel loop iterations, the sub-processes are created sequentially with an overhead penalty of ten cycles per subprocess. Simple chunk scheduling [17] is used to evenly divide the iterations of the loop among the sub-processes by assigning $\lceil N/T \rceil$ iterations to each process, where N is the number of iterations, and T is the number of sub-tasks or processes created for the loop execution. The newly created processes are then appended to the system work queue. At the end of a parallel loop, all sub-processes from that loop must be synchronized. When a process reaches a synchronization point, it checks if other processes that belong to the same application are active and have reached the same point. If so, all processes are released and can continue their execution. Otherwise, the process must busy-wait for the other processes.

For gang scheduling, the simulator keeps track of the number of idle processors and the size of the *gang* for each application. As with a time-sharing policy, all the ready processes are waiting in the system work queue, but the allocation is done as a gang instead of one process at a time. Therefore, a process is assigned to a processor only if there are enough idle processors for the entire gang. This implementation is similar to the one used in the SGI IRIX operating system [2].

For space sharing, each application can create a maximum of $\lceil P/J \rceil$ processes for its execution of a parallel loop, where P is the total number of physical processors and J is the total number of concurrent jobs.

When our loop level process control policy is used, the simulator operation is the same as with time-sharing except that, at the beginning of each parallel loop, an upper limit on the number of processes for the current parallel loop is determined before the processes are created (see Figure 3). The new processes are appended to the system queue where a time-sharing scheme is then used to schedule these processes for execution.

At the end of the simulation, the simulator reports the speedup value for each application and the total system utilization, distinguishing between the time used for the applications, and the system overhead. It also produces a trace of the number of busy processors versus the execution time.

5 Simulation Results

This section presents and discusses the simulation results. First, the performance of the different policies are presented when executing multiple copies of the same application. This allows us to compare the basic behavior of the policies. Then the performance is compared with a combination of different applications.

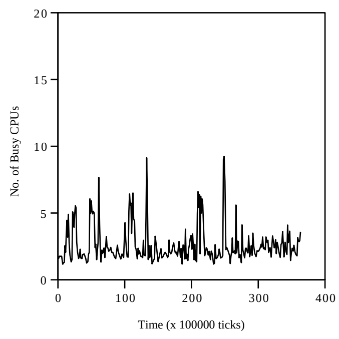

Figure 4: CPU status graph for a single copy of TRACK executing on a dedicated machine.

Figure 4 shows the change in the number of busy processors during the execution of a single copy of the TRACK application on a dedicated machine. Because of the large data set and the limited resolution of the graph, we averaged the number of busy processors within the range of 100000 cycles and treated that as one data point. This figure confirms that the parallelism of an application varies during its execution. This figure also demonstrates that many processors are idle most of the time which makes many processors available for multiprogramming. While we do not show it in this paper, the other application programs show similar behavior.

To compare the performance of the schemes under different system loads, we varied the number of concurrent applications in the system and measured the speedup of the applications and the overall system utilization. The performance of multiple executions of TRACK and TRFD are shown in Figure 5 and Figure 6, respectively. Figure 5 shows that LLPC

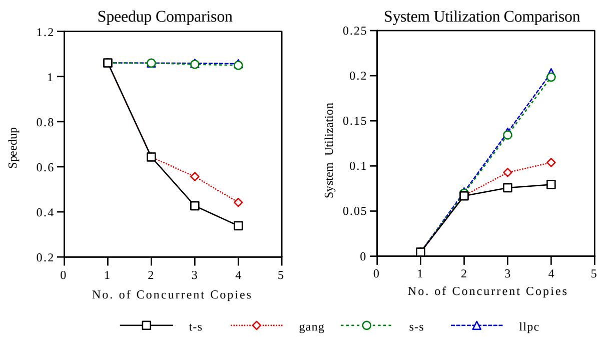

Figure 5: Performance comparison when executing multiple copies of TRACK.

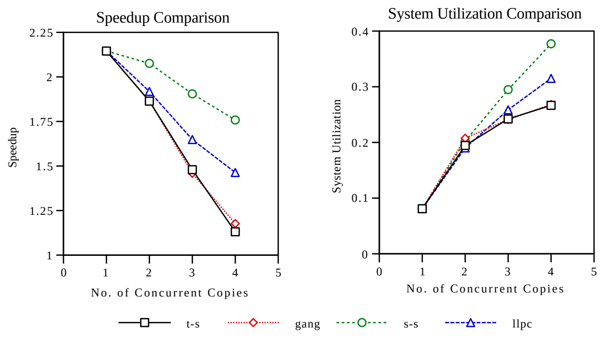

Figure 6: Performance comparison when executing multiple copies of TRFD.

performs as well as the space-sharing policy (*s-s*), in terms of both speedup of the application and the overall system utilization, for an application with a small degree of parallelism. The figure also shows that both space-sharing and LLPC maintain a constant speedup for the application when the number of concurrent applications increases.

On the other hand, the performance of the application under time sharing (*t-s*) and gang scheduling (*gang*) degrades rapidly, with gang scheduling slightly

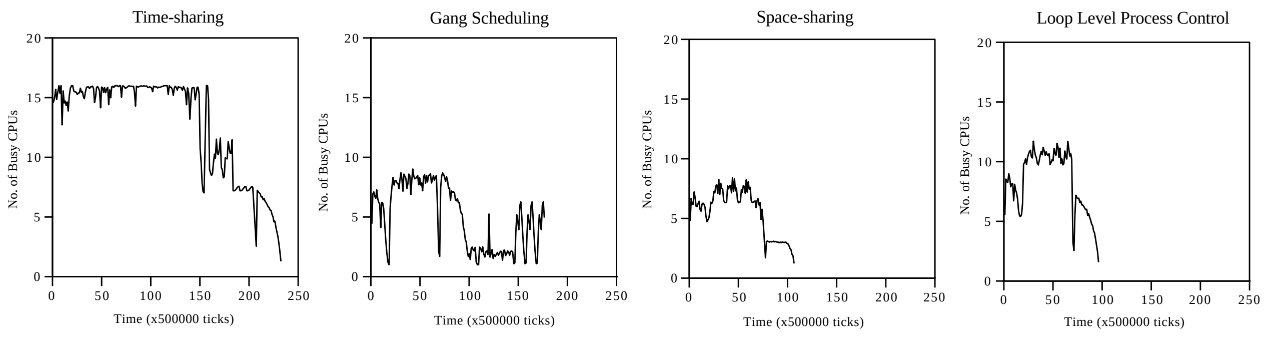

Figure 7: CPU status graphs for multiprogramming using different allocation policies.

outperforming time-sharing. For an application with a high degree of parallelism, space-sharing performs the best, followed by LLPC (Figure 6). Since this application requires more processors on average than TRACK, more context switches are needed to share the system. As a result, the speedup of TRFD under all the schemes decreases as the number of concurrent copies increases.

In Section 3, we claim that LLPC allocates the processors fairly among several applications. To validate this claim, we compare the execution times of four copies of the TRFD benchmark program when they are executing concurrently. Table 2 shows the execution time in number of cycles required by each copy under different scheduling policies with their means and variances. Since space-sharing evenly divides the

Table 2: Fairness comparison of the different polices.

App.	Time S.	Gang Sch.	Space S.	LLPC
1	83531121	80260361	53721381	64543401
2	84036061	80470991	53721381	64783441
3	84626191	81307611	53721381	64836561
4	84920471	82434731	53721381	64892961
mean	84278461	81118423.5	53721381	64864091
var.	536275.58	854970.32	0.0	133170.0

number of processors among the applications, all the applications have the same share of the processing power and, therefore, have the same execution time. Compared to time-sharing and gang scheduling, LLPC achieves a shorter execution time and a smaller variance. Therefore, LLPC not only performs better than time-sharing and gang scheduling, it is also a fair allocation scheme.

In addition to using multiple copies of the same application, we study the performance of the policies in a more practical environment by simultaneously executing applications with different characteristics. We chose four of the benchmark applications, two with small degrees of parallelism (TRACK and ADM) and two with higher degrees of parallelism (DYFESM and TRFD), to evaluate the performance of the different policies.

Figure 7 shows the change in the number of busy processors when all four applications are executed concurrently. Each data point in these graph represents the average number of busy processors within a range of 500000 cycles. Again, time-sharing has the longest execution time and generates a very high system load most of the time. Gang scheduling requires a shorter execution time, but it does not fully utilize the system. Space-sharing produces a much shorter execution time than either time-sharing or gang scheduling since it decreases the rate of context switching. However, LLPC improves the performance even further. It requires a shorter execution time and produces higher system utilization than space-sharing.

Figures 8 and 9 compare the performance in more detail. They also compare the performance to the case when the applications are executed on a dedicated machine (*single*). In terms of the speedup of the applications (Figure 8), both time-sharing (*t-s*) and gang scheduling (*gang*) perform badly, performing worse than if the applications were executed in a uniprocessor system. Gang scheduling performs well only for TRFD, which is the application with the highest degree of parallelism. Both space sharing (*s-s*) and loop level process control (*LLPC*) perform well. Space-sharing performs slightly better for applications with low degrees of parallelism, while LLPC performs better for applications with higher available parallelism. Moreover, LLPC maintains about 67–97% of the performance the applications obtained when executed on a dedicated system.

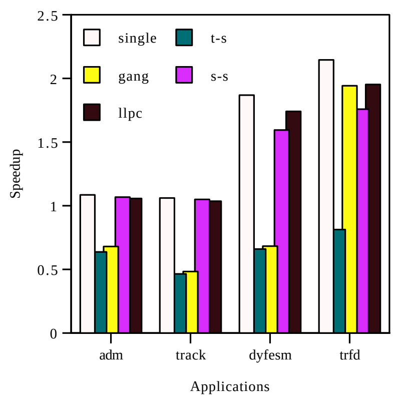

Figure 8: Speedup comparison of the different policies.

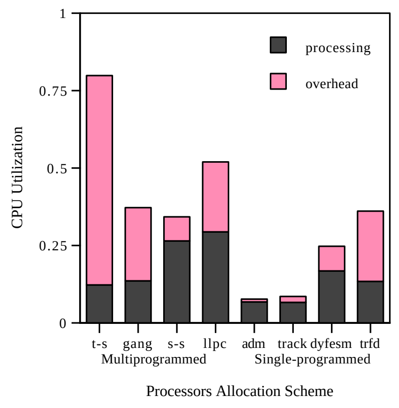

Figure 9: Utilization comparison of the different policies.

To compare the system utilization, we divided the cpu time into two categories: *processing* and *overhead*. The processing time is the total of all the useful computation time, while the overhead time includes all multiprocessing overhead, such as process creation and synchronization overhead, and multiprogramming overhead, such as context switching. Figure 9 compares the system utilization of the four policies when the four applications are executing simultaneously. Also shown in Figure 9 is the system utilization of the applications when they are executed on a dedicated system.

According to this figure, multiprogramming clearly utilizes more processing power than single-programming, with time-sharing having the highest system utilization and LLPC the next highest. However, most of the processing power with time-sharing is wasted in overhead. Although gang scheduling has a much smaller overhead, it does not utilize much more processing power for useful computation than time-sharing. Space sharing has the lowest overall system utilization but it produces the smallest overhead. It also utilizes more than 25% of the total processing power for useful computation. Although LLPC requires a larger overhead than space-sharing, it produces the highest system utilization (29%) in terms of useful computation.

Figure 10 shows how the individual processors are utilized under the different schemes. Again, time-sharing utilizes all the processors, but most of the processing time is spent in overhead. Gang scheduling significantly reduces the overhead, but it does not fully utilize all of the processors. Space sharing utilizes most of the processors for useful computation but it still has some idle time. LLPC requires less overhead than time-sharing or gang scheduling and it utilizes more processors than space sharing.

6 Conclusions and Future Work

In this paper, we have proposed a processor allocation scheme, called loop-level process control (LLPC), for multiprogrammed shared-memory multiprocessors. LLPC uses the system load to limit the number of processes an application can create for each parallel loop. This policy minimizes the number of context switches, which in turn eliminates the process thrashing problem, improves the efficiency of synchronization operations, and better utilizes the cache memory. The preliminary simulation results show that our scheme performs better than both time-sharing and gang scheduling. It works as well as space-sharing scheme in terms of the speedup of individual applications while attaining higher system utilization than all of the other schemes. Moreover, LLPC does not require any special programming model or modifications to the operating system, and it can be transparent to the programmer. It also has better portability and compatibility with existing systems. Therefore, it is a simple and effective method to achieve high system utilization without degrading the performance of individual applications.

In this study, we used simple chunk scheduling inside LLPC to assign iterations to the processes. However, this may cause imbalances among the processors' workloads. It has been shown that the performance of an application can be further improved by using dynamic loop scheduling to balance the workload [17]. We are planning to incorporate this type of loop scheduling into the loop level process control policy.

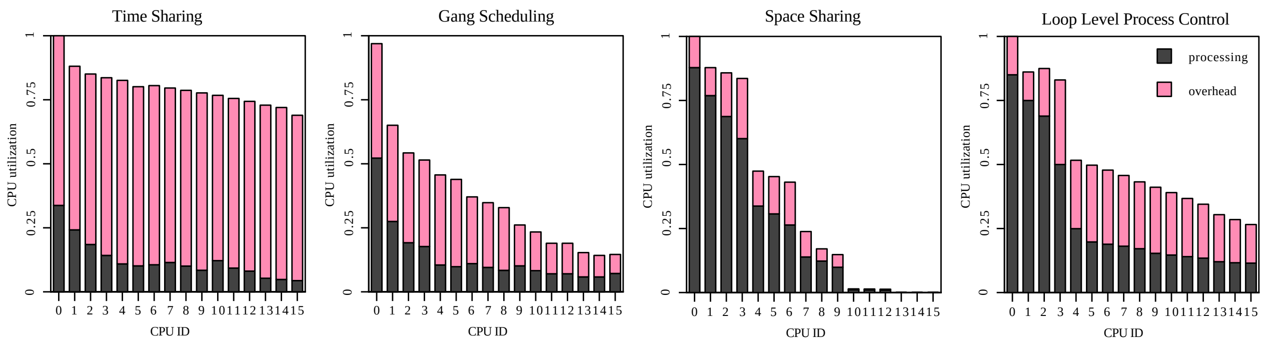

Figure 10: Comparison of the distribution of the CPU time.

7 Acknowledgements

We would like to thank Trung N. Nguyen for providing the program traces and for his comments on the trace generation methodology.

References

[1] Alliant Computer System Corp. *FX/Series Product Summary.* Littleton, MA, 1986.

[2] B. Bauer. *Practical Parallel Programming.* Academic Press, San Diego, 1992.

[3] M. Berry, D. Chen, P. Koss, D. Kuck, and S. Lo. The Perfect Club benchmarks: effective performance evaluation of supercomputers. *International Journal of Supercomputer Applications*, pages 5–40, Fall 1989.

[4] Cray Research Inc. *CRAY Y-MP, CRAY X-MP EA, and CRAY X-MP Multitasking Programmer's Manual.* Eagan, Minnesota, 1991.

[5] R. Dimpsey and R. Iyer. Modeling and measuring multiprogramming and system overheads on a shared-memory multiprocessor: Case study. *J. of Parallel and Distributed Computing*, 12:402 – 412, 1991.

[6] D. Feitelson and L. Rudolph. Wasted resources in gang scheduling. In *5th Jerusalem Conf. on Information Technology*, pages 127–136, oct 1990.

[7] D. Feitelson and L. Rudolph. Gang scheduling performance benefits for fine-grain synchronization. *J. of Parallel and Distrib. Computing*, 16:306–318, 1992.

[8] A. Gupta, A. Tucker, and S. Urushibara. The impact of operating system scheduling polices and synchronization methods on the performance of parallel applications. In *Conf. on the Measurement and Modeling of Comp. Sys.*, volume 19, pages 120–132, 1991.

[9] S. Leutenegger and M. Vernon. The performance of multiprogrammed multiprocessor scheduling policies. In *Conf. on the Measurement and Modeling of Comp. Sys.*, volume 18, pages 226–236, 1990.

[10] D. Lilja. Exploiting the parallelism available in loops. *Computer*, pages 13 – 26, February 1994.

[11] V. Naik, S. Setia, and M. Squillante. Performance analysis of job scheduling policies in parallel supercomputing environments. In *Supercomputing*, pages 824–833, 1993.

[12] C. Natarajan, S. Sharma, and R. Iyer. Impact of loop granularity and self-preemption on the performance of loop parallel applications on a multiprogrammed shared-memory multiprocessor. In *1994 International Conference on Parallel Processing*, volume II, pages 174–178, August 1994.

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

[14] C. D. Polychronopoulos, M. Girkar, M. R. Haghighat, C. L. Lee, B. Leung, and D. Schouten. Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *International Conference on Parallel Processing*, volume II, pages 39–48, 1989.

[15] Silicon Graphics Inc. *Symmetric Multiprocessing Systems.* Mountain View, CA 94043, 1993.

[16] A. Tucker and A. Gupta. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. In *12th ACM Sym. on Operating System Principles*, pages 159–166, 1989.

[17] K. Yue and D. Lilja. Parallel loop scheduling for high performance computers. In J. Dongarra and et. al., editors, *High Performance Computing: Issues, Methods, and Applications.* Elsevier Science B.V., Amsterdam, The Netherlands, 1995.

[18] J. Zahorjan, E. Lazowska, and D. Eager. Spinning versus blocking in parallel systems with uncertainty. In *Int. Sem. on Performance of Distributed and Parallel Systems*, pages 455–472, 1988.