Multiprocessor Scheduling: A Survey

Brad Huffaker Sean Peisert Otto Sievert Eric Tune

November 21, 1998

Department of Computer Science and Engineering
University of California
San Diego, CA

Abstract

We analyze the problems inherent in scheduling tasks to run on multiprocessor architectures. We present three levels into which scheduling decisions can be decomposed: short-term, long-term and loop. To schedule efficiently, one must consider maximizing processor utilization, minimizing communication and memory-access costs, and effectively synchronizing dependent tasks. We examine a variety of techniques to address these goals, and consider what architectures each technique is appropriate for. Finally, we predict what architectures will be most prevalent and which techniques will be the most valuable in the future.

1 Introduction

In this paper we investigate how to schedule tasks to run on multiple processors. To this end we present a framework for organizing ideas about scheduling and multiprocessors. Then we present current implementations of those scheduling techniques. In this introduction, we will introduce the need for effective scheduling by examining a number of performance problems that may occur on multiprocessor systems, and which may be avoided though careful implementation of scheduling policies. We will look at how scheduling can be decomposed into three different levels and define several different types of multiprocessor architectures.

We decompose scheduling into three levels: *short-term*, *long-term*, and *loop*. In the third, fourth, and fifth sections, respectively, we probe deeper into the problems each level must address and descibe specific solutions. The sixth section describes how task migration can be used to shift currently executing tasks between processors. In the seventh section, we predict which scheduling techniques will be most important in the future.

To avoid misconceptions, we reject overloaded terms, such as *process* and *thread*. We use the term *task* to mean the smallest unit of execution. That is, for every task, running or not, there is a corresponding program counter. We use the term *program* to mean a body of code comprised of one or more tasks, to perform some logically coherent operation. We use the phrase *task group* to mean a set of tasks which interact frequently with each other. Note that while these task groups might typically be comprised of tasks from the same program, this need not be the case.

1.1 The Three Scheduling Problems

Let us consider various reasons why a program might not be running as fast as it possibly could. One reason that a program might not be running as fast as it could is that some processors might not have work assigned to them. For example, If we have assigned a fixed portion of a task to each processor but some tasks finish sooner than others, the finish time of the program is limited by the finish time of the last processor. This is an example of poor *processor utilization*.

Another reason that a program might not be running efficiently is that a task might be waiting on communication and/or non-local memory access.

By waiting on communication, we could mean *waiting to receive a message*, or accessing non-local (slower) memory. Communication may be delayed because of contention for the communication medium. It may be possible to arrange tasks on processors so that there is less communication contention. If this were the case, we would call this decreasing *communication/memory-access costs*.

A task may also be waiting on communication because the task it communicates with is not running. For

[1]This work supported by a generous grant from no one at all[5].

example, in a producer-consumer relationship, the consumer may be idle if the producer has not sent any data for the consumer to do work on. No more data will arrive during the consumer's time quantum if the producer is not running simultaneously. We call this poor *synchronization effectiveness.*

The three scheduling goals further defined:

- Good Processor Utilization:
  All processors have work in their queues at all times. All processors which have tasks assigned to them from the same program finish execution at the same time, thus the user gets the expected speedup. Processors spend most of their time doing useful work rather than coordinating the division of work.

- Good Synchronization Effectiveness:
  Task are scheduled in such a way that interacting tasks across processors can cooperate effectively. This means that tasks with fine grained interaction should be running at the same time.

- Low Communication/Memory-Access Cost:
  Tasks are scheduled in such a way that communication time, either message passing or shared-memory latency is accounted for, and minimized. Scheduling data structures should be arranged so that they are not a source of contention. Tasks are scheduled so that the cache state built up by one process is not unneccesarily ruined by another.

1.2 The Three Scheduling Levels

As mentioned before, we divide the problem of scheduling into three levels, which we now discuss.

1.2.1 Short-term Scheduling

In time-sliced systems, it is necessary at each scheduling quantum to quickly choose a new task to run. Additionally, when a task *blocks*, one must decide whether or not to immediately context-switch. Normally, there is a fixed time quantum during which a task may execute for the whole quantum or the task may yield to another process during this quantum. The short term scheduler itself is typically a very small piece of code which is part of the operating system kernel. Thus, the purpose of short-term scheduling is to select when to do a context-switch between the current task and the next highest priority task.

1.2.2 Long-term Scheduling

In all systems, there must be some order in which tasks are executed. In a *batch* or *hardware-partitioned* machines, where there is no time-sharing of processors, the long-term scheduler may simply be a queue of jobs to run one after the other. The most interesting problems arise in *multiprogrammed* machines, where the long-term scheduler determines what jobs are ready to run and orders them, possibly in a fashion guaranteeing fairness. Since this may be a time-consuming calculation, we normally only evaluate long-term schedules at intervals much longer than the time quantum length. This is normally considered to be part of the operating system, but the code size and time required for long-term scheduling may be much greater than that required for short-term scheduling. Thus, long-term scheduling assigns tasks to processors and assigns their priority in a given time period.

1.2.3 Loop Scheduling

In some types of programs, there is a pool of work which can be allocated to any number of independently-working tasks. It is desirable for the work to be divided in such a way that all the tasks finish at the same time. This minimizes the overall execution time of the program. Frequently, this work is the iterations of a loop, hence the name *loop scheduling.* On a time-sliced machine, the throughput may be unaffected by load-balancing decisions. However, on a batch-oriented or hardware-partitioned machines, an increase in run-time directly corresponds to a decrease in throughput. This scheduling functionality would typically be implemented in the compiler-runtime library or a user library. Not all programs with multiple tasks can benefit from loop scheduling.

2 System Architectures

The architectural details of the machine a task is running on will affect what techniques a scheduler should implement. The most important distinction is the speed and nature of communication between processors and memory. *Shared-memory* machines are those in which all processors access the same memory. In a *uniform-memory access* (UMA) architecture, all processors can access all memory at the same latency. An example of a machine containing this type of architecture is a two-processor Pentium PC.

In a *distributed memory* machine, individual processors must communicate over a network to access non-local memory. Non-local memory access is thus much slower than local memory access. An example of this is a *network-of-workstations* (NOW), such as a network of DEC Alphas.

A *non-uniform memory access* (NUMA) machine might be thought of as a hybrid between a NOW and UMA architecture. Non-local memory access costs are generally lower than with a NOW but higher than with an UMA machine. This is usually achieved through complex cache-coherency protocols. A NUMA is a hierarchical collection of nodes, connected via a high-speed interconnect, and processors within nodes. There is shared memory between the processors in the node and message-passing between the nodes themselves. Memory access time varies depending on whether the access is to local or to slower, non-local memory. Examples of this type of machine are the IBM SP2, Sun Starfire, and SGI Origin 2000[1].

In an UMA machine, all the processors and all the memory access costs are the same. This is an example of a homogeneous system. In contrast, a NOW may have varying communication costs and processor speeds. An example of a homogeneous system would be a group of Macs and DEC Alphas connected by both Appletalk and switched Ethernet.

Another consideration, which is really a scheduling policy decision, is how multiple programs will be run on the multiprocessor. For our purposes, we will say that a scheduler which runs each program sequentially is a *batch* scheduler and one that can run more than one program at the same time is *multiprogrammed*. There are two techniques for implementing multiprogramming: *hardware partitioning* and *time-slicing*. In hardware partitioning, one program runs exclusively on a subset of all processing elements. Note that since hardware-partitioning and batch scheduling both have no context switching, many of the same arguments about scheduling apply to both. In time-slicing, a task runs for a small *quantum* of time, and then the processor context switches to run another task.

3 Short Term Scheduling

Ousterhout describes *pauses* as a technique to increase *synchronization effectiveness*[14, 4]. His experiences were based on working on the Medusa operating system running on the cm∗ multiprocessor[15]. The basic concepts presented by this older work are very important, while its techniques have been challenged by more recent work[18]. Pauses are where a task remains idle while waiting for a system call rather than yielding to another task.

Ousterhout considers systems that implement multiprogramming through time-slicing. In this scheme, a task runs for a certain quantum of time before being preempted by the short-term scheduler so that another task can run (not at all necessarily a task from the same program). If the task has to wait for some system call, then it may end its time quantum early.

Consider the ways in which tasks may communicate. A number of UNIX commands may communicate by means of buffers (called pipes) between them. So, when one producer adds one chunk of processed information to the buffer, it need not wait for the consumer to take it. Assuming the buffer is not full, it can continue to run.

However, if one task is slower, then the buffer will eventually fill up or become empty, and the faster task will have to wait. Either it waits trying to read an empty buffer, or it waits trying to write a full buffer.

Further, for certain types of programs (for example *Successive Over-Relaxation* [2], a numerical boundary value problem solver) a task which sends a message cannot continue until it gets a response. This is called *barrier synchronization*. In this case it is necessary to block on communication.

Now consider two processes communicating using blocking communication methods. The sender has to execute some operation to *send* its data, and the receiver must execute some operation to *receive* the data[1]. Since the two programs are not likely to do this at exactly the same time, whichever executes their part of the communication operation first will have to wait for the other.

In a naîve operating system implementation, the execution of the communication primitive will cause the waiting task to yield its processor. A context switch will occur and another, unrelated task will run. The first task is switched out, and then the second task may complete its part of the communication shortly thereafter. But the first task is no longer running. In this fashion, the rate of communication may be limited to one communication operation per time quantum. For tasks with fine-grained interaction, this may be a huge waste of time. More specifically, we call this poor synchronization effectiveness.

The solution is to have the processor sit idle, rather than context switch, while the task waits for the communication to complete. It may pause for a certain amount of time, or for as long as possible. In either case, it is still preempted when its time-quantum is up. This is called a pause time. Sobalvarro, *et al* refer to these two options as *spin-only*, and *spin-block*. Spin-only means that a task stays idle until either its time quantum expires or communication completes. Spin-block means that the task idles for a fixed amount of time typically two times the context switching delay, and then, if communication is still not completed, it yields the remainder of its time quantum. So pause times are a short-term scheduling technique to improve synchronization effectiveness.

[1]Please note that while the vocabulary may be suggestive of a message passing system, it could just as well be a shared-memory system.

4 Long Term Scheduling

Ousterhout also considered *coscheduling*. If two interacting tasks are not scheduled at the same time, then pause times alone may not be sufficient to insure good performance. Consider the worst case scenario for two interacting tasks: the tasks are never scheduled at the same time. In this case, each task is limited to one communication per quantum. For finely interacting processes, this can be a real slowdown. This has been called *process thrashing*, since it is similar to the thrashing that occurs in virtual-memory systems[14]. In general, one may not just need to schedule two tasks at the same time, but any number of tasks which communicate interdependently. We call that set of tasks a *task group*[2]. It is desirable to have some long term scheduling policy that tries to schedule task groups during the same time quantum across various processors. Such a policy is called *coscheduling* [14].

Markatos and LeBlanc[13] conclude that coscheduling is better than *uncoordinated* scheduling, and that hardware partitions are better than coscheduling. This is hardly surprising since hardware partitioning insures that there is no time wasted in context switching and that a whole task group is always coscheduled.

There are a number of algorithms to determine a good long term schedule that coschedules task groups. For most systems, it may be desirable for such an algorithm to be fair to all tasks, and to coschedule each task group, if at all possible, and have good processor utilization. Ousterhout believed that a static scheduler would be the best method. In static coscheduling, there must be some *a priori* definition of task group. Given that task groups are predefined, there are a number of algorithms that develop long term schedules that are both fair and coschedule task groups. Ousterhout describes the Matrix method, the Continuous method, and the Undivided algorithm[14].

4.1 Dynamic Coscheduling

4.1.1 Shared Memory Multiprocessors

Scheduling techniques for all levels of scheduling may be broadly classified as *static* and *dynamic*[3]. Static means determined before runtime and dynamic means determined during runtime. The more frequently two processes interact, the more important it is that they are coscheduled. If two processes typically communicate much less often than once per quantum, then coscheduling is not necessary. But how should one determine what processes *interact frequently*? That is to say, which tasks comprise a task group? While Ousterhout suggests that the task groups should be statically identified (presumably by the programmer), Feitelson and Rudolph [7] suggest that the way to identify task groups is to identify them at runtime. One benefit of this is that the programmer is relieved of the burden of determining the best task groups. They implemented their technique on a BBN Butterfly, a shared-memory multiprocessor.

The idea is that tasks communicate though some predefined channels, and that by observing how frequently each channel was used one can infer which tasks form task groups. To better visualize this technique, imagine that all the tasks on the machine are the vertices of a graph, and the most frequently used channels are edges of that graph. Hopefully, that graph will be disconnected, consisting of a number of connected subgraphs. The set of tasks which are part of a connected subgraph are exactly the set of tasks which form a task group. In practice, this is implemented at runtime by a table that is constantly updated as communication occurs. The table has counts of the usage of each channel, and the long term scheduler consults this table to re-evaluate what tasks form task groups[7]. Another possible advantage of this scheme is that dynamic coscheduling can adjust as communication patterns change between tasks.

4.1.2 Distributed Multiprocessors

One disadvantage of this implementation is that a centralized data structure is needed to keep track of the communication patterns. While this may be practical on a shared-memory machine, it may be impractical on a distributed memory machine. Sobalvarro, *et al* [18] considered these architectures and presented a different implementation. They had a network of workstations connected by a very fast network. By very fast, we mean that message latency is two to twenty times less than the scheduling quantum. The significance of this is that two tasks may have many interactions during the course of one quantum, and therefore coscheduling (and pause times) are important.

They considered a system with a typical multiple-priority round-robin scheduler running independently on each processor. There is no provision in their system for explicitly scheduling two tasks at the same time on different machines. Instead they used a coprocessor on the network interface controller to observe the message traffic, and increase the priority of tasks which received messages. While increasing the priority of a task does not immediately cause the task to start, it makes it more likely that the process will be the next to be selected to run by the short term scheduler. Over the course of a number of messages and scheduling quanta, tasks that interact frequently will gradually be brought into synchronization across multiple processors. There are two

[2] Others have called this an *activity working set* or *task force*. We follow the time honored tradition of making up a new name for something that already has too many names.

advantages to this scheme. First, the scheduling information is not centralized in a single data structure. If this were so, it could be a source of communication contention. Second, it uses a coprocessor to collect profiles of task interaction, so that the processor is not burdened with that work.

5 Loop Scheduling

Atif and Hamidzadeh[9, 8] consider how to schedule tasks effectively on heterogeneous systems. They are primarily interested in increasing processor utilization by distributing work in such a way that all tasks of a program complete at nearly the same time. For example, assume that there is a pool of work consisting of W *work units* that can be grouped into work chunks and assigned to various processors. There might be 1000 pairs of integers. A work unit is one pair, since both are needed to do an addition. The total number of work units is 1000. As mentioned before, a work unit frequently corresponds to an iteration of a loop.

5.1 Static Scheduling

So, a naîve approach to obtaining maximum processor utilization would be to assign W/P work to each task, where there are W units of work, and a total of P tasks: one on each of P processors. But if not all the computers on the network can add as fast, or if some are connected by slower communication links, then not all processors will finish at the same time. Conversely, even if the system were completely homogeneous, but the work units were uneven (for example, separately sort 100 lists of greatly varying length) then there would again be poor processor utilization. This naîve approach is called *single-chunk static scheduling* [9]. This algorithm has poor processor utilization.

5.2 Dynamic Scheduling

The processor utilization problem might be solved as follows. The approach, called *pure self-scheduling* [16] is to give each task one unit of work, and then having the task take another unit of work after it completes the first. While this algorithm assures good processor utilization, it has very poor communications efficiency. This is because there must be some shared variable or structure which keeps track of what work units have been completed so far. This is a characteristic of techniques that determine scheduling during run-time. Such a class of techniques can be called *dynamic* , and techniques which schedule before run-time can be called, *static.* [3]

Now, if each chunk is composed of k work units, rather than just one, then the synchronization cost is reduced by a factor of k. This is called *chunk self-scheduling* [16]. Ideally one would like to make k very large; however, as k grows in size, so does the possibility of load imbalance (poor processor utilization).

It has been noted that load imbalance is critical only near the end of program completion. Thus, one might take larger chunks at the beginning of a program execution, and take smaller chunks at the end. This technique, *guided self-scheduling* [16, 10], gives low synchronization costs at the beginning of execution, and also preserves load balance at the end of execution. This is a compromise between large chunking and self-scheduling.

Since synchronization costs can be high, even relative to loss of throughput due to load imbalance (processor utilization), there is another algorithm which attempts to further reduce synchronization costs at the expense of some load imbalance. Like guided self-scheduling, *trapezoid self-scheduling* [20] distributes more iterations at the beginning of execution than at the end. Unlike guided self-scheduling, however, the size of successive chunks in trapezoid self-scheduling is a constant. This reduces synchronization cost near the end of execution.

Another algorithm is *distributed self scheduling.* First, all the work is statically allocated just as in single-chunk static scheduling. Then, the first processor to finish takes work back from the slower processors and gives it to the idle ones[9, 11].

5.3 Heterogeneity

If all the task units are the same size, and all the processors are homogeneous in their speed and communication, then single-chunk static scheduling is the way to go. Only in this case, each work chunk will take the same amount of time, so there will be good processor utilization. Since there is no run-time assignment of work, then there is a minimal communication cost.

But if not all the computers on the network can add as fast, or if some are connected by slower communication links, then not all processors will finish at the same time. Conversely, even if the system were completely homogeneous, but the work units were uneven, (for example, separately sort 100 lists of greatly varying length), then we would again have poor processor utilization. One technique that tries to directly account for both these problems is the *Self-Adjusting Scheduler for Heterogeneous systems (SASH)*[9] .

First, all but one of the tasks are assigned a chunk of work, but only a fraction of the total work is distributed at this point. The remaining processor attempts to then calculate an optimal schedule for the remainder of the work. Based on estimates of the computation speed and communication speed of each of the processors and the complexity of each unit of work, it tries to compute a schedule for the remaining, unassigned work that will have maximum processor utilization (i.e. same finish time for all tasks). When one of the other processors finishes before the optimal schedule is calculated, the processor doing the scheduling distributes some of the remaining work based on a partial schedule calculated so far. If the algorithm is effective in its estimation of optimal schedules, then there should be good processor utilization. If good sized chunks of work are initially distributed, and the schedule-estimation algorithm is quick, then there should be relatively little time spent communicating new task assignments.

5.4 Affinity Scheduling

The techniques discussed so far assume that the data that is associated with a work chunk is local to the task assigned that work chunk, or that the time to distribute the data is small enough to make the use of multiple processors worthwhile.

Affinity scheduling [12], like guided self-scheduling, attempts to repeatedly match certain work chunks with certain processors. In a program with a loop, each iteration of the loop might be a separate work unit. If that loop gets executed more than once (e.g. it is the inner loop of a nested loop) then it is advantageous to assign a processor the same chunk (in this case, a range of iterations of the inner loop) for each iteration of the outer loop. It is advantageous because it is likely the data needed by this iteration may still be cached from the previous one. While a number of authors suggest that this is the one of the best loop scheduling techniques, it is only useful when work chunks are repeatedly processed.

A chunk is only reassigned to a different processor when load imbalance becomes significant. This occurs by means of having a central processor whose sole function it is to partition the data, schedule it, synchronize, maintain memory locality and balance the load through a dynamic scheduling algorithm. The claim is that this is effective despite potential performance loss by a dedicating one processor exclusively to the central scheduling task.

6 Task Migration

We now present an approach to load balancing that is complementary to the long-term scheduling techniques discussed, yet provides unparalleled processor utilization at the expense of communication costs. All of the techniques discussed this far schedule by assigning work to tasks, and tasks to specific processors. Each task is sent to and executed entirely on its assigned processor. For batch-mode system architectures, time-sliced architectures where throughput is more important than processor utilization, or systems with few users, this approach is optimal. Once a task has begun execution, it stays with that processor until completion.

For systems like these, there is no advantage in moving a task from one processor to another, because processor utilization is adequately addressed by the other techniques described in previous sections, and because the cost to perform *task migration* is quite high.

There are systems which benefit from task migration. One example is a multiprogrammed system with dynamically changing processor loads including those that would occur when programs are added and removed from the system. In this case, techniques that assume homogeneous execution time would result in poor processor utilization. A widely varying workload can even foil techniques such as guided self-scheduling, which actively attempt to balance the workload among processors. Another example of a system that would benefit from task migration is a network of workstations. In typical commercial installations, a single user has primary control over his or her personal networked workstation. However, in this situation there are almost always workstations which are not currently in use. A user with a parallel program could benefit from distributing the program to execute on workstations other than his/her own.

In both of these situations, a processor load imbalance is the issue. Task migration complements fine-grained long-term scheduling techniques in that it provides an alternative method of achieving good load balance. However, task migration is a complex process which requires the following actions:

- the migrating task must be halted;
- any processor state associated with that task, such as processor registers, instruction reorder buffers, or state vectors must be transferred to the new processor;
- all virtual memory pages must be transferred to the new processor;
- operating system specific information including file lock information must be transferred to the new processor;
- the task itself must be moved to the new processor and resumed.

In addition to the penalties associated with the above migration activities, one must also consider the cost associated with the loss of cached information, whether that is instruction cache, data cache, or file cache. All of these costs are significant; task migration is not performed without undue need.

The simplest method of implementing the task migration activities is to simply halt the task, move the task and associated baggage, and restart the task. This is the process used in the LOCUS operating system[17].

This procedure may be improved upon, however. For example, the V kernel implements *pre-copying*, where data is migrated while the task continues to execute[19]. This may require that some data be copied twice (if it changes after being pre-copied). Once the task's baggage has been migrated, the task is halted, transferred (with any remaining data or state), and immediately resumed.

Another migration method was implemented in the Accent[21] operating system and in the Sprite operating system[6]. When a task is migrated in Accent or Sprite, it is halted immediately, transferred with minimal state, and immediately resumed. Data and other state are migrated only as the task requires. This procedure, called *lazy-copying*, decreases the task execution downtime, but requires that the original processor maintain information for some time after the task has been migrated.

All of these migration techniques must be implemented with policies which determine, for example, when load imbalance is great enough to initiate task migration, or what other events (a user returning to their personal workstation and finding it bogged down with their neighbor's lawn-watering analysis program) may trigger migration. Additionally, the selection of a new processor to migrate to is not trivial. While providing an alternative method of load balancing, the high execution time costs and difficult policy problems are reasons why process migration is not widely used today.

7 Future Trends

While coscheduling and pause times were once only a technique for shared-memory multiprocessors, the recent development of very fast, optimized networks has definitely extended the applicability of coscheduling and pause times to distributed systems[18].

Current trends in technology indicate that the gap between processor speed and memory latency will continue to widen. This means that cache misses will be increasingly important to avoid. For this reason, affinity based scheduling should become increasingly important[13], and process migration should become less important.

On the other hand, if increasingly heterogeneous systems are built, techniques such as self-adjusting scheduling that consider communication costs in order to load balance are likely to perform significantly better than algorithms that do not. Increasingly heterogeneous distributed systems may arise, due to the proliferation of global networking, and heterogeneous shared-memory systems may be more common because of the scaling limitations of UMA architectures.

8 Conclusion

All of the scheduling techniques addressed in this paper are primarily concerned with improving execution time of individual programs on multiprocessor machines. The job of the scheduler is to assign tasks to processors. Because of the complexity inherent in determining an optimal schedule, effort has been focused on finding heuristic approaches. These approaches attempt to address the major costs in multiprocessor execution. These areas are processor utilization (balancing loads, and minimization of time spent dividing work), communication/memory-access costs, and synchronization effectiveness.

To help analyze these different approaches we divided the problem of scheduling into three different levels, short-term, long-term, and loop. The primary goal of short-term scheduling is to select when to do a context-switch between the current task and the next highest priority task. Long-term scheduling assign tasks to processors and assigns their priority in an given time period. Loop scheduling determines how to divide the work of one program among its various tasks/processors in order to minimize the overall execution time.[3]

Pause times are a technique for improving synchronization efficiency in short term schedulers. Coscheduling is a long-term scheduling technique for improving synchronization efficiency. Depending on the application and machine architecture, different loop level scheduling techniques may be achieve best processor utilization. There are both static and dynamic approaches to both loop and long-term scheduling.

References

[1] Baden, S. personal communication.

[2] Carter, J. B., Bennett, J. K., and Zwænepœl, W. Implementation and performance of Munin. In *Thirteenth ACM Symposium on Operating Systems Principles* (Oct. 1991), pp. 152–164.

[3] Casavant, T., and Kuhl, J. G. A taxonomy of scheduling in general-purpose distibuted computering systems. *IEEE Transactions on Software Engineering 14*, 2 (Feb. 1998), 141–154.

[4] Clark, R., O'Quin, J., and Weaver, T. Symmetric multiprocessing for the AIX operating system. In *Digest of Papers, COMPCON '95* (Mar. 1995), pp. 100–115.

[3] Editors note: if you read nothing else in this paper, read this paragraph. It is the best paragraph in the whole paper. Perhaps it is not surprising that the authors spent the most time arguing over these 4 sentences.

[5] Doof, U. R. A. Do people actually read bibliographies? *Journal of Computational Psychology 0*, 13 (1903).

[6] Douglis, F., and Ousterhout, J. Transparent process migration: Design alternatives and the Sprite implementation. *Software - Practice and Experience 21*, 8 (Aug. 1991).

[7] Feitelson, D., and Rudolph, L. Coscheduling based on runtime identification of activity working sets. *International Journal of Parallel Programming 23*, 2 (1995).

[8] Hamidzadeh, B., and Lilja, D. J. Self-adjusting scheduling: An on-line optimization technique for locality management and load balancing. In *International Conference on Parallel Processing* (Aug. 1994), vol. II:Software, pp. 39–46.

[9] Hamidzadeh, B., Lilja, D. J., and Atif, Y. Dynamic scheduling techniques for hetrogeneous computing systems. *Concurrency: Practice and Experience, Special Issue on Resource Management in Parallel and Distributed Systems 7*, 7 (Oct. 1995), 633–652.

[10] Hummel, S. F., and Schonberg, E. Factoring: A method for scheduling parallel loops. *Communications of the ACM 36*, 8 (Aug. 1992).

[11] Liu, J., and Saletore, V. A. Self-scheduling on distributed memory machines. In *Proceedings Supercomputing '93* (Nov. 1993), pp. 814–23.

[12] Markatos, E. P., and LeBlanc, T. J. Using processor affinity in loop scheduling on shared-memory multiprocessors. In *Proceedings of Supercomputing* (1992), pp. 104–113.

[13] Markatos, E. P., and LeBlanc, T. J. Locality-based scheduling for shared-memory multiprocessors. *Current and Future Trends in Parallel and Distributed Computing* (1995).

[14] Ousterhout, J. K. Scheduling techniques for concurrent systems. *Proceedings of Distributed Computing Systems* (Oct. 1982), 22–30.

[15] Ousterhout, J. K., Scelza, D. A., and Sindhu, P. S. Medusa: An experiemnt in distributed operating system structure. *Communications of the ACM 23*, 2 (1980).

[16] Polychronopoulos, C., and Kuck, D. Guided self-scheduling: A practical scheme for parallel supercomputers. *IEEE Transactions on Computers C-36*, 12 (Dec. 1987), 1425–1439.

[17] Popek, G. J., and Walker, B. J. The LOCUS distributed system architecture. *Computer Systems Series* (1985).

[18] Sobalvarro, P. G., Pakin, S., Weihl, W. E., and Chien, A. A. Dynamic coscheduling on workstation clusters. In *Proceedings of the Workshop on Job Scheduling Strategies for Parallel Processing* (Mar. 1998).

[19] Theimer, M., Lantz, K., and Cheriton, D. Preemptable remote execution facilities for the V system. In *Proceedings of the 10th Symposium on Operating System Principles* (Dec. 1985), pp. 2–12.

[20] Tzen, T., and Ni, L. Dynamic loop scheduling on shared-memory multiprocessors. In *Proceedings of International Conference on Parallel Processing* (1991), vol. II, pp. 247–250.

[21] Zayas, E. Attacking the process migration bottleneck. In *Proceedings of the 11th ACM Symposium on Operating System Principles* (Nov. 1987), pp. 13–22.

About the Authors

Brad Huffaker maintains his stoic appearance by heading a startup company whose vision statement explicitly involves the terms *Microsoft* and *acquisition*.

Sean Peisert is amazingly democratic, but we don't hold that against him, because his knowledge of things parallel is nigh-unbounded.

When not wasting his employers time and money, Otto Sievert enjoys hacking LaTeX . His interests also include the wholesale ginger ale market.

Eric. S. Tune raises inchworms, talks excessively in his classes, surfs, and drinks ginger ale. Currently his interests include procrastination algorithms.