Syracuse University
SUrface

Electrical Engineering and Computer Science | L.C. Smith College of Engineering and Computer Science

1-1-2002

Distributed and Multiprocessor Scheduling

Steven J. Chapin
Syracuse University, Department of Electrical Engineering and Computer Science

Jon B, Weissman
University of Minnesota

Follow this and additional works at: http://surface.syr.edu/eecs

Part of the Computer Sciences Commons

Recommended Citation
Chapin, Steven J. and Weissman, Jon B,, "Distributed and Multiprocessor Scheduling" (2002). *Electrical Engineering and Computer Science.* Paper 40.
http://surface.syr.edu/eecs/40

This Working Paper is brought to you for free and open access by the L.C. Smith College of Engineering and Computer Science at SUrface. It has been accepted for inclusion in Electrical Engineering and Computer Science by an authorized administrator of SUrface. For more information, please contact surface@syr.edu.

DISTRIBUTED AND MULTIPROCESSOR SCHEDULING

Steve J. Chapin, Syracuse University
Jon B. Weissman, University of Minnesota

Introduction

This chapter discusses CPU scheduling in parallel and distributed systems. CPU scheduling is part of a broader class of resource allocation problems, and is probably the most carefully studied such problem. The main motivation for multiprocessor scheduling is the desire for increased speed in the execution of a workload. Parts of the workload, called tasks, can be spread across several processors and thus be executed more quickly than on a single processor. In this chapter, we will examine techniques for providing this facility.

The scheduling problem for multiprocessor systems can be generally stated as "How can we execute a set of tasks T on a set of processors P subject to some set of optimizing criteria C?" The most common goal of scheduling is to minimize the expected runtime of a task set. Examples of other scheduling criteria include minimizing the cost, minimizing communication delay, giving priority to certain users' processes, or needs for specialized hardware devices. The scheduling policy for a multiprocessor system usually embodies a mixture of several of these criteria.

Section 2 outlines general issues in multiprocessor scheduling and gives background material, including issues specific to either parallel or distributed scheduling. Section 3 describes the best practices from prior work in the area, including a broad survey of existing scheduling algorithms and mechanisms. Section 4 outlines research issues and gives a summary. Section 5 lists the terms defined in this chapter, while sections 6 and 7 give references to important research publications in the area.

Issues in Multiprocessor Scheduling

There are several issues that arise when considering scheduling for multiprocessor systems. First, we must distinguish between **policy** and **mechanism**. Mechanism gives us the ability to perform an action; policy decides what we do with the mechanism. Most automobiles have the power to travel at speeds of over 150 kilometers per hour (the mechanism), but legal speed limits are usually set well below that (the policy). We will see examples of both scheduling mechanisms and scheduling policies.

Next, we will distinguish between **distributed** and **parallel** systems. Past distinctions have been based on whether an interrupt is required to access some portion of memory; in other words, whether communication between processors is via shared memory (also known as **tight coupling**) or via message passing (also known as **loose coupling**). Unfortunately, while this categorization applies well to systems such as shared-memory symmetric multiprocessors (obviously parallel), and networks of workstations (obviously distributed), it breaks down for message-passing multiprocessors such as hypercubes. By common understanding, the hypercube is a parallel machine, but by the memory test, it is a distributed system.

The true test of whether a system is parallel or distributed is the support for **autonomy** of the individual nodes. Distributed systems support autonomy, while parallel systems do not. A node is autonomous if it is free to behave differently than other nodes within the system.[1] By this test, a hypercube is classified as a parallel machine. There are four components to the autonomy of a multiprocessor system: **design autonomy**, **communication autonomy**, **execution autonomy**, and **administrative autonomy**.

Design autonomy frees the designers of individual systems from being bound by other architectures; they can design their hardware and software to their own specifications and needs. Design autonomy gives rise to heterogeneous systems, both at the level of the operating system software and at the underlying hardware level. Communication autonomy allows each node to choose what information to send, and when to send it. Execution autonomy permits each processor to decide whether it will honor a request to execute a task. Furthermore, the processor and has the right to stop executing a task it had previously accepted. With administrative autonomy, each system sets its own resource allocation policies, independent of the policies of other systems. The local policy decides what resources are to be shared. In effect, execution autonomy allows each processor to have a local scheduling policy; administrative autonomy allows that policy to be different from other processors within the system.

A **task** is the unit of computation in our computing systems, and several tasks working towards a common goal are called a **job**. There are two levels of scheduling in a multiprocessor system: **global** scheduling and **local** scheduling [Casavant and Kuhl, 1988]. Global scheduling involves assigning a task to a particular processor within the system. This is also known as **mapping**, **task placement**, and **matching**. Local scheduling determines which of the set of available tasks at a processor runs next on that processor.

Global scheduling takes places before local scheduling, although **task migration**, or dynamic reassignment, can change the global mapping by moving a task to a new processor. To migrate a

[1] We speak of behavior at the operating system level, not at the application level.

task, the system freezes the task, saves its state, transfers the saved state to a new processor, and restarts the task. There is substantial overhead involved in migrating a running task.

Given that we have several jobs, each composed of many tasks, competing for CPU service on a fixed set of processors, we have two choices as to how we allocate the tasks to the processors. We can assign several processors to a single job, or we can assign several tasks to a single processor. The former is known as **space sharing**, and the latter is called **time sharing**.

Under space sharing, we usually arrange things so that the job has as many processors as it has tasks. This allows all the tasks to run to completion, without any tasks from competing jobs being run on the processors assigned to this job. In many ways, space sharing is similar to old-fashioned batch processing, applied to multiprocessor systems. Under time sharing, tasks may be periodically preempted to allow other tasks to run. The tasks may be from a single job or from multiple jobs. Generally speaking, space sharing is a function of the global scheduling policy, while timesharing is a function of local scheduling.

One of the main uses for global scheduling is to perform **load sharing** between processors. Load sharing allows busy processors to offload some of their work to less busy, or even idle, processors. **Load balancing** is a special case of load sharing, in which the goal of the global scheduling algorithm is to keep the load even (or balanced) across all processors. **Sender-initiated** load sharing occurs when busy processors try to find idle processors to offload some work. **Receiver-initiated** load sharing occurs when idle processors seek busy processors. It is now accepted wisdom that, while load *sharing* is worthwhile, load *balancing* is generally not worth the extra effort, as the small gain in execution time of the tasks is more than offset by the effort expended in maintaining the balanced load.

A global scheduling policy may be thought of as having four distinct parts: the **transfer policy**, the **selection policy**, the **location policy**, and the **information policy**. The transfer policy decides when a node should migrate a task, and the selection policy decides which task to migrate. The location policy determines a partner node for the task migration, and the information policy determines how node state information is disseminated among the processors in the system. For a complete discussion of these components, see [Singhal and Shivaratri, 1994, ch. 11].

An important feature of the selection policy is whether it restricts the candidate set of tasks to new tasks which have not yet run, or allows the transfer of tasks that have begun execution. **Nonpreemptive** policies only transfer new jobs, while **preemptive** policies will transfer running jobs as well. Preemptive policies have a larger set of candidates for transfer, but the overhead of migrating a job that has begun execution is higher than for a new job because of the accumulated state of the running job (such as open file descriptors, allocated memory, etc.).

As the system runs, new tasks arrive while old tasks complete execution (or, equivalently, are

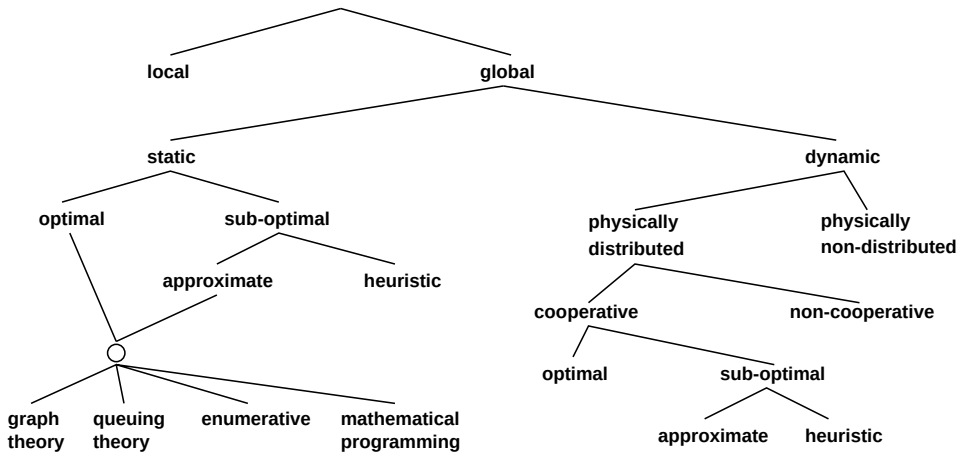

Figure 1: A taxonomy of distributed scheduling algorithms

served). If the arrival rate is greater than the service rate, then the process waiting queues within the system will grow without bound and the system is said to be **unstable**. If, however, tasks are serviced at least as fast as they arrive, the queues in the system will have bounded length and the system is said to be **stable**. If the arrival rate is just slightly less than the service rate for a system, it is possible for the additional overhead of load sharing to push the system into instability. A stable scheduling policy does not have this property, and will never make a stable system unstable.

Distributed Scheduling

In most cases, work in distributed scheduling concentrates on global scheduling because of the architecture of the underlying system. Casavant and Kuhl [Casavant and Kuhl, 1988] defines a taxonomy of task placement algorithms for distributed systems, which we have partially reproduced in figure 1. The two major categories of global algorithms are static and dynamic.

Static algorithms make scheduling decisions based purely on information available at compilation time. For example, the typical input to a static algorithms would include the machine configuration and the number of tasks and estimates of their running time. Dynamic algorithms, on the other hand, take factors into account such as the current load on each processor. Adaptive algorithms are a special subclass of dynamic algorithms, and are important enough that they are often discussed separately. Adaptive algorithms go one step further than dynamic algorithms, in that they may change the policy based on dynamic information. A dynamic load-sharing algorithm might use the current system state information to seek out a lightly-loaded host, while an adaptive algorithm might switch from sender-initiated to receiver-initiated load sharing if the system load

rises above a threshold.

In **physically non-distributed**, or centralized, scheduling policies, a single processor makes all decisions regarding task placement—this has obvious implications for the autonomy of the participating systems. Under **physically distributed** algorithms, the logical authority for the decision-making process is distributed among the processors that constitute the system.

Under **non-cooperative** distributed scheduling policies, individual processors make scheduling choices independent of the choices made by other processors. With **cooperative** scheduling, the processors subordinate local autonomy to the achievement of a common goal.

Both static and cooperative distributed scheduling have **optimal** and **suboptimal** branches. Optimal assignments can be reached if complete information describing the system and the task force is available. Suboptimal algorithms are either **approximate** or **heuristic**. Heuristic algorithms use guiding principles, such as assigning tasks with heavy inter-task communication to the same processor, or placing large jobs first. Approximate solutions use the same computational methods as optimal solutions, but use solutions that are within an acceptable range, according to an algorithm-dependent metric.

Approximate and optimal algorithms employ techniques based on one of four computational approaches: enumeration of all possible solutions, graph theory, mathematical programming, or queuing theory. In the taxonomy, the subtree appearing below optimal and approximate in the static branch is also present under the optimal and approximate nodes on the dynamic branch; it is elided in figure 1 to save space.

In future sections, we will examine several scheduling algorithms from the literature in light of this taxonomy.

Scheduling for Shared-Memory Parallel Systems

Researchers working on shared-memory parallel systems have concentrated on local scheduling, because of the ability to trivially move processes between processors. There are two main causes of artificial delay that can be introduced by local scheduling in these systems: cache corruption and preemption of processes holding locks.

cache corruption As a process runs, the operating system caches several types of information for the process including its working set and recently read file blocks. If this information is not accessed frequently, the operating system will replace it with cache information from other processes.

lock preemption Spin locks, a form of busy waiting, are often used in parallel operating systems when contention for a critical section is expected to be low and the critical section is short.

The problem of lock preemption occurs when a process that holds the lock is preempted on one processor, while another process waiting to enter the lock is running on a different processor. Until the first process runs again and releases the lock, all of the CPU time used by the second process is wasted.

In an upcoming section, we will examine methods to alleviate or avoid these delays.

Best Practices

In this section, we will examine the current state-of-the-art in multiprocessor scheduling. We will first consider the techniques used in parallel systems, and then examine scheduling algorithms for message-passing systems. Finally, we will study scheduling support mechanisms and algorithms for distributed systems, including computational grids.

Parallel Scheduling

We will examine three aspects of scheduling for parallel systems: local scheduling for shared-memory systems such as the SGI Altix family; static analysis tools that are beneficial for producing global schedules for parallel systems; and dynamic scheduling for distributed-memory systems.

Local Scheduling for Parallel Systems

For most shared-memory timesharing systems, there is no explicit global placement: all processors share the same ready queue, and so any task can be run on any processor. In contrast, local scheduling is crucial for these systems, while it is nonexistent in space-sharing systems. We will examine several local scheduling techniques for parallel systems. In general, these techniques are attempting to eliminate one of the causes of delay mentioned earlier. All of these techniques are discussed in chapter 17 of [Singhal and Shivaratri, 1994].

Coscheduling, or gang scheduling, schedules the entire pool of subtasks for a single task simultaneously. This can work well with fine-grained applications where communication dominates computation, so that substantial work can be accomplished in a single time slice. Without coscheduling, it is easy to fall into a pattern where subtasks are run on a processor, only to immediately block waiting for communication. In this way, coscheduling combines aspects of both space sharing and time sharing.

Smart Scheduling tries to avoid the preemption of a task that holds a lock on a critical section. Under smart scheduling, a process sets a flag when it acquires a lock. If a process has its flag set, it

will not be preempted by the operating system. When a process leaves a critical section, it resets its flag.

The Mach operating system uses scheduler hints to inform the system of the expected behavior of a process. Discouragement hints inform the system that the current thread should not run for a while, and hand-off hints are similar to coroutines in that they "hand off" the processor to a specific thread.

Under affinity-based scheduling, a task is said to have an affinity for the processor on which it last ran. If possible, a task is rescheduled to run on the processor for which it has affinity. This can ameliorate the effects of cache corruption. The disadvantage of this scheme is that it diminishes the chances of successfully doing load sharing because of the desire to retain a job on its current processor. In effect, affinity-based scheduling injects a measure of global scheduling into the local scheduling policy.

Static Analysis

There are several systems that perform static analysis on a task set and generate a static task mapping for a particular architecture. Examples of such systems include Parallax, Hypertool, Prep-P, Oregami, and Pyrros (see [Shirazi et al., 1995] for individual papers on these systems). Each of these tools represents the task set as a directed acyclic graph, with the nodes in the graph representing computation steps. Edges in the graph represent data dependencies or communication, where the result of one node is made ready as input for another node. These tools attempt to map the static task graph onto a given machine according to an optimizing criterion (usually, minimal execution time, although other constraints such as minimizing the number of processors used can also be included). The scheduling algorithm then uses some heuristic to generate a near-optimal mapping.

Parallax (Lewis and El-Rewini) is a partitioning and scheduling system that implements seven different heuristic policies. The input to the system is a graph representing the structure of the tasks and a user-selectable representation of the machine architecture. Parallax will then generate schedules based on each of the available heuristics, and present the expected results to the user. This system is unique in its ability to permit the user to explore different combinations of scheduling heuristics and machine architecture for a given task set.

Hypertool (Wu and Gajski) takes as input C source code and generates the task graph representing the program. This is distinct from Parallax, where the user must supply the task graph (and may therefore study the behavior of algorithms which have not been explicitly expressed in any particular programming language). Hypertool then schedules the derived task graph on a hypercube.

Sarkar and Hennessy built one of the first tools to extract parallelism from a functional program, partition the individual tasks into jobs, and then place the jobs on a multiprocessor. They developed a new representation for parallel computation called Macro-Dataflow, and applied their work to programs written in SISAL for the VAX.

Prep-P (Berman and Stramm) is a mapping tool that runs in conjunction with the Poker programming environment for the Pringle machine. Prep-P was one of the earliest program mapping tools, and uses a Graph Description Language as input to describe the program structure. Prep-P uses an iterative partitioning algorithm, wherein an initial partitioning is derived and the system repeatedly attempts to improve upon the partition by moving a task from one partition into another. Whenever the proposed move results in a lower-cost schedule, the move is kept.

Oregami (Lo, et al.) is similar to Prep-P in functionality, with the addition of a new model for representing the computation called the Temporal Communications Graph (TCG). The TCG represents each event (computation, message send, or message receipt) as a node within a directed acyclic graph. Thus, it represents a combination of the static task graph with Lamport's process-time graphs.

In many ways, Pyrros (Yang and Gersoulis) represents a merger between several ideas from earlier static scheduling systems. Pyrros uses Sarkar and Hennessy's Macro-Dataflow model, and is targeted for a hypercube architecture. The system takes a Macro-Dataflow graph as input, then performs partitioning and scheduling. It can produce optimal schedules for several restricted classes of algorithms.

Distributed-Memory Systems

In distributed-memory systems, such as hypercube systems, global scheduling is done. Most hypercube systems use space sharing, in that they reserve subcubes of the larger hypercube for use by a single application. Several algorithms have been proposed for this, including that found in [Huang et al., 1989]. A typical algorithm maintains a binary tree listing the various sizes of hypercubes available in the system. When a request for an m-dimensional cube is made, the scheduling system searches the tree to see if a hypercube of that exact size is available, and if so, allocates it. If no such hypercube is available, the system splits the smallest hypercube of dimension $> m$ into multiple hypercubes, allocates one, and updates the binary tree to reflect the new set of available hypercubes.

For example, consider a request for four processors from a 16-processor hypercube, with all nodes currently free. Four processors comprise a two-dimensional hypercube (a square). The scheduling system would split the 16 processors into two eight-processor cubes, and then split one of the eight-processor cubes into two four-processor squares. One of the squares would be allocated

to the job, leaving one two-dimensional and one three-dimensional hypercube for other jobs.

It is interesting to examine the coexistence of space sharing and time sharing on a single machine. The ASCI Red machine at Sandia National Laboratories is a distributed-memory system that divides its nodes into two partitions: a service partition which runs a general-purpose, full-featured operating system (UNIX), and a compute partition that runs a special-purpose, highly efficient operating system (Cougar). The service partition runs in time-sharing mode, while Cougar provides low-latency communication under a mixed space-sharing and time-sharing paradigm (nodes are dedicated to a single job, but that job can run multiple tasks on a single node). Users launch their jobs from the service partition, and the scheduling system reserves a portion of the compute partition to run the tasks for those jobs.

Distributed Scheduling Algorithms

Many researchers have devised algorithms for task placement in distributed systems. This section categorizes several of these techniques in terms of the taxonomy presented earlier.

Table 1 displays information garnered from a survey of existing scheduling algorithms. For each algorithm, an entry indicates whether the method is distributed or centralized, supports heterogeneity, minimizes overhead, or supports scalability. Entries are either *Y, N*, *P*, or *x*, indicating the answer is yes, no, partially, or not applicable. The remainder of this section contains a brief description of each method, with a discussion of its place in the taxonomy and its salient properties. Interested readers are referred to the cited publications to obtain full details about the algorithms.

Dynamic, Distributed, Cooperative, Suboptimal Algorithms

Blake [Blake, 1992] describes four suboptimal, heuristic algorithms. Under the first algorithm, Non-Scheduling (NS), a task is run where it is submitted. The second algorithm is Random Scheduling (RS), wherein a processor is selected at random and is forced to run a task. The third algorithm is Arrival Balanced Scheduling (ABS), in which the task is assigned to the processor that will complete it first, as estimated by the scheduling host. The fourth method uses receiver-initiated load balancing, and is called End Balanced Scheduling (EBS). NS, RS, and ABS use one-time assignment; EBS uses dynamic reassignment.

Casavant and Kuhl [Casavant and Kuhl, 1984] describes a distributed task execution environment for UNIX System 7, with the primary goal of load balancing without altering the user interface to the operating system. As such, the system combines mechanism and policy. This system supports execution autonomy, but not communication autonomy or administrative autonomy.

Ghafoor and Ahmad [Ghafoor and Ahmad, 1990] describes a bidding system that combines

Table 1: Summary of distributed scheduling survey

Method	Distributed	Heterogeneous	Overhead	Scalable
Blake [Blake, 1992] (NS, RS)	Y	N	Y	Y
(ABS, EBS)	Y	N	N	Y
(CBS)	N	N	N	N
Casavant and Kuhl [Casavant and Kuhl, 1984]	Y	N	x	P
Ghafoor and Ahmad [Ghafoor and Ahmad, 1990]	Y	N	Y	P
Wave Scheduling [Van Tilborg and Wittie, 1984]	Y	N	x	P
Ni and Abani [Ni and Abani, 1981] (LED)	Y	N	x	N
(SQ)	Y	N	Y	Y
Stankovic and Sidhu [Stankovic and Sidhu, 1984]	Y	N	x	P
Stankovic [Stankovic, 1985]	Y	N	x	N
Andrews et al. [Andrews et al., 1982]	Y	x	x	Y
Greedy Load-Sharing [Chowdhury, 1990]	Y	N	X	Y
Gao, et al. [Gao et al., 1984] (BAR)	Y	N	x	N
(BUW)	Y	N	x	N
Stankovic [Stankovic, 1984]	Y	N	x	P
Chou and Abraham [Chou and Abraham, 1983]	Y	N	x	Y
Bryant and Finkel [Bryant and Finkel, 1981]	Y	N	x	Y
Casey [Casey, 1981] (dipstick, bidding)	Y	N	x	N
(adaptive learning)	Y	N	Y	Y
Klappholz and Park [Klappholz and Park, 1984]	Y	N	x	Y
Reif and Spirakis [Reif and Spirakis, 1982]	Y	N	x	N
Ousterhout, et al., see [Singhal and Shivaratri, 1994]	N	N	x	N
Hochbaum and Shmoys [Hochbaum and Shmoys, 1988]	N	Y	x	x
Hsu, et al. [Hsu et al., 1989]	N	Y	x	x
Stone [Stone, 1977]	N	Y	x	x
Lo [Lo, 1988]	N	Y	x	x
Price and Salama [Price and Salama, 1990]	N	Y	x	x
Ramakrishnan et al. [Ramakrishnan et al., 1991]	N	Y	x	x
Sarkar and Hennessy, in [Shirazi et al., 1995]	N	Y	x	x

mechanism and policy. A module called an Information Collector/Dispatcher runs on each node and monitors the local load and that of the node's neighbors. The system passes a task between nodes until either a node accepts the task or the task reaches its transfer limit, in which case the current node accepts the task. This algorithm assumes homogeneous processors and has limited support for execution autonomy.

Van Tilborg and Wittie [Van Tilborg and Wittie, 1984] presents Wave Scheduling for hierarchical virtual machines. The task force is recursively subdivided and the processing flows through the virtual machine like a wave, hence the name. Wave Scheduling combines a non-extensible mechanism with policy, and assumes the processors are homogeneous.

Ni and Abani [Ni and Abani, 1981] presents two dynamic methods for load balancing on systems connected by local area networks: Least Expected Delay and Shortest Queue. Least Expected Delay assigns the task to the host with the smallest expected completion time, as estimated from data describing the task and the processors. Shortest Queue assigns the task to the host with the fewest number of waiting jobs. These two methods are not scalable because they use information broadcasting to ensure complete information at all nodes. [Ni and Abani, 1981] also presents an optimal stochastic strategy using mathematical programming.

The method described in Stankovic and Sidhu [Stankovic and Sidhu, 1984] uses task clusters and distributed groups. Task clusters are sets of tasks with heavy inter-task communication that should be on the same host. Distributed groups also have inter-task communication, but execute faster when spread across separate hosts. This method is a bidding strategy, and uses non-extensible system and task description messages.

Stankovic [Stankovic, 1985] lists two scheduling methods. The first is adaptive with dynamic reassignment, and is based on broadcast messages and stochastic learning automata. This method uses a system of rewards and penalties as a feedback mechanism to tune the policy. The second method uses bidding and one-time assignment in a real-time environment.

Andrews, et al. [Andrews et al., 1982] describes a bidding method with dynamic reassignment based on three types of servers: free, preferred, and retentive. Free server allocation will choose any available server from an identical pool. Preferred server allocation asks for a server with a particular characteristic, but will take any server if none is available with the characteristic. Retentive server allocation asks for particular characteristics, and if no matching server is found, a server, busy or free, must fulfill the request.

Chowdhury [Chowdhury, 1990] describes the Greedy load-sharing algorithm. The Greedy algorithm uses system load to decide where a job should be placed. This algorithm is non-cooperative in the sense that decisions are made for the local good, but it is cooperative because scheduling assignments are always accepted and all systems are working towards a global load balancing policy.

Gao, et al. [Gao et al., 1984] describes two load-balancing algorithms using broadcast information. The first algorithm balances arrival rates, with the assumption that all jobs take the same time. The second algorithm balances unfinished work. Stankovic [Stankovic, 1984] gives three variants of load-balancing algorithms based on point-to-point communication that compare the local load to the load on remote processors. Chou and Abraham [Chou and Abraham, 1983] describes a class of load-redistribution algorithms for processor-failure recovery in distributed systems.

The work presented in Bryant and Finkel [Bryant and Finkel, 1981] combines load balancing, dynamic reassignment, and probabilistic scheduling to ensure stability under task migration. This method uses neighbor-to-neighbor communication and forced acceptance to load balance between pairs of machines.

Casey [Casey, 1981] gives an earlier and less complete version of the Casavant and Kuhl taxonomy, with the term *centralised* replacing *non-distributed* and *decentralised* substituting for *distributed*. This paper also lists three methods for load balancing: Dipstick, Bidding, and Adaptive Learning, then describes a load-balancing system whereby each processor includes a two-byte status update with each message sent. The Dipstick method is the same as the traditional watermark processing found in many operating systems. The Adaptive Learning algorithm uses a feedback mechanism based on the run queue length at each processor.

Dynamic Non-cooperative Algorithms

Klappholz and Park [Klappholz and Park, 1984] describes Deliberate Random Scheduling (DRS) as a probabilistic, one-time assignment method to accomplish load balancing in heavily-loaded systems. Under DRS, when a task is spawned, a processor is randomly selected from the set of ready processors, and the task is assigned to the selected processor. DRS dictates a priority scheme for time-slicing, and is thus a mixture of local and global scheduling. There is no administrative autonomy or execution autonomy with this system, because DRS is intended for parallel machines.

Reif and Spirakis [Reif and Spirakis, 1982] presents a Resource Granting System (RGS) based on probabilities and using broadcast communication. This work assumes the existence of either an underlying handshaking mechanism or of shared variables to negotiate task placement. The use of broadcast communication to keep all resource providers updated with the status of computations in progress limits the scalability of this algorithm.

Dynamic Non-distributed Algorithms

Ousterhout, et al. (see [Singhal and Shivaratri, 1994]) describes Medusa, a distributed operating system for the Cm* multiprocessor. Medusa uses static assignment and centralized decision making,

making it a combined policy and mechanism. It does not support autonomy, nor is the mechanism scalable.

In addition to the four distributed algorithms already mentioned, Blake [Blake, 1992] describes a fifth method called Continual Balanced Scheduling (CBS), that uses a centralized scheduler. Each time a task arrives, CBS generates a mapping within two time quanta of the optimum, and causes tasks to be migrated accordingly. The centralized scheduler limits the scalability of this approach.

Static Algorithms

All the algorithms in this section are static, and as such, are centralized and without support for autonomy. The are generally intended for distributed-memory parallel machines, in which a single user can obtain control of multiple nodes through space sharing. However, they can be implemented on fully distributed systems.

Hochbaum and Shmoys [Hochbaum and Shmoys, 1988] describes a polynomial-time, approximate, enumerative scheduling technique for processors with different processing speeds, called the dual-approximation algorithm. The algorithm solves a relaxed form of the bin packing problem to produce a schedule within a parameterized factor, ϵ, of optimal. That is, the total run time is bounded by $(1 + \epsilon)$ times the optimal run time.

Hsu, et al. [Hsu et al., 1989] describes an approximation technique called the critical sink underestimate method. The task force is represented as a directed acyclic graph, with vertices representing tasks and edges representing execution dependencies. If an edge (α, β) appears in the graph, then α must execute before β. A node with no incoming edges is called a *source*, and a node with no outgoing edges is a *sink*. When the last task represented by a sink finishes, the computation is complete; this last task is called the critical sink. The mapping is derived through an enumerative state space search with pruning, which results in an underestimate of the running time for a partially mapped computation, and hence, the name critical sink underestimate.

Stone [Stone, 1977] describes a method for optimal assignment on a two-processor system based on a Max Flow/Min Cut algorithm for sources and sinks in a weighted directed graph. A maximum flow is one that moves the maximum quantity of goods along the edges from sources to sinks. A minimum cutset for a network is the set of edges with the smallest combined weighting, which, when removed from the graph, disconnects all sources from all sinks. The algorithm relates task assignment to commodity flows in networks, and shows that deriving a Max Flow/Min Cut provides an optimal mapping.

Lo [Lo, 1988] describes a method based on Stone's Max Flow/Min Cut algorithm for scheduling in heterogeneous systems. This method utilizes a set of heuristics to map from a general system representation to a two-processor system so that Stone's work applies.

Price and Salama [Price and Salama, 1990] describes three heuristics for assigning precedence-constrained tasks to a network of identical processors. With the first heuristic, the tasks are sorted in increasing order of communication, and then are iteratively assigned so as to minimize total communication time. The second heuristic creates pairs of tasks that communicate, sorts the pairs in decreasing order of communication, then groups the pairs into clusters. The third method, simulated annealing, starts with a mapping and uses probability-based functions to move towards an optimal mapping.

Ramakrishnan, et al. [Ramakrishnan et al., 1991] presents a refinement of the A* algorithm that can be used either to find optimal mappings or to find approximate mappings. The algorithm uses several heuristics based on the sum of communication costs for a task, the task's estimated mean processing cost, a combination of communication costs and mean processing cost, and the difference between the minimum and maximum processing costs for a task. The algorithm also uses ϵ-relaxation similar to the dual-approximation algorithm of Hochbaum and Shmoys [Hochbaum and Shmoys, 1988].

Sarkar and Hennessy (in [Shirazi et al., 1995]) describes the GR graph representation and static partitioning and scheduling algorithms for single-assignment programs based on the SISAL language. In GR, nodes represent tasks and edges represent communication. The algorithm consists of four steps: cost assignment, graph expansion, internalization, and processor assignment. The cost assignment step estimates the execution cost of nodes within the graph, and communication costs of edges. The graph expansion step expands complex nodes, e.g. loops, to ensure that sufficient parallelism exists in the graph to keep all processors busy. The internalization step performs clustering on the tasks, and the processor assignment phase assigns clusters to processors so as to minimize the parallel execution time.

Wide-Area Distributed Scheduling: the Grid

Recent advances in distributed scheduling algorithms have occurred in a distributed computing environment known as the Grid, an ensemble of shared, geographically-dispersed computers that may not be fully under the control of a single scheduler.

Scheduling in a Computational Grid environment introduces several challenges that go beyond that of traditional distributed system schedulers: heterogeneity, scale, and ownership. In a Grid, resources are highly heterogeneous ranging from supercomputers to PCs, from high-end mass storage devices to low-end disks. Grid resources may number in hundreds or thousands making efficient run-time scheduling more difficult. Finally, in a Grid resources may be under the control of resource owners, putting constraints on how schedulers can use them. The scheduling approaches described below deal with these issues in different ways and to differing degrees.

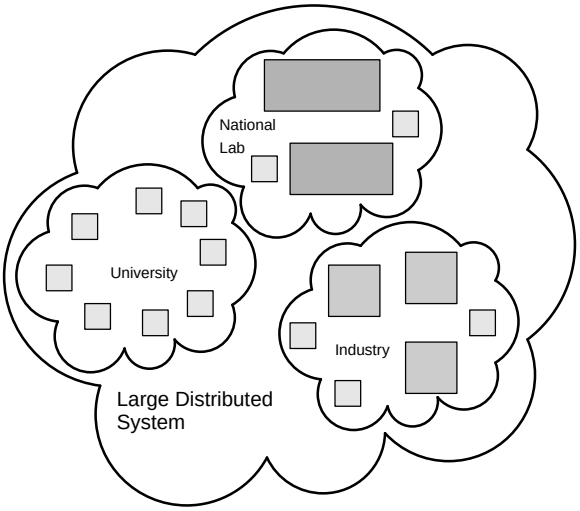

Figure 2: A sample virtual system

MESSIAHS

The first scheduling system for grids, then called metasystems, was the MESSIAHS (**M**echanisms **E**ffecting **S**cheduling **S**upport **I**n **A**utonomous, **H**eterogeneous **S**ystems) system [Chapin and Spafford, 1994]. MESSIAHS provided extensive support for autonomy, using hierarchically structured systems based on administrative domains. This structuring was based on an observation of a social aspect of computing: people are willing to allow outside utilization of their unused resources, as long as they maintain control of the system. This means that the local systems are autonomous, and the local administrators can set their own access policies. An example distributed autonomous system is in figure 2.

Each node within the MESSIAHS system is a **virtual system**, which represents a subset of the resources of one or more real systems, and has a hierarchical structure modeling the administrative hierarchies of computer systems and institutional organization. Virtual systems can be combined into encapsulating virtual systems. For example, in figure 2, the University, National Laboratory, and Industry are each virtual systems, and are collected into a single large distributed (virtual) system. Within the University, National Lab, and Industry virtual systems are other virtual systems, giving a hierarchical structure. These intermediate groupings may correspond to divisions which contain departments, and the departments may contain research groups, etc.. At the lowest level of grouping, each virtual system typically consists of a subset of the capabilities of a single machine. Together, these virtual systems form a directed acyclic graph topology.

There is no centralized resource management in MESSIAHS. Instead, each virtual system runs

a **scheduling module**, which maintains the system description information for that node. The scheduling module also exchanges service requests with neighboring virtual systems within the hierarchy, and is responsible for starting and stopping jobs.

MESSIAHS provides two interfaces with which administrators can define the scheduling policy for a system: an interpreted language called the MESSIAHS Interface Language (MIL) dedicated to implementing schedulers, and a library of functions for the C programming language. Using these interfaces, administrators write small event handlers that are linked into the scheduling system. Thus, MESSIAHS has extensive support for autonomy.

Matchmaking

Matchmaking is a highly distributed scheduling process that accounts for the heterogeneity of resources and the distribution of ownership. This approach consists of a semi-structured data model that combines schema, data, and query in a specification language, and a separation of matching and claiming phases of resource allocation. Both resources and jobs use a mechanism known as classads to specify attributes and constraints (Figure 3). A classad is a mapping from attribute names to expressions. This mechanism allows the heterogeneity and ownership issues inherent to the Grid to be addressed. Issues of scale are not addressed by this mechanism, and the locality of resources is not captured by classads.

Scheduling is accomplished by a matchmaker that matches resource classads to job classads, satisfying the constraints of both. The implementation of the decision is negotiated between the resource and the job using matchmaking and resource claiming protocols. The matching (aka scheduling) algorithm first determines feasible matches (schedules) by determining compatible resource and job classads via the Constraint expression. The next phase of the algorithm is to choose among compatible matches by maximizing the Rank attribute of the job. This scheme has been designed for localized resources within a single Grid site.

AppLeS

The AppLeS scheduling approach is a conceptual framework for the design of Grid schedulers tuned specifically for the application. The goal is to select a potentially efficient configuration of resources based on load and availability, evaluate the potential performance of such configurations based on application-specific performance criteria, and interact with the relevant resource management systems to carry out the decision. AppLeS agents perform the scheduling task for the user and exploit several generic components such as the Network Weather Service [NWS, 1999] which provides resource load predictions. Specific AppLeS scheduling agents have been developed for tightly-coupled

Machine Classad

```
[
Type = "Machine";
Activity = "Idle";
DayTime = 36107;
KeyboardIdle = 1432;
Disk = 323496;
Memory = 64;
State = "Unclaimed";
LoadAvg = 0.042969;
Mips = 104;
Arch = "INTEL";
OpSys = "SOLARIS251";
KFlops = 21893;
Name = "leonardo.cs.wisc.edu";
ResearchGroup = { "raman", "miron",
    "solomon", "jbasney" };
Friends = { "tannenba", "wright" };
Untrusted = { "rival", "riffraff" };
Rank = member(other.Owner,
ResearchGroup)
    *10+member(other.Owner, Friends);
Constraint = !member(other.Owner,
Untrusted)
    && Rank >= 10 ?  true
    :  Rank > 0 ?  LoadAvg<0.3
    && KeyboardIdle>15*60
    :  DayTime < 8*60*60
     DayTime > 18*60*60;
]
```

Job Classad

```
[
Type = "Job";
QDate = 886799469;
    // Submit time secs.  past 1/1/1970
CompletionDate = 0;
Owner = "raman";
Cmd = "run_sim";
WantRemoteSysCalls = 1;
WantCheckpoint = 1;
Iwd = "/usr/raman/sim2";
Args = "-Q 17 3200 10";
Memory = 31;
Rank = KFlops/1E3 + other.Memory/32;
Constraint = other.Type == "Machine"
    && Arch == "INTEL"
    && OpSys == "SOLARIS251"
    && Disk >= 10000
    && other.Memory >= self.Memory;
]
```

Figure 3: Classads describing a Machine and a Job

parallel scientific computations [Berman et. al., 1996], data replica selection [Su et. al., 1999], parameter sweeps [Casanova et. al., 2000], and gene sequence comparison [Spring and Wolski]. These AppLeS agents address the issue of Grid scale by considering a subset of Grid resources and the issue of heterogeneity by dynamic cost modeling via static and dynamic cost functions. An example of an AppLeS scheduling algorithm is shown in Figure 4. The issue of ownership is not addressed. The more recent thrust in this project is to generalize the AppLeS approach to build template schedulers applicable to a wider class of applications. Templates have been constructed for master-slave applications and parameter sweep applications [Casanova et. al., 2000].

Prophet and Gallop

Data parallel applications are a large and important class of parallel scientific applications. Efficiently scheduling these applications in a Grid is difficult due to the potential sharing and hetero-

Resource Selection

```
let locus = machine having the maximum criterion value
let list = a sort of the remaining machines according to
    their logical distance
begin for k = 0 to I − 1
        let S = locus + the first k elements of list
        parameterize C_i and P_i for 1 ≤ i ≤| S |
        with Weather Service forecasts
        solve linear system of equations using this
            parameterization
        if (not all A_i > 0)
          then reject partitioning as infeasible fi
        else if (there exists an A_i that does not fit
                in free memory of processor i)
                then reject partitioning as infeasible fi
        else record expected execution time for subset S
  end
  implement the partitioning corresponding to the minimum
  execution time using the S for which it was computed
```

Performance Model

T_i = time for processor i to compute region i
A_i = the area of region i
P_i = time for processor i to compute a point
C_i = time for processor i to send/recv its borders

$T_i = A_i \times P_i + C_i$
Require $T_1 = T_2 = \ldots = T_I$ s.t. $\sum_{i=1}^{I} A_i = N \times M$
$C_i = f\{\text{Recv } (i \pm 1, i), \text{Send } (i, i \pm 1)\}$
Send/Recv $(i, j) = N \times \text{sizeof(element)} \times \text{Bandwidth}(i, j)$

Figure 4: Jacobi Resource Selection and Performance Model

geneity of resources. In addition, selecting the appropriate number of processors and data domain decomposition are all difficult problems that depend on the available resources, their load and heterogeneity, network load, and problem characteristics. Most systems leave this complex problem up to the end-user. Even if the user figures out the best resource combination by running their application multiple times, this same resource combination may not be available all of the time due to resource sharing, and even if it was, it is unlikely to be best for a different problem instance of the same application. Finally, such optimizations are tailored to the specific application and are not available to another user's application. Prophet [Weissman, 1999] is an automated scheduler for data parallel applications that utilizes a performance model for predicting application performance on different resource combinations that automates processor selection, task placement, and data domain decomposition for data parallel applications. Prophet uses a callback mechanism to allow the run-time scheduler to obtain application-specific information to construct cost functions.

Gallop [Weissman, 1998] is a wide-area Grid scheduling system that provides a federated scheduling model across different sites in a Grid. In the Gallop model, each site can run its own local scheduler, and decide which resources are to be made available to Gallop. Gallop utilizes a bid-based distributed scheduling algorithm to decide which site to select from among a network of sites (See Figure 5.) A multi-site Gallop testbed was constructed in which the local sites each ran a version of Prophet to provide performance prediction estimates for using their resources. Gallop demonstrated that the overhead of remote execution can be tolerated for applications of sufficiently

Local SM component

1. local SM receives WA_sched (Job_class) request
2. local SM chooses k candidate $SM_1 \ldots SM_k$
3. for each SM_i
   call $best_i = SM_i$.get_best(Job_class)
4. the_best = min($best_i$.elapsed_time) over all i in $1 \ldots k$
5. for each SM_i
   if (SM_i == the_best_site)
      result = SM_i.go_SM(Job_class,the_best)
   else
      SM_i.no_go_SM(Job_class,the_best)
6. Return result and scheduling info to front-end

Remote SM Component

1. remote SM receives get_best(Job_class) from local SM
2. call scheds = LS.get_scheds(Job_class)
3. call best = LS.eval_scheds(scheds,Job_class)
4. lock best, return best to local SM
5. if remote SM receives go_SM(Job_class,the_best)
   result = LS.go_ls(Job_class,the_best)
   return result to local SM
   else if remote SM receives no_go_SM(Job_class,the_best)
   store the_best in table
6. release lock on best

Figure 5: Sample Wide-area Scheduling Algorithm

large granularity in the Grid. It achieved performance gains up to 25% over local site execution for a range of applications: a Poisson solver, an image processing pipeline, and a genomic sequence comparison.

Stochastic Scheduling

Stochastic scheduling [Schopf and Berman, 1999] harnesses the variability inherent in Grid computing to produce performance-efficient schedules. Stochastic scheduling models the performance variance of a resource using a stochastic value (i.e. distribution) and then proposes scheduling heuristics that use this value. By parameterizing models with stochastic information, the resulting performance prediction is also a stochastic value. Such information can be more useful to a scheduler than point predictions with an unknown range of accuracy. The authors introduce a tuning factor which represents the variability of the system as a whole as some constant number of standard deviations away from a mean value. Resource platforms with a smaller variability are given scheduling priority in this scheme. Extensions of well-known scheduling methods such as time-balancing are shown to achieve better performance under stochastic time-balancing.

Co-allocation

Resource co-allocation of multiple resources introduces a new scheduling problem for Grid applications. In the co-allocation model, the Grid application requires access to a specific set of resources concurrently. Several approaches for addressing this problems have been proposed

[Chapin, et al., Foster et. al., 1999]. Scheduling techniques include atomic all-or-nothing semantics in which either all resources are acquired and the application starts, or it must wait. If the application starts and one or more resources are taken away or fail, then the application must abort. Another approach is advanced reservations in which resources can be locked at some future time so that they are available together. Co-allocation based reservation schemes usually try to provide the soonest time a reservation of the desired length can be accomodated across all desired resources (See Algorithm 1).

```
begin
    rh-a ← CreateReservation(contact-a, "&(reservation_type=compute)
    (start_time="10:30pm") (duration="1 hour") (nodes=32)");
    if rh-a is null then exit ;
    repeat
        (contact-b, id-b, contact-net) ← FindNextCandidate();
        rh-b ← CreateReservation(contact-b, "&(reservation_type=compute)
        (start_time="10:30pm") (duration="1 hour") (percent_cpu=75)");
        if rh-b is null then continue ;
        rh-net ← CreateReservation(contact-net, "&(reservation_type=network)
        (start_time="10:30pm") (duration="1 hour") (bandwidth=200)
        (endpoint-a=id-a)(endpoint-b=id-b)");
        if rh-net is null then CancelReservation (rh-b);
    until rh-b and rh-net are defined;
    if rh-b is null then signal that search failed;
end
```

Algorithm 1: Pseudo-code for a ReserveResources Subroutine

Legion [Chapin, et al.] provided the first co-allocation system for the Grid, and supports both co-allocation and advanced reservations. Legion provides full autonomy support, allowing an scheduling request to return a scheduling token which is later redeemed when the job is actually started. Legion supports multiple simultaneous schedulers, including per-application, per-user, and default schedulers; these can in fact implement the techniques described here, such as Prophet or stochastic scheduling. Under Legion, schedulers produce a list of proposed target schedules ranked by desirability, and an Enactor component is responsible for verifying that the resources are in fact available and implementing the schedule. This allows maximum flexibility and support for autonomy.

Research Issues and Summary

The central problem in distributed scheduling is assigning a set of tasks to a set of processors in accordance with one or more optimizing criteria. We have reviewed many of the algorithms and

mechanisms developed thus far to solve this problem. However, much work remains to be done.

Algorithms

Until now, scheduling algorithms have concentrated mainly on systems of homogeneous processors. This has worked well for parallel machines, but has proved to be unrealistically simple for distributed systems. Some of the simplifying assumptions made include constant-time or even free inter-task communication, processors with the same instruction set, uniformity of available files and devices, and the existence of plentiful primary and secondary memory. In fact, the vast majority of the algorithms listed in the survey model the underlying system only in terms of CPU speed and a simplified estimate of interprocessor communication time.

These simple algorithms work moderately well to perform load balancing on networks of workstations that are, in most senses, homogeneous. However, as we look to the future and attempt to build wide-area distributed systems composed of thousands of heterogeneous nodes, we will need policies capable of making good scheduling decisions in such complex environments.

Distributed Scheduling Mechanisms

As we have seen, current scheduling systems do a good job of meeting the technical challenges of supporting the relatively simplistic scheduling policies that have been developed to date. Future work will expand in new directions, especially in the areas of heterogeneity, security, and the social aspects of distributed computing.

Just as scheduling algorithms have not considered heterogeneity, distributed scheduling mechanisms have only just begun to support scheduling in heterogeneous systems. Some of the major obstacles to be overcome include differences in the file spaces, speeds of the processors, processor architectures (and possible task migration between them), operating systems and installed software, devices, and memory. Future support mechanisms will have to make this information available to scheduling algorithms to fully utilize a large, heterogeneous distributed system.

The challenges in the preceding list are purely technical. Another set of problems arises from the social aspects of distributed computing. Large-scale systems will be composed of machines from different administrative domains; the social challenge will be to ensure that computations that cross administrative boundaries do not compromise the security or comfort of users inside each domain. To be successful, a distributed scheduling system will have to provide security both for the foreign task and for the local system; neither should be able to inflict harm upon the other. In addition, the scheduling system will have to assure users that their local rules for use of their machines will be followed. Otherwise, the computing paradigm will break down, and the large

system will disintegrate into several smaller systems under single administrative domains.

Defined Terms

Autonomy The freedom to be different or behave differently than other nodes within the system.

Centralized mechanisms Mechanisms in which data is stored at a single node, or which pass all data to a single node for a decision.

Distributed mechanisms Mechanisms in which decisions are made on the local system, based on data located on that system.

Distributed memory A system in which processors have different views of memory. Often, each processor has its own memory and cannot directly access another processor's local memory.

Distributed systems Systems with a high degree of autonomy.

Global scheduling The assignment of tasks to processors (also called task placement and matching).

Heterogeneous systems The property of having different underlying machine architecture or systems software.

Job A group of tasks cooperating to solve a single problem.

Load balancing A special form of load sharing in which the system attempts to keep all nodes equally busy.

Load sharing The practice of moving some of the work from busy processors to idle processors. The system does not necessarily attempt to keep the load equal at all processors; instead, it tries to avoid the case where some processors are heavily loaded while others sit idle.

Local scheduling The decision as to which task, of those assigned to a particular processor, will run next on that processor.

Loosely-coupled hardware A message-passing multiprocessor.

Mechanism The ability to perform an action.

Parallel systems Systems with a low degree of autonomy.

Policy A set of rules that decide what action will be performed.

Shared memory A system in which all processors have the same view of memory. If processors have local memories, then other processors may still access them directly.

Space sharing A system in which several jobs are each assigned exclusive use of portions of a common resource. For example, if some of the processors in a parallel machine are dedicated to one job, while another set of processors is dedicated to a second job, the jobs are space sharing the CPUs.

Stability The property of a system that the service rate is greater than or equal to the arrival rate. A stable scheduling algorithm will not make a stable system unstable.

Task The unit of computation in a distributed system; an instance of a program under execution.

Task migration The act of moving a task from one node to another within the system.

Tightly-coupled hardware A shared-memory multiprocessor.

Time sharing A system in which jobs have the illusion of exclusive access to a resource, but in which the resource is actually switched among them.

References

[Andrews et al., 1982] G. R. Andrews, D. P. Dobkin, and P. J. Downey. Distributed allocation with pools of servers. In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 73–83. ACM, August 1982.

[Berman et. al., 1996] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-Level Scheduling on Distributed Heterogeneous Networks. *Proceedings of Supercomputing 1996*.

[Blake, 1992] B. A. Blake. Assignment of Independent Tasks to Minimize Completion Time. *Software–Practice and Experience*, 22(9):723–734, September 1992.

[Bryant and Finkel, 1981] R. M. Bryant and R. A. Finkel. A stable distributed scheduling algorithm. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 314–323. IEEE, April 1981.

[Casanova et. al., 2000] H. Casanova, G. Obertelli, F. Berman, and R. Wolski, The AppLeS Parameter Sweep Template: User-level Middleware for the Grid. *Proceedings of SC 2000*, 2000.

[Casavant and Kuhl, 1984] T. L. Casavant and J. G. Kuhl. Design of a loosely-coupled distributed multiprocessing network. In *Proceedings of the International Conference on Parallel Processing*, pages 42–45. IEEE, August 1984.

[Casavant and Kuhl, 1988] T. L. Casavant and J. G. Kuhl. A Taxonomy of Scheduling in General-Purpose Distributed Computing Systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, February 1988.

[Casey, 1981] L. M. Casey. Decentralised scheduling. *Austrialian Computer Journal*, 13(2):58–63, May 1981.

[Chapin, et al.] S. J. Chapin, D. Katramatos, and J. Karpovich. The Legion Resource Management System *Proceedings of the IPPS Workshop on Job Scheduling Strategies for Parallel Processing*, pp. 105–114, San Juan, Puerto Rico, 1999.

[Chapin and Spafford, 1994] S. J. Chapin and E. H. Spafford. Support for Implementing Scheduling Algorithms Using MESSIAHS. *Scientific Programming*, Vol. 3, pp 325–340, 1994.

[Chou and Abraham, 1983] T. C. K. Chou and J. A. Abraham. Load redistribution under failure in distributed systems. *IEEE Transactions on Computers*, C-32(9):799–808, September 1983.

[Chowdhury, 1990] S. Chowdhury. The Greedy Load Sharing Algorithm. *Journal of Parallel and Distributed Computing*, 9:93–99, 1990.

[Foster et. al., 1999] I. Foster, et al. A Distributed Resource Management Architecture that supports Advanced Reservations and Co-Allocation. *Proceedings of the International Workshop on Quality of Service*, 1999.

[Gantz et al., 1989] C. A. Gantz, R. D. Silverman, and S. J. Stuart. A Distributed Batching System for Parallel Processing. *Software–Practice and Experience*, 19, 1989.

[Gao et al., 1984] C. Gao, J. W. S. Liu, and M. Railey. Load balancing algorithms in homogeneous distributed systems. In *Proceedings of the International Conference on Parallel Processing*, pages 302–306. IEEE, August 1984.

[Ghafoor and Ahmad, 1990] A. Ghafoor and I. Ahmad. An Efficient Model of Dynamic Task Scheduling for Distributed Systems. In *Computer Software and Applications Conference*, pages 442–447. IEEE, 1990.

[Hochbaum and Shmoys, 1988] D. Hochbaum and D. Shmoys. A Polynomial Approximation Scheme for Scheduling on Uniform Processors: Using the Dual Approximation Approach. *SIAM Journal of Computing*, 17(3):539–551, June 1988.

[Hsu et al., 1989] C. C. Hsu, S. D. Wang, and T. S. Kuo. Minimization of Task Turnaround Time for Distributed Systems. In *Proceedings of the 13th Annual International Computer Software and Applications Conference*, 1989.

[Huang et al., 1989] C. H. Huang, T. L. Huang, and J. Y. Juang. On Processor Allocation in Hypercube Systems. In *Proceedings of the 13th annual international computer software and Applications Conference*, 1989.

[Klappholz and Park, 1984] D. Klappholz and H. C. Park. Parallelized process scheduling for a tightly-coupled MIMD machine. In *Proceedings of the International Conference on Parallel Processing*, pages 315–321. IEEE, August 1984.

[Litzkow, 1987] M. J. Litzkow. Remote UNIX: Turning Idle Workstations Into Cycle Servers. In *USENIX Summer Conference*, pages 381–384, 2560 Ninth Street, Suite 215, Berkeley, CA 94710, 1987. USENIX Association.

[Lo, 1988] V. M. Lo. Heuristic Algorithms for Task Assignment in Distributed Systems. *IEEE Transactions on Computers*, 37(11):1384–1397, November 1988.

[Matchmaking, 1998] Matchmaking: Distributed Resource Management for High Throughput Computing. *Proceedings of the 7th IEEE Interna-*

tional Symposium on High Performance Distributed Computing, 1998.

[NWS, 1999] R. Wolski, N. T. Spring, and J. Hayes. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *The Journal of Future Generation Computing Systems*, 1999.

[Ni and Abani, 1981] L. M. Ni and K. Abani. Nonpreemptive load balancing in a class of local area networks. In *Proceedings of the Computer Networking Symposium*, pages 113–118. IEEE, December 1981.

[Nichols, 1987] D. A. Nichols. Using Idle Workstations in a Shared Computing Environment. In *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, pages 5–12. ACM, 1987.

[Price and Salama, 1990] C. C. Price and M. A. Salama. Scheduling of Precedence-Constrained Tasks on Multiprocessors. *Computer Journal*, 33(3):219–229, June 1990.

[Ramakrishnan et al., 1991] S. Ramakrishnan, I. H. Cho, and L. Dunning. A Close Look at Task Assignment in Distributed Systems. In *INFOCOM '91*, pages 806–812, Miami, FL, April 1991. IEEE.

[Reif and Spirakis, 1982] J. Reif and P. Spirakis. Real time resrouce allocation in distributed systems. In *Proceedings of the Symposium on Principles of Distributed Computing*, pages 84–94. ACM, August 1982.

[Schopf and Berman, 1999] J. Schopf and F. Berman Stochastic Scheduling. *Proceedings of SC99*, November, 1999.

[Shirazi et al., 1995] B. A. Shirazi, A. R. Hurson, and K. M. Kavi, editors. *Scheduling and Load Balancing in Parallel and Distributed Systems.* IEEE Computer Society Press, 1995. ISBN 0-8186-6587-4.

[Singhal and Shivaratri, 1994] M. Singhal and N. G. Shivaratri. *Advanced Concepts in Operating Systems.* McGraw–Hill, 1994. ISBN 0-07-13668-1.

[Spring and Wolski] N. Spring and R. Wolski. Application Level Scheduling of Gene Sequence Comparison on Metacomputers. *Proceedings of the 12th ACM Conference on Supercomputiing*, July 1998.

[Squillante et. al., 2001] M. S. Squillante, Y. Zhang, A. Sivasubramaniam, N. Gautam, H. Franke, and J. Moreira Modeling and Analysis of Dynamic Coscheduling in Parallel and Distributed Environments *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, 2002.

[Stankovic, 1984] J. A. Stankovic. Simulations of three adaptive, decentralized controlled, job scheduling algorithms. *Computer Networks*, 8(3):199—217, June 1984.

[Stankovic and Sidhu, 1984] J. A. Stankovic and I. S. Sidhu. An adaptive bidding algorithm for processes, clusters and distributed groups. In *Proceedings of the International Conference on Distributed Computing Systems*, pages 49–59. IEEE, May 1984.

[Stankovic, 1985] J. A. Stankovic. Stability and distributed scheduling algorithms. In *Proceedings of the 1985 ACM Computer Science Conference*, pages 47–57. ACM, March 1985.

[Stone, 1977] H. S. Stone. Multiprocessor Scheduling with the Aid of Network Flow Algorithms. *IEEE Transactions on Software Engineering*, SE-3(1):85–93, January 1977.

[Stumm, 1988] M. Stumm. The Design and Implementation of a Decentralized Scheduling Facility for a Workstation Cluster. In *Proceedings of the 2nd IEEE Conference on Computer Workstations*, pages 12–22. IEEE, March 1988.

[Su et. al., 1999] A. Su, F. Berman, R. Wolski, and M. Mills Strout. Using AppLeS to Schedule Simple SARA on the Computational Grid. *International Journal of High Performance Computing Applications*, pages 253–262. Vol. 13, No. 3, 1999.

[Swanson et al., 1993] M. Swanson, L. Stoller, T. Critchlow, and R. Kessler. The design of the schizophrenic workstation system. In *Proceedings of the Mach III Symposium*, pages 291–306. USENIX Association, 1993.

[Theimer and Lantz, 1989] M. M. Theimer and K. A. Lantz. Finding Idle Machines in a Workstation-Based Distributed System. *IEEE Transactions on Software Engineering*, 15(11):1444–1458, November 1989.

[Van Tilborg and Wittie, 1984] A. M. Van Tilborg and L. D. Wittie. Wave scheduling—decentralized scheduling of task forces in multicomputers. *IEEE Transactions on Computers*, C-33(9):835–844, September 1984.

[Weissman, 1999] J.B. Weissman. Prophet: Automated Scheduling of SPMD Programs in Workstation Networks. *Concurrency: Practice and Experience*, 11(6):301–321, 1999.

[Weissman, 1998] J.B. Weissman. Gallop: The Benefits of Wide-Area Computing for Parallel Processing. *Journal of Parallel and Distributed Computing*, Vol. 54, No. 2, pp. 183-205, November 1998.

[Zhou et al., 1993] S. Zhou, X. Zheng, J. Wang, and P. Delisle. Utopia: a Load Sharing Facility for Large, Heterogeneous Distributed Computer Systems. *Software—Practice and Experience*, 23(12):1305–1336, 1993.

For Further Information

Many of the seminal theoretical papers in the area are contained in *Scheduling and Load Balancing in Parallel and Distributed Systems*, edited by Shirazi, Hurson, and Kavi [Shirazi et al., 1995]. This volume contains many of the papers cited in this chapter, and is an excellent starting point for those interested in further reading in the area.

Advanced Concepts in Operating Systems by Singhal and Shivaratri [Singhal and Shivaratri, 1994] contains two chapters discussing scheduling for parallel and distributed systems. These two references contain pointers to much more information than could be presented here.

Descriptions of other distributed scheduling systems may be found in papers describing Stealth [Singhal and Shivaratri, 1994][Ch. 11], Utopia [Zhou et al., 1993], as well as in [Theimer and Lantz, 1989, Litzkow, 1987, Stumm, 1988]. The Global Grid Forum (http://www.gridforum.org) is codifying best practices for all aspects of Grid systems. More information about scheduling in distributed operating systems such as Sprite, the V System, Locus, and MOSIX can also be found in [Singhal and Shivaratri, 1994].