MULTIPROGRAMMING AND MULTIPROCESSING

Most multiprocessors, and certainly all large-scale multiprocessors, represent an enormous capital investment with a short half-life. As an example of the expense associated with these machines, in 1996 the United States Department of Energy purchased a 9072-node multiprocessor from Intel Corporation for roughly $55 million (http://www.sandia.gov/media/teraflop.htm). Smaller configurations costing between $1 M and $10 M are often purchased by commercial companies and universities. This investment in hardware must be amortized over a short period of time because most multiprocessors become obsolete within a few years. Processor speeds improve so quickly that a new generation of multiprocessors appears every three years with faster processors, larger memories and caches, and improved communication networks. To amortize the cost, multiprocessors are often purchased by consortiums of institutions and departments, representing a large number of end users and applications.

Given the costs of large-scale machines and their relatively short lifetime, each machine must be utilized fully in order to recoup the investment. Furthermore, the user community in a consortium will typically place severe response-time demands on the shared resource. To maximize utilization of the resource and minimize throughput for users, multiprocessors must be multiprogrammed, just like the large mainframe computers of the past.

This expectation of multiprogramming of multiprocessors is further exacerbated by architectural trends. In particular, we can expect current and near future multiprocessors to be constructed using networks of workstations (NOWs) (1), wherein high-performance workstations are connected by a high-speed interconnection network. The very nature of the NOW architecture, which allows an individual workstation to be used both as a workstation and as a node in a multiprocessor, suggests that several applications (sequential, parallel, and distributed) will share a NOW-based multiprocessor at the same time.

The problem of multiprogramming a single processor among competing sequential applications is well understood, having been studied both theoretically and experimentally since the 1960s. The problem of multiprogramming a multiprocessor among competing applications, which may include sequential, parallel, and even distributed programs, is a subject of on-going research. Several new dimensions to the problem complicate the search for solutions. First, a multiprocessor may have thousands of processors, and the multi-

programming policy must strive to keep them all busy. Thus, a simple extension of traditional multiprogramming approaches (wherein each application uses the machine for a brief period of time and then passes it on to another) will not suffice because many applications cannot fully utilize all processors. Second, parallel applications have multiple processess, and these processes interact (via synchronization and communication) during execution. Therefore, the processes that make up a single application cannot be scheduled independently since they must be executing simultaneously in order to interact efficiently. Third, multiprocessors typically have complex memory hierarchies that include local and remote memory, multi-level caches, and hardware coherence protocols. Any scheduling policy designed to maximize throughput or utilization must take these architectural features into account since they are important factors in application performance.

An appropriate multiprogramming policy must simultaneously address multiple issues: fairness to each application, overall system utilization, the underlying architectural features, and the differing needs of sequential and parallel applications. Different policies stress different aspects of the problem, depending on the particular features of the system under consideration. In this paper, we consider each of these issues and describe multiprogramming policies designed to address them. We also explain the relative advantages of each policy and suggest which policies are most appropriate for each multiprocessor environment.

MULTIPROGRAMMING SHARED-MEMORY MULTIPROCESSORS

The first widely available multiprocessors were small-scale bus-based cache-coherent shared-memory machines such as the Sequent Symmetry and Balance, the Encore MultiMax, and the DEC Firefly workstation. In these machines, there was a single main memory accessed through the shared bus, and each processor was equipped with a small cache, so that the most frequently accessed data could be kept nearby. Since all of the main memory was accessible by all processors for roughly the same cost, these machines became known as uniform-memory-access (UMA) multiprocessors.

One defining attribute of an UMA multiprocessor is that a process can be executed on any processor since all processors share the same main memory. The cost of moving a process from one processor to another is dominated by the cost of refilling the local cache, but the cache sizes on these early machines (4 kB on the Sequent Balance and 64 kB on the Sequent Symmetry) were such that this cost was not significant.

These early multiprocessors were often used as central servers that had to accommodate multiple users running both sequential and parallel applications. The operating systems on these early multiprocessors were extensions of UNIX, and the multiprogramming policy was a straight-forward extension of the UNIX scheduling policy. There was a single ready queue of processes waiting to execute, and idle processors would take the process at the head of the queue and execute it for one quantum. At the end of the quantum, the operating system would suspend the process and put it back in the central ready queue.

Under this policy, all processes have a chance to make forward progress. Furthermore, this policy perfectly balances the load since no processor remains idle while there exist processes to be executed. Unfortunately, this policy is not always suitable for parallel applications, even though it is effective for time-sharing workloads of sequential jobs. The problem is that it doesn't treat all applications fairly. The policy is fair to each individual process in that over a long period of time, all processes receive roughly the same amount of computing power. However, not all applications create the same number of processes, and an application that creates an artificially large number of processes receives additional quanta as a result. Malicious users could easily monopolize the system by creating extra processes. Although this situation exists in traditional uniprocessor systems too, the complexity of adding processes to a sequential application serves as a barrier against malicious behavior. The problem is more acute in a multiprocessor environment, where almost every application already contains some number of processes, and the barrier to adding more is low.

The round-robin job (RRjob) scheduling policy (2) was developed to provide fair treatment to each application. Instead of a central ready queue for processes, RRjob keeps a central queue of applications (or jobs). Scheduling is performed in a round-robin fashion among the applications. Each time an application comes to the front of the queue, it receives P quanta of size q, where P is the number of processors, and q is a basic quantum size on each processor. If the application has N processes (where $N, < P$), it receives N quanta of size $P/N \times q$. By changing the number and size of quanta assigned to each application, RRjob ensures that all applications receive the same aggregate computing power. Each idle processor takes the first process out of the first application of the central queue of applications, executes it for one quantum (of size $P/N \times q$), then context switches and takes the next process in the front of the queue. Since neither all processors context switch at the same time, nor all processes receive the same quantum, it is not guaranteed that all processes of an application will be scheduled for execution at the same time.

Both RRjob and the simple UNIX-extension policy impose significant overheads on parallel programs because neither policy respects the constraints imposed by a parallel application's need to communicate and synchronize. Neither of these policies makes an attempt to coordinate the scheduling of processes that belong to the same application. In particular, there is no guarantee that the processes comprising a single application are scheduled for execution together. If a currently executing process needs to synchronize with another process in the same application, and that other process is currently suspended, the synchronization operation cannot complete until the suspended process is scheduled for execution, by which time the original process may have been suspended. If the operating system scheduler decides to preempt a process (due to quantum expiration) that is inside a critical section, then no other process will be able to enter the critical section until the preempted process is rescheduled for execution and exits the critical section. In such cases, the speed at which a process is able to execute depends not on how many processor cycles it receives, but whether the process is able to make forward progress when it receives the processor cycles (due to synchronization constraints). Both simulation and experimental results suggest that the possibility of preemption

within a critical section may degrade the performance of an application significantly (3). Any multiprogramming policy designed to meet the needs of parallel applications must deal with these effects.

Coscheduling

The problem of efficiently scheduling cooperating processes was first addressed experimentally by Ousterhout (4) in the Medusa multiprocessor operating system. Ousterhout suggested an analogy between scheduling in parallel processors and disk thrashing in early paging systems. Just as applications have a working set of memory pages that should be resident in main memory at the same time (so as to avoid delaying execution while waiting for a page request from disk), parallel applications have an activity working set that consists of all processes that interact with each other frequently. Processes that belong to the same activity working set must be *coscheduled* (that is, scheduled for simultaneous execution) in order to ensure efficient communication and synchronization. If the scheduler does not coschedule all processes that belong to the same activity working set, then the system will suffer from activity thrashing. In such a case, the progress of the parallel program is limited by the rate at which processes are scheduled for execution, rather than the speed of the processors.

In coscheduling, all processes of an application (which usually comprise an activity working set) are scheduled to run at the same time. As an example, consider an 8-processor system with three applications, A_1, A_2, and A_3, requiring 4, 4, and 8 processors, respectively. Under coscheduling, the processes of applications A_1 and A_2 can run in parallel during one scheduling quantum, after which all processors context switch at the same time, and then execute the processes of application A_3. When another quantum expires, the process repeats, so that all the processes of an application always execute simultaneously, and yet all processors are utilized to the fullest extent possible. As this example makes clear, another goal of coscheduling is to maximize the number of applications that are coscheduled, so as to maximize processor utilization. To achieve both goals simultaneously (minimize the extent of activity thrashing and maximize utilization), Ousterhout proposed three algorithms that allocate processors and schedule processes: the matrix algorithm, the continuous algorithm, and the undivided algorithm.

Matrix Algorithm. Under the matrix algorithm, the multiprocessor is assumed to consist of P processors, each of which can be multiplexed among at most A applications. The algorithm maintains an $A \times P$ allocation matrix, representing the process that will be run by each processor at each quantum. The algorithm works as follows:

- *Allocation:*. To accommodate a new application with p processes, the rows of the matrix are scanned (starting from row 0), until a row with at least p empty slots is found (success), or all A rows have been examined (failure).
- *Scheduling:*. The scheduler multiplexes the machine among the different rows of the matrix. During quantum 0, each processor executes the corresponding process assigned to row 0. At the end of the quantum, all processors perform a context switch (at the same time) and begin executing the corresponding process in row 1. This process continues through each row of the matrix, and then returns to row 0.

The reservation matrix for our earlier example (with applications A_1, A_2, and A_3) would look like this:

A_1	A_1	A_1	A_1	A_2	A_2	A_2	A_2
A_3	A_3	A_3	A_3	A_3	A_3	A_3	A_3

Unfortunately, it is not always easy to fill up the allocation matrix as nicely as the above example suggests. In particular, if application A_2 completes its execution and is removed from the allocation matrix, then the matrix would look like this:

A_1	A_1	A_1	A_1				
A_3	A_3	A_3	A_3	A_3	A_3	A_3	A_3

Under this allocation scheme, A_1 will be coscheduled on one half of the machine during its scheduling quantum, while the other half of the machine remains idle, thereby wasting processor cycles. Any attempt to utilize those wasted cycles by assigning them to A_3 is of limited benefit, since only half of the processes in A_3 could be assigned to the idle processors, and those processes would have to suspend execution as soon as they needed to synchronize with any of the processes not allocated to an idle processor.

If application A_3 completes its execution and another application, A_4, requiring five processors enters the system, then the matrix would look like this:

A_1	A_1	A_1	A_1				
			A_4	A_4	A_4	A_4	A_4

In this scenario, 7/16 (44%) of the available slots are unused. It is easy to see that in the worst case about half the processors remain idle or execute processes of non-coscheduled applications. Using simulation, Ousterhout concluded that on average, about 80% of the processors are executing coscheduled applications, while the remaining 20% of the processors are either idle or executing processes belonging to non-coscheduled applications.

Fortunately, pathological cases are uncommon, in part because the number of processors requested by many parallel applications is a power of two. In that case, more effective allocation algorithms are possible. For example, Feitelson and Rudolph (5) suggest a hierarchical scheduler that is reminiscent of the buddy system for memory allocation. In this scheme, all processors are organized as leaves in a binary tree, where the inner nodes of the tree are called controllers. A request for 2^n processors is passed to the appropriate level in the tree, where the peer controllers cooperate to assign the application to the least loaded controller. The result of this policy is coscheduled applications and a balanced workload across processors.

Continuous Algorithm. Most of the drawbacks of the matrix algorithm arise because all processes of an application are al-

located within the same row of the matrix, thus leaving several "holes" in the allocation matrix. The continuous algorithm was designed to overcome this problem by allocating processes in a linear array of scheduling slots, instead of a two-dimensional matrix of scheduling slots. In the linear array, the N_{th} slot is the $(N\ div\ P)_{th}$ slot of processor $(N\ mod\ P)$. Thus, the linear array is simply a row-major allocation of the matrix described in the previous algorithm. For example, if N equals 80, and P equals 8, then slot 1 is the first slot of processor 1, slot 2 is the first slot of processor 2, slot 3 is the first slot of processor 3, etc. Slot 9 is the second slot of processor 1; slot 10 is the second slot of processor 2, etc.

The continuous algorithm works as follows:

- *Allocation.* In a machine with P processors, initialize a window of width P, such that the leftmost slot in the window is the leftmost empty slot in the linear array. If there are enough empty slots in the window to accommodate the needs of the application, assign those slots to the application (success). If not, slide the window to the right, repeating the process until either a window position that can contain the entire applications is found (success), or the end of the linear array is reached (failure).
- *Scheduling.* Initialize a scheduling window of width P at the leftmost end of the linear array. At the beginning of each quantum, move the window one or more slots to the right, until the leftmost activity in the window is the leftmost activity of an application that was not coscheduled in the previous quanta.

The continuous algorithm is superior to the matrix algorithm, since it can pack activity working sets more nicely within the available slots. In fact, the matrix algorithm can be viewed as a special case of the continuous algorithm, in which the window *always* moves P slots to the right. The continuous algorithm can still suffer unused slots; however, as applications complete execution and free up their slots, the continuous algorithm may end up with lots of small sequences of empty slots, a phenomenon similar to external fragmentation of memory. In such a scenario, there are several empty slots to accommodate new applications, but these slots are widespead over the linear space of slots and, thus, cannot be used.

Undivided Algorithm. The undivided algorithm is an attempt to reduce the fragmentation that arises in the continuous algorithm. The undivided algorithm is identical to the continuous algorithm in every aspect, except that during allocation, all the processes of each new application are required to be contiguous in the linear array.

Demand-Based Coscheduling. All of the coscheduling algorithms proposed by Ousterhout were static, in that once an application was given some activity slots, it could not change them, even if the needs of the application changed. Thus, none of these algorithms can adapt to changes in communication and synchronization behavior. Demand-based coscheduling (6) was designed to make coscheduling work more like demand paging, in that only those processes that communicate or synchronize frequently need to be coscheduled.

Demand-based coscheduling requires some mechanism to identify processes that actively interact and, therefore, should be coscheduled. A trace of the messages in a message-passing system would identify the sender and recipient of a message, who would then become candidates for coscheduling. In shared-memory multiprocessors, a trace of cache invalidations may be enough to identify processes that actively share data. There is also lots of hardware state that can be examined to identify communicating processes. For example, if a page is actively shared by two processes, then the virtual-to-physical address translation of the page will be in the TLBs (translation lookaside buffers) of the two processors on top of which the two processes execute; communicating processes can be identified by examining TLB contents regularly.

Once communicating processes are identified, the demand-based coscheduling policy makes every effort to coschedule them. Under demand-based coscheduling, messages arriving at a node, if addressed to a process other than the one currently executing, may cause preemption of the running process in favor of the process to which the message is addressed, effectively achieving simultaneous execution of communicating processes.

Operating System Issues. Although coscheduling is a general-purpose scheduling method that avoids activity thrashing, it has not been implemented in many operating systems. Two problems arise in any practical implementation of coscheduling:

- *Processor Synchronization.* Under coscheduling, all processors must switch context at the same time. Some small-scale, bus-based multiprocessors have a common interrupt line that can be used for this purpose. Unfortunately, medium-scale and large-scale systems rarely have such hardware support. In these systems, another mechanism, presumably implemented in software, is needed. To make matters worse, context switching all processors at the same time in modern NOW-based multiprocessors is increasingly difficult due to the distributed nature of the system.
- *Single processes–daemons.* In most multiprocessor systems, there exist sequential applications and operating system daemons that are not parallel applications, and thus do not need coscheduling. In fact, these processes (especially operating system daemons) tend to steal processor cycles from coscheduled processes, undermining the coscheduling policy.

The first implementation of coscheduling was by Ousterhout; he used the matrix algorithm in the Medusa operating system (4). Twelve years later, coscheduling was implemented in the Psyche multiprocessor operating system (7,8), primarily for the purpose of comparing alternative scheduling policies experimentally. In Psyche, each processor has its own ready queue of processes; the matrix algorithm is used to determine when to run a process. All processors switch context at the same time, using a scalable barrier (9) embedded in the clock handler on each processor. The experimental results obtained from the Psyche implementation confirmed expectations: coscheduling processes that communicate frequently provide significant performance advantages

over asynchronous time-sharing of processes among processors.

Despite these positive results, coscheduling is still rarely used. One reason may be the assumption that any implementation will be at best 80% effective as predicted in Oustерhout's simulations (4). Another reason may be the modifications to the operating system kernel that are often difficult to implement. Another contributing factor has been the development of simpler mechanisms that, if exploited appropriately by user applications, can also reduce the synchronization overheads imposed on parallel applications by naive scheduling policies. Each of these mechanisms is based on the sharing of information between the operating system kernel, which is responsible for process scheduling, and the thread library, which is responsible for creating user-level processes and providing the appropriate communication and synchronization operations.

The Psyche operating system incorporates a mechanism called the *two-minute warning* (10). When a process is about to be preempted due to quantum expiration, the kernel initiates an upcall (the so-called two-minute warning) to the process. This upcall is nothing more than an asynchronous notification that the process is about to be preempted, in which case, the process should refrain from engaging in any synchronization activity that could delay other processes from making forward progress. In particular, if a process is about to be preempted, it should avoid entering a critical section. Experimental results suggest that the two-minute warning can improve performance by a factor of three or more by ensuring that no process is blocked while attempting to enter a critical section occupied by a process that is not running (10).

In the Symunix operating system (11), each process shares a *do-not-preempt-me* bit with the kernel. When a process is about to enter a critical section, it sets the do-not-preempt-me bit and resets it when it exits the critical section. When the scheduler is about to preempt a process due to quantum expiration, it checks the do-not-preempt-me bit first. If the bit is set, the scheduler does not preempt the process and allows it to execute for another quantum. Of course, a malicious user might leave the do-not-preempt-me bit set forever in order to monopolize the processor. To avoid this possibility, the scheduler abides by the do-not-preempt-me bit a limited number of times.

Scheduler activations (12) are another mechanism designed to avoid the overhead associated with preemption inside a critical section. Unlike Psyche and Symunix, scheduler activations allow processes to be preempted inside a critical section. Whenever a process requests a lock that is held by a preempted process, the preempted process is resumed for as long as it needs to exit the critical section. Then it gets suspended, and the process that requested the lock resumes execution and enters the critical section.

Kontothanassis et al. (13) propose novel synchronization primitives that interact with the operating system scheduler to reduce the overheads associated with synchronization in the presence of multiprogramming.

Space-Sharing Policies

All of the previously described policies (including RRjob and coscheduling) are *time-sharing* policies because they allow each individual processor to be time-shared (on a quantum basis) among processes that belong to different applications. Time-sharing policies tend to increase the cache miss rates since each scheduling quantum involves a new process which must load the cache with its own data. Modern caches are very large, consisting of several Mbytes of data, and a process can spend a significant portion of its quantum just to reload its working set into the processor's cache. In doing so, a process creates an affinity for a processor, but that affinity is destroyed if the process is moved to another processor. Preserving affinity by always running a process on the same processor improves cache hit rates, but may lead to load imbalance. That is, if processes are statically assigned to processors and *never* migrate, as jobs enter and leave the system, some processors may become overloaded while others become idle.

A hierarchical structure of ready queues can be used to remedy this form of load imbalance while preserving affinity. In such a scheme, each processor has its own ready queue, but there is also a single system ready queue where newly created processes are placed. Each processor takes processes to execute from its own ready queue. When those processes complete and the queue empties, the processor takes processes from the single system ready queue and places them in its own ready queue. In the rather rare case where the single system queue is empty as well, a processor takes processes from another processor (destroying affinity, but utilizing what would otherwise be an idle processor). Simulation results show that affinity-preserving policies such as this outperform other scheduling policies in shared-memory multiprocessor systems (14).

Space-sharing policies are a general class of scheduling policy that preserve affinity. In this class of policy, processors are allocated among the applications so that no two applications share a processor. The operating system is responsible for processor allocation; the compiler and run-time system are responsible for scheduling the application's processes (or threads) on those processors. The allocation of processors to applications may be static, semistatic (reevaluated when a new application arrives or an application completes execution), or fully dynamic (reevaluated as an application's parallelism changes).

Static policies are the easiest to implement. Their main advantage is that they provide applications with a dependable computing environment that is guaranteed not to change for the duration of the execution. Given such a guarantee, applications can optimize the granularity of parallelism, communication pattern, and synchronization behavior to match the set of processors that are available. Unfortunately, static policies tend to underutilize the machine since (for example) any application that completes its execution frees up processors, but those processors cannot be assigned to any other currently executing application. To improve system utilization, semi-static space-sharing scheduling policies have been proposed. These policies re-evaluate the allocation of processors to applications when an application arrives or departs. When a new application arrives, the scheduler must take processors from the other applications in order to run the new application; when an application completes, its processors are re-distributed to the other running applications.

Crovella et al. (7) implemented a semi-dynamic space-sharing policy on a BBN Butterfly Plus parallel processor and compared it to coscheduling and naive time-sharing. Their results suggest that semi-static space-sharing outperforms all

time-sharing policies, including coscheduling. One major advantage of space-sharing stems from the fact that parallel applications have sublinear speedup. As a consequence, an application can use a small number of dedicated processors more efficiently than a larger number of shared processors. Given a 100-node machine and two applications, it is better to give each application 50 dedicated nodes for as long as is necessary, rather than require the two applications to alternate their use of all 100 nodes. In the latter case, the overhead of communication and synchronization on 100 nodes does not offset the benefits of having 100 nodes available half the time.

Having observed this same effect, Tucker and Gupta (15) argued that run-time systems should cooperate with the operating system to make sure that no application employs more processes than processors. In their scheme, the operating system informs the run-time system of the number of processors available to an application; the run-time system adjusts the parallelism of the application to match the number of processors available. This adjustment can be done easily for parallel applications that use their task model, wherein the work of an application is broken into small tasks and placed in a central task queue. Each processor executes the next available task from the queue until there are no more tasks. Adding processors to such an application is easy: the processor starts executing the next available task. Removing a processor from such an application is also easy: once the (short) task that the processor executes completes, the processor can be assigned to a different application. This scheduling method was implemented on an Encore shared-memory multiprocessor and was shown to outperform other time-sharing approaches (15).

Space-sharing was also implemented in the Mach operating system (16). In this implementation, each application is guaranteed a number of processors for a long period of time (several minutes). After that interval, an application must be prepared to give the processors back, or else it will be scheduled in a processor pool along with several other applications.

Although better than static scheduling, semi-static scheduling may still underutilize the multiprocessor, especially when applications have varying levels of parallelism. In such cases, allocating a fixed number of processors to each application results in significant load imbalance because applications whose parallelism decreases over time will underutilize their allocation, while applications whose parallelism increases over time would need additional processors. Thus, if applications have a varying degree of parallelism, dynamic space-sharing policies should be employed (17).

Unfortunately, semistatic and dynamic space sharing place a significant burden on the application programmer. In such environments, applications must be written so as to cope with a varying number of processors. Although applications based on a central task queue can easily adapt to a change in the number of processors, many applications create a fixed amount of parellelism (defined at execution start-up time) and never change it. If an application receives fewer processors than it expects, it will incur significant overhead. For this reason, very few multiprocessors today use dynamic or semi-static space sharing.

In any space-sharing policy, we must decide how many processors will be assigned to each application. The *equipartition* policy assigns the same number of processors to each application. This simple policy is used by most space-sharing systems; however, it may result in underutilization of the processors since some applications may not have enough parallelism to exploit their assigned partition. More elaborate allocation methods require that some information about the performance characteristics of the parallel application be made available. Such performance characteristics include the speedup of the application (as a function of the number of processors it is given) and the maximum degree of parallelism in the application. If no other information is known, an application should be given a number of processors equal to its average parallelism, ensuring that both its speedup and its efficiency will be reasonable (at most a factor of two away from the optimal) (18).

If the range of parallelism (maximum−minimum) is known, it can be used in conjunction with average parallelism to improve the allocation policy. When the load is low, each application should receive a number of processors equal to its maximum parallelism; when the load is high, each application should receive a number of processors equal to its minimum parallelism (19). If the amount of work of each application is also known, then an optimal processor partitioning allocates processors in proportion to the square root of the amount of work each application executes. If more application characteristics are known (e.g., the variance of parallelism), then even better allocation heuristics can be used (20).

The performance advantages of most of these heuristics have been demonstrated through analytical evaluation and simulation. However, several studies agree that these heuristics are not robust. That is, if the application and the system characteristics are only *approximately* known, then the scheduling policy may perform even worse than policies that do not take application characteristics into consideration (21).

Discussion

Many different scheduling algorithms have been implemented on multiprocessors, while an even larger number of algorithms have been simulated or analyzed theoretically. Some of these algorithms have contradictory goals. The need for a wide variety of algorithms and the rationale for (seemingly) contradictory approaches lies in the assumptions made about parallel applications. The following categories of parallel applications have been defined (22):

The number of processors used by a {/em rigid} application is determined at compile time.

The number of processors used by a {/em moldable} application is determined at load time. Once the application begins execution, it cannot vary the number of processors in use.

An {/em evolving} application can change the number of processors it uses during execution time at predefined times, usually the beginning of a new computation phase.

Evolving applications can adapt to the number of available processors but only at the start of a phase.

A {/em malleable} application can change the number of processors it uses anytime during its execution.

Different scheduling policies are needed for different categories of applications. For example, the process control policy

described in Ref. (15) is based on the assumption that applications are practically malleable; that is, they can increase or decrease the number of processors they use instantly (or within a very short amount of time). For those applications, process control has been demonstrated to be the best scheduling policy. On the other hand, parallel applications based on a message-passing programming model like PVM or MPI are moldable; once they start execution, they do not change the number of processors in use, since such a change would require data reallocation, which is an expensive operation. For these applications a static or semidynamic space-sharing policy like the one proposed in Ref. (16) usually results in best performance.

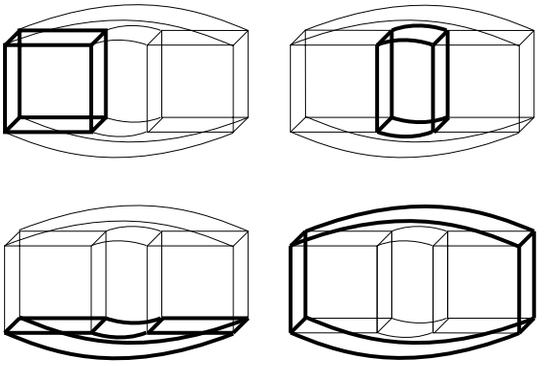

Figure 1. A hypercube of dimension 4 contains several hypercubes of dimension 3, some of which are highlighted in the figure. It is very difficult to recognize small subcubes within a larger cube.

MULTIPROGRAMMING DISTRIBUTED-MEMORY MULTIPROCESSORS

We use the term *distributed-memory multiprocessor* or *multicomputer* to describe loosely-coupled multiprocessors, like the Intel Paragon and the CM-5 from Thinking Machines Inc. Because the processors are only loosely coupled, multicomputers can easily scale to hundreds or even thousands of processors. In these machines, each processor typically executes its own copy of the operating system and manages its own local resources, such as memory and disks. Access to non-local resources requires cooperation between processors, which is implemented via message-passing software.

Multiprogramming of applications on multicomputers is usually based on static space sharing. That is, each application receives a dedicated set of processors for the duration of its execution, or least for a relatively long time interval. Time sharing was avoided in most early multicomputer systems because they employed low-level operating systems that did not allow multiple processes per processor. Even now, multicomputers often have many more processors than any single application can effectively use; thus, there is no point in requiring applications to share processors when there are idle processors available for allocation.

Since multicomputers scale to thousands of processors, applications can usually be allocated as many processors as needed. In order to use such a large number of processors effectively, programs designed to run on multicomputers try to exploit the underlying architecture (including the communication network) as much as possible. For example, processes that communicate frequently may be scheduled for execution on neighboring processors, so as to minimize the latency of communication between them. Applications may even structure their communication pattern to match the underlying architecture in order to avoid long communication delays. Thus, scheduling algorithms for multicomputers have focused on sophisticated partitioning policies that give each application some number of processors arranged in a specific communication topology.

Processor Allocation in Hypercubes

The most popular architecture for multicomputers has been the hypercube. A hyercube (or cube) of dimension N is a set of 2^N processors, where two processors are connected if their binary representations differ in exactly one bit. A 2-D cube is a square; a 3-D cube is a regular cube. Figure 1 shows a 4-D hypercube and highlights some of its 3-D subcubes.

Assuming that message latency increases with the length of the path through the hypercube, two processes that communicate frequently should be allocated processors that are directly connected within the cube. If an application is allocated a random set of processors, it may incur unnecessarily high communication overhead. For this reason, most schedulers for hypercube multicomputers allocate, manage, and free hypercubes of various sizes. The processor allocation problem in a hypercube can be defined as follows:

Definition. There are m users in a p processor hypercube system. Each of the m users requests a subcube of dimensions $k_1, k_2, \ldots, k_m$. The scheduler is to allocate these subcubes and return to each user the address of a subcube within the larger hypercube.

This scheduling problem is similar to the memory management problem where users request memory, and the operating system must satisfy their requests. Memory allocation is a one-dimension problem, however, since memory is simply a linear sequence of bytes. Subcube allocation is a multi-dimensional problem, and in large systems, it can be very difficult to identify and manage all of the subcubes. For example, Fig. 1 shows a 4-D cube and four of the 3-D subcubes it contains (drawn in bold lines). As can be seen in this figure, most of the 3-D subcubes cannot be easily recognized when looking at the 4-D cube. The problem is even more complex in the general case, since a hypercube of dimension N has

$$\binom{N}{k} 2^{N-k}$$

subcubes of dimension k.

Since there are an exponential number of subcubes contained within the larger cube, an optimal algorithm to recognize and manage the subcubes takes exponential time. To avoid the run-time overhead of an exponential algorithm, practical implementations resort to heuristics for approximate solutions. One approach uses a data structure called the

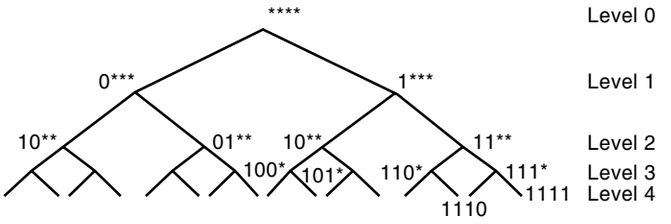

Figure 2. The buddy tree for a 4-D cube. At level i, 2^i subcubes of dimension $4 - i$ can be recognized.

buddy tree. (Figure 2 shows the buddy tree for a 4-D cube.) The subcube allocation algorithm based on the buddy tree works as follows:

- To see if a free subcube of dimension i exists, go to level i and look at all the nodes in that level. If any node is marked `free`, mark it (and all its descendants) as `allocated` and return its address. Otherwise, return failure.
- When an application completes its execution, release its cube by marking the corresponding node in the buddy tree (and all its descendants) as `free`.

This buddy allocation algorithm is simple and fast, but it also fails in some cases to allocate subcubes that are available. For example, if a 3-D cube is needed, the algorithm will only examine the cubes `0***` and `1***`. If one of them is free, the algorithm will allocate it. However, there are several other 3-D subcubes (including `***0`, `***1`, `*0**`, and `**0*`) that are not examined by the algorithm, even though they may be free.

Several extensions to this basic algorithm have been proposed to overcome the limitations of the buddy tree (23–25). Most of these extensions extend the buddy tree data structure into more than one dimension, so as to be able to recognize more subcubes.

Processor Allocation in Mesh-Connected Computers

Another popular architecture for multicomputers is the mesh: a grid of processors, where each processor is connected only to its neighbors. Meshes can be two-dimensional (where each processor is connected to four neighbors) or three-dimensional (where each processor is connected to six neighbors). Much like in the hypercube, programs designed to run on mesh-connected multicomputers exploit the mesh connectivity and adapt their communication patterns to exploit the underlying interconnection network. If an application written for a mesh-connected multicomputer is allocated a non-mesh partition (e.g., a non-rectangular area), it will suffer unnecessary communication overhead and may also affect the execution of other neighboring applications. For this reason, schedulers for mesh-connected computers attempt to allocate rectangular (or cubic) areas to applications. Partitioning the mesh into rectangular (or cubic) areas in order to satisfy user requests is an NP-complete optimization problem (26), so once again, practical algorithms need to employ heuristics.

One of the most popular heuristics for submesh allocation is the 2-D version of the buddy allocation scheme. However, since buddy always manages quantities that are powers of two, 2-D submesh allocation is limited to square submeshes, where the side is a power of two (27).

Another approach is to use the buddy system, to keep track of the allocated submatrices, but not to allocate square submeshes (where the side is a power of two) to incoming requests. Instead, each incoming request may be allocated more than one square submesh (28). Unfortunately, this allocation system may result in noncontiguous submeshes being allocated to a single request. Although allocating noncontiguous submeshes to a single request seems to contradict the original goal of mesh allocation, sometimes noncontiguous submeshes are unavoidable in order to isolate faulty processors within a mesh (29). Other algorithms for submesh allocation include the interval set algorithm (30) and the coverage set (31).

MULTIPROGRAMMING NETWORKS OF WORKSTATIONS

As large-scale multicomputers and multiprocessors became prohibitively expensive, networks of workstations (NOWs) emerged as a viable solution to the need for more computing cycles (1). The network-of-workstations architecture is simply a set of commodity workstations connected via a high-speed interconnection network. In contrast to multiprocessor families that employed custom-made interconnection networks and processors (introducing a lag time of 1 to 2 years in the product development cycle), the NOW architecture employs low-cost, high-performance commodity components, which makes them more than competitive (both in terms of performance and cost) with other parallel computer architectures. NOWs also have the advantage that they can be easily upgraded, often adding linear performance improvements for a linear increase in cost.

To the extent that the processors in a NOW architecture are truly workstations, they may be used to execute parallel applications, sequential applications (e.g., word processing, compiling), and distributed applications (e.g., e-mail, www, file systems). Multiprogramming the processors in a NOW among all these different applications is a challenging task. The responsibility for this task is usually delegated to job scheduling and placement software which makes every effort to meet the competing goals of all scheduled applications.

Condor is one of the first load balancing systems implemented for a network of workstations (32). Its main objective is to take advantage of idle workstations. Each workstation is assumed to have an owner. When the workstation is idle (because the owner is not currently using it), processes belonging to other applications in the system are allowed to use it. Condor allocates the idle processors to processes in its job mix. When the owner returns and starts using the workstation, Condor suspends its processes on the workstation and migrates them to another idle workstation. Suspension and restart are implemented using checkpoints; when a process must be migrated, it is restarted from the last checkpoint. Because of the expense of process management and migration, Condor is most effective for coarse-grain CPU-intensive parallel applications, such as large-scale simulations. The success of Condor lead to the development of commercial job scheduling products, like LSF (Load Sharing Facility; Platform Computing, Canada) (33) and CODINE (Computing in Distributed Networked Environments; GENIAS, GmbH, Germany) (34).

Condor and similar tools were initially designed to support the transparent, non-local execution of *sequential* applications. With the evolution of parallel processing, however, scheduling and load balancing tools for NOWs are presented with a mixture of parallel and sequential applications, often with differing needs. Sequential applications need fast response, while parallel applications need computing cycles and support for efficient communication and synchronization.

The needs of parallel and sequential applications are accommodated best when they are given separate partitions of a NOW, so that no sequential application shares a processor with a parallel application. Recent research results suggest that in an environment with 60 workstations, 32 of them are idle most of the time (35). These idle workstations could be allocated to the execution of parallel applications. Thus, parallel applications could be confined to their own partition, where all the previously described scheduling algorithms could be used. Unfortunately, it is not always the *same* 32 processors that are idle at the same time. Experimental observations suggest that when a workstation goes from idle to active, another idle workstation is likely to be available within the same NOW. Under these conditions, parallel applications run in the idle machines that form a separate partition. When a machine becomes idle, it joins the partition for parallel applications. When a machine goes from idle to active, a parallel process is migrated to another (preferably idle) machine. Parallel applications within a partition may use a standard UNIX scheduler, or coscheduling if they synchronize very frequently (36–38).

Process Migration

Multiprogramming a NOW among several applications usually involves process migration among participating workstations. Process migration requires significant changes to the operating system and may incur large run-time overhead. Despite these drawbacks, several operating systems, including Sprite (39), MOSIX (40), LOCUS (41), the V system (42,43), Accent (44), DEMOS/MP (45), Charoltte (46), and AMOEBA (47), have experimentally implemented process migration.

There are several difficulties that arise in implementing process migration, such as virtual memory copy, migration of open files, and migration of communication channels. The most expensive aspect of migration is usually the transfer of the virtual memory of the migrating process (39). Charlotte and LOCUS transfer the process's entire address space to the destination machine at migration time. Although simple, this method introduces unnecessary overhead if the process will not access its entire address space during the remainder of its execution. Moreover, since the transfer of an address space (several tens of Mbytes) may take several seconds (even over a Fast Ethernet or an ATM interconnection network), the process will be idle for a long period of time. To avoid this idle time, the V system allows a process to continue execution on the workstations it was using, while its address space is transferred to the new workstation. After the whole address space is transferred, the process is suspended, the pages that have been modified since the transfer started are transferred again, and the process is restarted at the new workstation.

To avoid the overhead of copying the *entire* address space of the migrated process, Accent uses a copy-on-reference approach. When a process is migrated, its pages are not copied; only the information needed to create a new process control block on the designation workstation is transferred. When the process starts running on the new workstation and begins accessing its memory, it will page fault and copy only the pages it needs. Sprite uses a similar method: when a process is migrated, all its dirty pages are flushed to the disk where its swap space is stored. When the process is restarted on the new workstation, it will cause a page fault and bring from the file server (one page at a time) all the pages it needs.

In addition to the runtime overhead of copying address spaces, migration involves substantial changes to traditional operating systems, since most operating systems are not designed to support migration. Some of the problems relate to naming conventions used by current operating systems. For example, each process has a name (process ID) that is a small integer, which is valid only within the workstation on which it was created. If the process migrates to another processor, then its name may have already been allocated to another process, and thus there will be two processes with the same name on the same machine. Similar naming problems exist with process control blocks, open files, and communication ports.

To accommodate migration, several operating system calls may have to be rewritten to use the state of migrated processes. To avoid major kernel modifications, some systems choose to forward the location-dependent system calls of the migrated processes to the workstation where the process was originally created. In message-based systems like V and DEMOS/MP, system call forwarding is easy because a system call is nothing more than a message sent from the user process to the kernel. When the process moves on a new workstation, its messages are just forwarded to the appropriate kernel.

SUMMARY

Multiprocessors represent an expensive investment in hardware than must be amortized over a relatively short period of time. Multiprogramming is one way to most efficiently utilize these machines. In this paper, we described several different scheduling policies for multiprogramming a multiprocessor among competing applications in a fair and efficient way. Although each of the policies we described has particular advantages within a certain domain of applications, the space-sharing policies are the simplest and most effective overall. Space-sharing policies employ two-level scheduling. At one level, the operating system dedicates a number of processors to each application for a reasonably long period of time. At another level, the user software deals with scheduling threads (or processes) on top of its dedicated processors so as to preserve synchronization, communication, and load balancing constraints.

ACKNOWLEDGMENTS

This work was supported by GSRT through the PENED project 2041 2270/1-2-95, and the National Science Foundation (grants no. CDA-9401142 and CCR-9510173). The authors gratefully acknowledge this support.

BIBLIOGRAPHY

1. T. E. Anderson, D. E. Culler, and D. A. Patterson, A case for NOW (Networks of Workstations). *IEEE Microw.,* **15** (1): 54–64, 1995.
2. S. C. Leutenegger and M. K. Vernon, The performance of multiprogrammed multiprocessor scheduling policies, *Proc. 1990 ACM SIGMETRICS Conf. Meas. Model. Comput. Syst.,* pp. 226–236, 1990.
3. J. Zahorjan, E. D. Lazowska, and D. L. Eager, The effect of scheduling discipline on spin overhead in shared memory parallel systems, *IEEE Trans. Parallel Distrib. Process.,* **2** (2): 180–198, 1991.
4. J. K. Ousterhout, D. A. Scelza, and P. S. Sindu, Medusa—an experiment in distributed operating system structure, *Commun. ACM,* **23**: 92–105, 1980.
5. D. G. Feitelson and L. Rudolph, Distributed hierarchical control for parallel processing, *IEEE Comput.,* **23** (5): 65–77, 1990.
6. P. G. Sobalvarro and W. E. Weihl, Daemon-based coscheduling of parallel jobs on multiprogrammed multiprocessors, in *Job Scheduling Strategies for Parallel Processing,* Berlin: Springer-Verlag, 1995, *Lect. Notes Comput. Sci.,* **949**: 106–126.
7. M. Crovella et al., Multiprogramming on multiprocessors, *Proc. 3rd Symp. Parallel Distrib. Process.,* pp. 590–597, 1991
8. M. L. Scott et al., Implementation issues for the psyche multiprocessor operating system, *Comput. Syst.,* **3** (1): 101–137, 1990.
9. J. M. Mellor-Crummey and M. L. Scott, Algorithms for scalable synchronization on shared-memory multiprocessors, *ACM Trans. Comput. Syst.,* **9** (1): 21–65, 1991.
10. B. D. Marsh et al., First-class user-level threads, *Proc. 13th Symp. Oper. Syst. Prin.,* Pacific Grove, CA, pp. 110–121, 1991.
11. J. Edler, J. Lipkis, and E. Schonberg, Process management for highly parallel UNIX systems, *Proc. USENIX Workshop Unix Supercomput.,* Pittsburgh, PA, 1988; available as Ultracomputer Note 136, COURANT, 1988.
12. T. E. Anderson et al., Scheduler activations: Effective kernel support for the user-level management of parallelism, *ACM Trans. Comput. Syst.,* **10** (1): 53–79, 1992.
13. L. J. Kontothanassis, R. W. Wisniewski, and M. L. Scott, Scheduler-conscious synchronization. *ACM Trans. Comput. Syst.* **15** (1): 3–40, 1997.
14. M. S. Squillante and E. D. Lazowska, Using processor-cache affinity information in shared-memory multiprocessor scheduling, *IEEE Trans. Parallel Distrib. Process.,* **4** (2): 131–143, 1993.
15. A. Tucker and A. Gupta, Process control and scheduling issues for multiprogrammed shared-memory multiprocessors, *Proc. 12th Symp. Oper. Syst. Prin.,* pp. 159–166, 1989.
16. D. L. Black, Scheduling support for concurrency and parallelism in the mach operating system, *IEEE Comput.,* **23** (5): 35–43, 1990.
17. C. McCann, R. Vaswani, and J. Zahorjan, A dynamic processor allocation policy for multiprogrammed shared memory multiprocessors, *ACM Trans. Comput. Syst.,* **11** (2): 1993.
18. D. L. Eager, J. Zahorjan, and E. D. Lazowska, Speedup versus efficiency in parallel systems, *IEEE Comput.,* **38** (3): 408–423, 1989.
19. K. C. Sevcik, Characterizations of parallelism in applications and their use in scheduling, *Proc. 1989 ACM SIGMETRICS Conf. Meas. Model. Comput. Syst.,* pp. 171–180, 1989.
20. K. C. Sevcik, Application scheduling and processor allocation in multiprogrammed multiprocessors, *Perform. Eval.,* **9** (2-3): 107–140, 1994.
21. E. Rosti et al., Robust partitioning policies for multiprocessor systems, *Perform. Eval.,* **19** (2-3): 141–165, 1994.
22. D. G. Feitelson et al., Theory and practice in parallel job scheduling, in *Job Scheduling Strategies for Parallel Processing,* Berlin: Springer-Verlag, 1997, pp. 1–34.
23. F. Ercal, J. Ramanujam, and P. Sadayappan, Task allocation onto a hypercube by recursive mincut bipartitioning, *J. Parallel Distrib. Comput.,* **10**: 35–44, 1990.
24. M. S. Chen and K. G. Shin, Processor allocation in an n-cube multiprocessor using gray codes, *IEEE Trans. Comput.,* **C-36** (12): 1396–1407, 1987.
25. S. A. Bassam et al., Processor allocation for hypercubes, *J. Parallel Distrib. Comput.,* **16**: 394–401, 1992.
26. K. Li and K. H. Cheng, Job scheduling in a partitionable mesh using a two-dimensional buddy system partitioning scheme, *IEEE Trans. Parallel Distrib. Process.,* **2** (4): 413–422, 1991.
27. K. Li and K.-H. Cheng, A two-dimensional buddy system for dynamic resource allocation in partitionable mesh connected systems. *J. Parallel Distrib. Comput.,* **12** (1): 79–83, 1991.
28. M. Wan et al., A batch scheduler for the Intel Paragon with a non-contiguous node allocation algorithm, in *Job Scheduling Algorithms for Parallel Processing,* Berlin: Springer-Verlag, 1996, pp. 48–64.
29. P. Mohapatra and C. R. Das, On dependability evaluation of mesh-connected processors, *IEEE Trans. Comput.,* **44** (9): 1073–1084.
30. C. Morgenstern, Methods for precise submesh allocation, *Sci. Comput.,* **3**(4): 353–364, 1994.
31. Y. Zhu, Efficient processor allocation strategies for mesh-connected parallel computers, *J. Parallel Distrib. Comput.,* **16** (4): 328–337, 1992.
32. M. J. Litzkow, M. Livny, and M. W. Mutka, Condor—a hunter of idle workstations, *Proc. 8th Int. Conf. Distrib. Comput. Syst.,* 1988.
33. Platform Computing, LSF User's and Administrator's Guides.
34. GENIAS Software GmbH, http://www.genias.de/, 1996.
35. R. H. Arpaci et al., The interaction of parallel and sequential workloads on a network of workstations, *Proc. 1995 ACM SIGMETRICS Conf. Meas. Model. Comput. Syst.,* Ottawa, Ontario, Canada, 1995.
36. A. C. Dusseau, R. H. Arpaci, and D. E. Culler, Effective distributed scheduling of parallel workloads, *Proc. 1996 ACM SIGMETRICS Conf. Meas. Model. Comput. Syst.,* pp. 25–36, 1996.
37. X. Du and X. Zhang, *Coordinating Parallel Processes on Networks of Workstations,* Tech. Rep., San Antonio: High Performance Computing and Software Lab, University of Texas, 1996.
38. X. Zhang and X. Du, Coordinating parallel processes on networks of workstations, *J. Parallel Distrib. Comput.,* **46** (2): 1997.
39. F. Douglis and J. Ousterhout, Transparent process migration: Design alternatives and the sprite implementation, *Softw. Pract. Exp.,* November, 1989.
40. A. Barak, O. Laden, and A. Braveman, The NOW MOSIX and its preemptive process migration scheme, *Bull. IEEE Tech. Comm. Oper. Syst. Appl. Environ.,* **7** (2): 5–11, 1995.
41. G. J. Popek and B. J. Walker, *The LOCUS Distributed System,* Cambridge, MA: MIT Press, 1985.
42. M. Theimer, K. Lantz, and D. Cheriton, Preemptable remote execution facilities for the v-system, *Proc. 11th ACM Symp. Oper. Syst. Prin.,* pp. 13–22, 1987.
43. M. Theimer, Preemtable remote execution facilities for loosely-coupled distributed systems, Ph.D. thesis, Stanford University, Stanford, CA, 1986.
44. E. R. Zayas, Attacking the process migration bottleneck, *Proc. 11th Symp. Oper. Syst. Prin.,* pp. 13–24, 1987.
45. M. L. Powell and B. P. Miller, Process migration in DEMOS/MP, *Proc. 9th Symp. Oper. Syst. Prin.,* pp. 110–119, 1983.

46. Y. Artsy and R. Finkel, Designing a process migration facility: The Charlotte experience, *IEEE Comput.,* **22** (9): 47–56, 1989.
47. C. Steketee, W. P. Zhu, and P. Moseley, Implementation of process migration in amoeba, *Proc. 14th Int. Conf. Distrib. Comput. Syst.,* Poznan, Poland, 1994.

EVANGELOS P. MARKATOS
Foundation for Research and Technology Hellas
University of Crete

THOMAS J. LEBLANC
University of Rochester