Parallel Processing Letters

Parallel Incremental Scheduling

Min-You Wu
Department of Computer Science
State University of New York at Buffalo
Buffalo, New York 14260, USA

Received (February 1995)
Revised (August 1995)
Communicated by (Gerasoulis and Yang)

ABSTRACT

Parallel incremental scheduling is a new approach for load balancing. In parallel scheduling, all processors cooperate together to balance the workload. Parallel scheduling accurately balances the load by using global load information. In incremental scheduling, the system scheduling activity alternates with the underlying computation work. This paper provides an overview of parallel incremental scheduling. Different categories of parallel incremental scheduling are discussed.

Keywords: Task-parallel model, load balancing, incremental scheduling, parallel scheduling algorithms.

1. Introduction

There are two basic scheduling strategies: *static scheduling* and *dynamic scheduling*. Static scheduling distributes the workload at compile time. Most existing static scheduling algorithms are sequential, being executed on a single processor system. Dynamic scheduling performs scheduling activities at runtime.

Static scheduling utilizes the knowledge of problem characteristics to reach a global optimal, or nearly optimal, solution[20,9,21,4]. Although many people have conducted their researches in various manners, they all share a similar underlying idea: take a task graph representing the parallel program as input and schedule it onto processors of a target machine to minimize the completion time. The quality of static scheduling relies heavily on accuracy of weight estimation. Scalability of static scheduling is restricted because a large memory space is required to store the task graph. In addition, it is not able to balance the load for problems with an unpredictable structure.

Dynamic scheduling has certain advantages. It is a general approach suitable for a wide range of applications. It can adjust load distribution based on runtime system load information[7,8,15]. However, most runtime scheduling algorithms, when making a load balancing decision, utilize neither problem characteristics nor global load information, with which a better scheduling decision could be made. Efforts to collect load information for a scheduling decision certainly compete resources with the underlying computation during runtime. In a dynamic system, when it intends to quickly and accurately balance the load, the system could become unstable.

It is possible to design a scheduling strategy that combines the advantages of static and dynamic scheduling. This scheduling strategy should be able to generate a well-balanced load without incurring large overhead. With advanced parallel scheduling techniques, this ideal scheduling becomes feasible. In parallel scheduling, all processors cooperate together to schedule work. Parallel scheduling is stable because of its synchronous operation. It utilizes global load information and is able to accurately balance the load.

In this paper, we propose a new approach, called *parallel incremental scheduling*. In parallel incremental scheduling, the system scheduling activity alternates with the underlying computation work during runtime. In the next section, applications of parallel incremental scheduling are discussed.

2. Models and Applications

A parallel program can be modeled by a task graph. In a task graph, nodes are computations and edges are dependences between nodes. For problems with a predictable structure, which are called *static problems*, the task graph can be generated at compile time. This class of problems include matrix multiplication, Gaussian elimination, FFT, etc. For problems with an unpredictable structure, which are called *dynamic problems*, only partial task graphs could be generated. When computation proceeds, the rest of the task graph can be generated incrementally. This class of problems includes multi-grid matrix operation, sparse LU decomposition, etc. For some problems that are completely dynamic, the structure is totally unpredictable, and producing a partial graph is difficult. Only tasks that are ready to execute are generated. Scheduling algorithms for this class of problems could be simple since we do not consider dependences between these nodes. This class of problems include Chess, N-Queens, and many divide-and-conquer algorithms.

Parallel incremental scheduling can be applied to all kinds of problems. A static problem can be scheduled by parallel static scheduling. Algorithms for parallel scheduling will be presented in Section 4. Static problems may be scheduled by parallel incremental scheduling, too, because parallel incremental scheduling provides additional advantages. Dynamic problems can be scheduled by parallel incremental scheduling at runtime, which will be discussed in Section 5. A scheduling algorithm for completely dynamic problems will be presented in Section 6.

The next section provides an overview of parallel incremental scheduling.

3. System Overview

The parallel incremental scheduling system paradigm is shown in Figure 1. A parallel incremental scheduling system starts with a system phase which schedules initial tasks. It is followed by a user computation phase to execute the scheduled tasks. In the second system phase, the old tasks that have not been executed will be scheduled together with the newly-generated tasks. This process will repeat iteratively until all computations are completed.

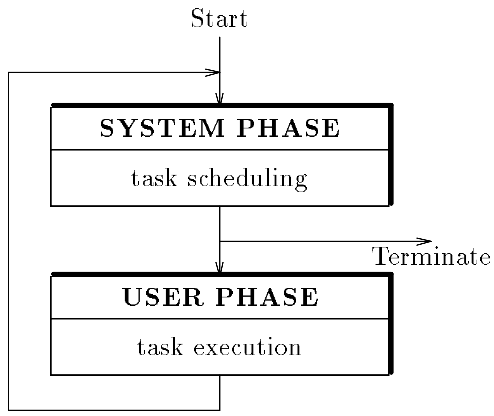

Figure 1: Parallel Incremental Scheduling.

Parallel incremental scheduling and static scheduling share some common ideas. Both of them utilize the systemwise information and perform scheduling globally to achieve a high-quality of load balancing. They also clearly separate the time to conduct scheduling and the time to perform computation. But, parallel incremental scheduling is different from static scheduling in three aspects. First, the scheduling activity is performed at runtime. Therefore, it can apply to dynamic problems. Second, the possible load imbalance caused by inaccurate grain-size estimation can be adjusted by the next turn of scheduling. Third, most existing static scheduling algorithms are sequential and require large memory space to store task graphs. This storage requirement can be reduced by parallelization. Parallel incremental scheduling can further reduce the storage requirement, as scheduling is conducted in an incremental fashion. It then leads to a better scalability for massively parallel machines and large-size applications.

Parallel incremental scheduling is similar to dynamic scheduling to a certain degree. Both methods schedule tasks at runtime instead of compile time. Their scheduling decisions adapt to the runtime system information. Parallel incremental scheduling could be considered as a sub-category of dynamic scheduling because the scheduling process happens at runtime. However, there exist substantial differences. First, the system functions and user computation are mixed together in dynamic scheduling. But, there is a clear cutoff between system and user phases in parallel incremental scheduling, which potentially offers easy management and low overhead. Second, placement of a task in dynamic scheduling is basically an individual action by a processor, based on partial system information. Whereas the scheduling activity in parallel incremental scheduling is always an aggregate operation based on global information. The major characteristics of the three categories are summarized in Table 1.

Table 1: Characteristics of Three Scheduling Approaches

	Static	Dynamic	Parallel Incremental
Load information	global	partial	global
Scheduling	compile time	runtime	runtime
Adaptive to system load	no	yes	yes
Sched. & comp. separation	yes	no	yes
Storage requirement	large	small	small

There are other scheduling algorithms in this "stop-and-schedule" fashion. This category of algorithms is sometimes referred to as *prescheduling* which is more closely related to parallel incremental scheduling. Fox *et al.* first adapt prescheduling to application problems with geometric structures[10,14]. Some other works also deal with this type of problems[6,2]. The project PARTI automates prescheduling for nonuniform problems[3]. The dimension exchange method (DEM) is a parallel scheduling algorithm applied to application problems without geometric structure[5]. It balances the load for independent tasks with an equal grain size. The method has been extended by Willebeek-LeMair and Reeves[18] so that the algorithm can run incrementally to correct the imbalanced load due to varied task grain sizes. An incremental scheduling for N-body simulation is presented in[11]. The task graph is rescheduled iteratively to correct the load imbalance. However, its runtime scheduling has not been parallelized yet. Parallel incremental scheduling is a generalized prescheduling approach which uses optimal parallel scheduling algorithms to reach high-quality load balancing.

4. Parallel Scheduling

Parallel incremental scheduling can be presented in its two major components: incremental scheduling and parallel scheduling. The incremental scheduling policy decides when to transfer from a user phase to a system phase and which tasks are selected for scheduling. The parallel scheduling algorithm is applied in the system phase to balance the load. In this section, methodologies and algorithms for parallel scheduling are described.

The memory and speed of a single processor limit the size of a task graph that can be handled. A natural solution to this problem is *using multiprocessors to schedule tasks to multiprocessors.* In fact, without parallelizing the scheduler and running it on a parallel computer, a scheduler for large task graphs is not feasible.

Specifically, the graph generation algorithm and the scheduling algorithm will be parallelized. A parallelized scheduler can run faster. The memory space required in each processor is reduced because the task graph is distributed.

The parallelized scheduler can run on the same machine where the user program runs. In the following, we will call the processors that execute the scheduling algorithm the *physical processors* in order to distinguish them from the *target processors*. It is not necessary for the number of physical processors to be equal to the number of target processors.

A graph generator generates a task graph from user programs. By parallelizing the graph generator, the processors can generate the graph in parallel, and the graph itself is distributed. In general, parallelizing a graph generation algorithm is not difficult, since it is inherently parallel. The other issue to be considered is load balancing. However, even if the load is not well balanced, it will not greatly affect the overall performance because graph generation consumes much less time than scheduling.

On the other hand, parallelizing scheduling algorithms is difficult because of heavy dependences in the task graph. A parallel scheduling algorithm should have the following features:

- **High quality** — it is able to balance the load well.
- **Low overhead** — scheduling overhead must be reduced to a minimum. It is extremely important when a parallel scheduling algorithm is used in a runtime system.

The basic idea behind the parallel scheduling algorithm is that instead of identifying one node to be scheduled each time, we identify a set of nodes that can be scheduled in parallel. We consider two approaches for parallel scheduling. The first one is called the *vertical* scheme. Each physical processor is assigned a set of graph nodes using *time domain partitioning*. Also, each physical processor maintains schedules for one or more target processors. The second one is called the *horizontal* scheme. Each physical processor is assigned a set of graph nodes using *space domain partitioning*. The resultant schedule is also partitioned so that each physical processor maintains a portion of the schedule of every target processors. Each processor will schedule its own portion of the graph before exchanging information with each other to determine the final schedule. The vertical and horizontal schemes are illustrated in Figure 2. The task graph is mapped to the time-space domain. Here, we assume three physical processors schedule the graph to six target processors. Thus, in the vertical scheme, each physical processor holds schedules of two target processors. In the horizontal scheme, each physical processor holds a portion of schedules of six target processors.

The vertical scheme is outlined in Figure 3. The graph should be so partitioned that the dependences between partitions are minimized. The horizontal scheme is outlined in Figure 4. The graph partitioning is easier than that in the vertical scheme. A simple method is to sort the graph in a topological order and partition the graph into P equal sized blocks. With horizontal partitioning, each processor can schedule its graph partition without exchanging information with each other. The problem with this method is that the start time of partitions other than the first one is unknown. The time needs to be estimated and the scheduling quality

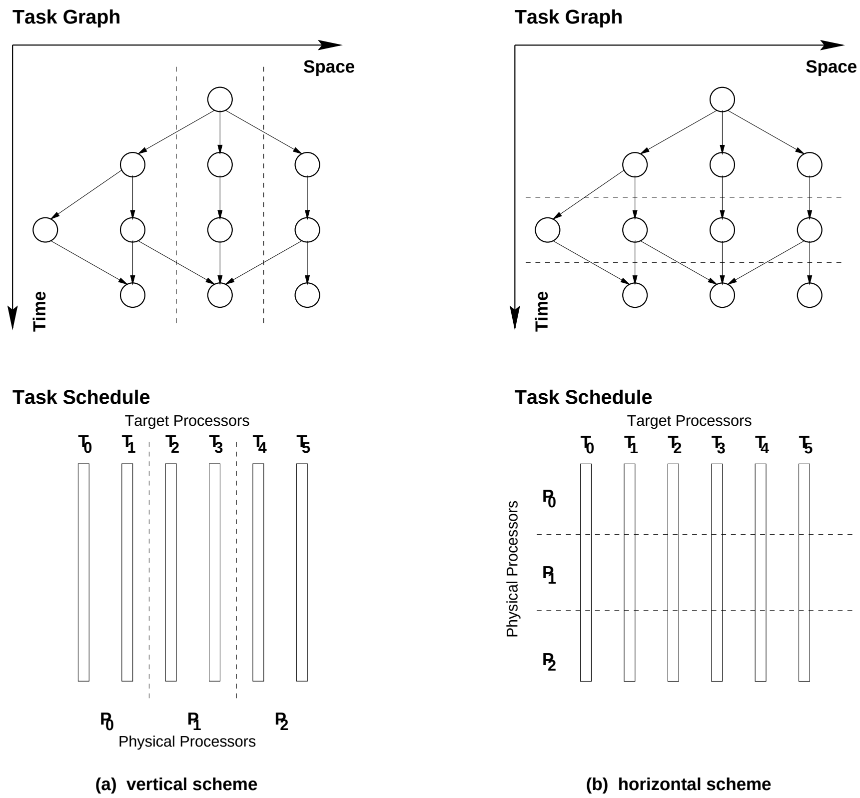

Figure 2: Vertical and Horizontal Schemes.

---

1. Partition the graph into P equal sized sets using *space domain partitioning.*
2. Every processor cooperates together to generate a schedule and each processor maintains schedules for one or more target processors.

Figure 3: The Vertical Scheme for Parallel Scheduling.

---

1. Partition the graph into P equal sized sets using *time domain partitioning.*
2. Each processor schedules its graph partition to generate a sub-schedule.
3. Processors exchange information to concatenate sub-schedules.

Figure 4: The Horizontal Scheme for Parallel Scheduling.

depends on the estimation. In the last step, processors exchange information of their sub-schedules and concatenate them to obtain the final schedule.

Some algorithms have been developed in the horizontal scheme. Ahmad and Kwok proposed a parallel BSA (PBSA) algorithm[1]. Here, we show another algorithm in Figure 5. It is called the parallel modified critical path (PMCP) algorithm, which is a parallelized version of the MCP algorithm[20]. PMCP is simpler and faster than PBSA. First, the nodes are sorted by the as-late-as-possible (ALAP) time[20]. The node list is then partitioned into P equal sized blocks which are assigned to P processors. In this way, the graph is partitioned horizontally. Each processor schedules its partition without considering the dependences between partitions. This is done by ignoring the edges between a node and its remote parent nodes (RPN). An RPN is a parent node belonging to another partition. After every processor finishes its sub-schedule, the final schedule is constructed by concatenating these sub-schedules and determining the actual start time of each node.

---

1. (a) Compute the ALAP time of each node and sort the node list in an increasing ALAP order.

   (b) Partition the graph into equal sized partitions and each partition is assigned to a processor.

2. Each processor schedules its partition:
   Repeat

   (a) Schedule the first node in the node list to a target processor that allows the earliest execution. When determining the execution time of a node on each target processor, idle time slot is considered. Edges between a node and its remote parent nodes are ignored.

   (b) Remove the node from the node list.

   Until all nodes in the partition are scheduled.

3. Concatenate each pair of adjacent sub-schedules. Walk through the schedule to determine the actual start time of each node.

Figure 5: The PMCP Algorithm.

5. Parallel Incremental Scheduling

Parallelizing a scheduling algorithm can simply make it scalable in terms of scheduling speed and memory space. However, the entire task graph could still be too large to fit in the available memory space. Moreover, it cannot adapt to dynamic problems. We suggest an *incremental scheduling* scheme. In this incremental scheme, a portion of the task graph is generated and scheduled to processors. The processors then execute until most tasks have been completed. Then, the next set of tasks are generated and scheduled, and so on. This incremental scheduler has many advantages:

- It can be applied to dynamic problems. In general, the structure of dynamic problems cannot be known at compile-time. An incremental scheduler can adapt to this property by scheduling subgraphs for execution when partial information becomes available.
- The demand for an accurate estimation of execution time becomes less critical. While an inaccurate estimate may result in load imbalance, the load imbalance can be adjusted when scheduling the next set of tasks. An estimator is still needed for a rough estimation, but accuracy of estimation will not directly affect the overall performance.
- This approach can further relax the memory space requirement because the task graph is generated incrementally.
- The application programs need not be recompiled for each problem size since this scheme adapts to different problem sizes. The cost of this approach is the runtime overhead, which can be minimized by grain-size control.

The incremental scheme does not only apply to graph scheduling, but also to graph generation. The graph generation is based on demand of scheduling. It can be triggered when any processor runs out of its tasks. Other policies may resume the graph generation earlier. It needs also to decide how much of the graph will be generated at a time. The size of the subgraph to be generated each time is limited by the available memory space. For some problems, only a small portion of the graph can be generated because the rest of the graph is not known at the time.

6. Parallel Incremental Scheduling for Dynamic Problems

Some problems are completely dynamic, that is, only tasks that are ready to execute are available for scheduling. Scheduling these ready tasks instead of a subgraph is simpler. A scheduling algorithm, called runtime incremental parallel scheduling (RIPS), has been designed for this purpose[16].

Incremental scheduling in RIPS consists of two policies: a local policy and a global policy. Each individual processor determines if it is ready to transfer to the next system phase based on its local condition. Then all processors cooperate together to determine the transfer from the user phase to the system phase based on the global condition.

The local policies used in RIPS include *eager scheduling* and *lazy scheduling*. In the eager scheduling, every task must be scheduled before it can be executed. In the lazy scheduling, scheduling is postponed as much as possible so that some tasks could be executed without being scheduled. Two global policies are the *ALL* policy and the *ANY* policy. The ALL policy states that the transfer from a user phase to the next system phase will be initiated only when all the processors satisfy their local conditions. Whereas, with the ANY policy, as long as one processor has met its local condition, the transfer is initiated. The ANY-Lazy policy has shown to be the best of all four combinations. The details of these policies can be found in[16].

In each system phase, a parallel scheduling algorithm is executed. All processors cooperate together to collect load information and to exchange workload in parallel. The objective of parallel scheduling is to schedule works so that each processor has the same workload. This objective requires an estimation of task execution time. The estimation can be application-specific, leading to a less general approach. For a dynamic problem, such an estimation is difficult to obtain. Therefore, each task is presumed to require the equal execution time and the objective of the algorithm is that each processor has the same number of tasks. Inaccuracy caused by grain-size variation can be corrected in the next system phase.

Parallel scheduling algorithms in RIPS is outlined in Figure 6. This algorithm is in a vertical scheme since all tasks to be scheduled are ready tasks. In the first step, the total number of tasks in the system is counted with a parallel reduction operation. In step 2, processor 0 calculates the average number of tasks per processor and then broadcasts the number to every processor so that each processor knows whether it is overloaded or underloaded. In step 3, the workload is exchanged so that each processor has the almost same number of tasks. This algorithm can maximize locality while achieving a well-balanced load. The details of algorithms for different topologies and their performance can be found in[17,19].

---

1. Let w_i be the number of tasks in processor i. Perform a *sum* reduction of w_i.
2. Processor 0 calculates $w_{avg} = \frac{\sum w_i}{N}$ and broadcasts w_{avg} to all other processors. Each processor determines whether it is overloaded or underloaded by comparing its w_i to w_{avg}.
3. Each overloaded processor determines where to send its excess tasks. Each underloaded processor determines where to receive tasks.

Figure 6: Parallel Scheduling Algorithm for Dynamic Problems.

In Table 2, we compare RIPS to three other dynamic load balancing strategies: random allocation, gradient model, and the receiver-initiated diffusion (RID) algorithm. This comparison has been done on a 32-node CM-5 machine. Two dynamic application problems were tested. The first one, the exhaustive search of the N-Queens problem, has an irregular and dynamic structure. The number of tasks generated and the computation amount in each task are unpredictable. The second one, iterative deepening A* (IDA*) search, is a good example of parallel search techniques [12]. The sample problem is the 15-puzzle with three different configurations. The grain size may vary substantially, since it dynamically depends on the currently estimated cost. Also, synchronization at each iteration reduces the effective parallelism. The comparison is done with (1) the overhead time, which includes all system overhead; (2) the idle time, which is a measure of load imbalance; (3) the execution time; and (4) the efficiency. Here, the efficiency is defined as $\mu = \frac{T_s}{T_p * N}$, where N is the number of processors, T_s is the sequential execution time, and T_p is the parallel execution time. The randomized allocation, although

Table 2: Comparison of Four Scheduling Algorithms

		overhead (seconds)	idle time (seconds)	exec. time (seconds)	efficiency
Exhaustive search 13-Queen	Random	0.40	0.08	1.50	68%
	Gradient Model	0.97	0.88	2.87	36%
	RID	0.41	0.10	1.53	67%
	RIPS	0.29	0.05	1.36	75%
Exhaustive search 14-Queen	Random	0.62	0.25	7.14	88%
	Gradient Model	1.28	5.45	13.0	48%
	RID	0.57	0.27	7.11	88%
	RIPS	0.52	0.08	6.87	91%
Exhaustive search 15-Queen	Random	1.10	1.50	43.4	94%
	Gradient Model	2.10	30.5	73.4	56%
	RID	0.78	1.42	43.0	95%
	RIPS	0.97	0.23	42.0	97%
IDA* search config. #1	Random	0.34	0.63	2.31	58%
	Gradient Model	0.85	1.70	3.89	34%
	RID	0.95	2.57	4.86	28%
	RIPS	0.53	0.13	2.04	66%
IDA* search config. #2	Random	0.65	1.97	9.02	71%
	Gradient Model	2.34	7.25	16.0	40%
	RID	3.20	14.8	23.4	27%
	RIPS	1.17	0.41	7.98	80%
IDA* search config. #3	Random	1.33	4.65	33.5	82%
	Gradient Model	2.28	22.6	52.3	52%
	RID	2.35	6.83	36.7	75%
	RIPS	1.04	0.52	30.2	91%

its locality is not good, can balance the load fairly well. The gradient model [13] does not show good performance for the N-Queens problem. Generally speaking, it cannot balance the load well, since the load is spread slowly. In addition, the system overhead is large because information and tasks are frequently exchanged. RID is a near-neighbor diffusion algorithm[18]. RID shows a better performance than the randomized allocation in some cases. However, it does not perform well for IDA* because of low parallelism. It has been known that a receiver-initiated approach does not perform well in a lightly loaded system[8]. When the problem size becomes large, as in configuration #3, RID's performance is improved. RIPS can balance the load very well, and incremental scheduling is able to correct the load imbalance. One may suspect large overhead from this accurate load balancing algorithm. A surprising observation is that the overhead of RIPS is slightly larger than that of the randomized allocation, and much smaller than that of other dynamic scheduling algorithms. It is partly due to the fact that many tasks are packed together for transmission, which reduces communication overhead. Whereas, in dynamic scheduling, tasks are distributed individually.

7. Concluding Remarks

Parallel scheduling gives load balancing a new direction. In this approach, processors schedule the load in parallel. It balances the load very well and effectively reduces the processor idle time. Parallel scheduling is fast and scalable.

Parallel incremental scheduling combines the advantages of static scheduling and dynamic scheduling, adapts to dynamic problems, and produces high-quality load balance. It has been widely believed that a scheduling method that utilizes global information is neither practical, nor scalable. It is not necessarily true when an advanced parallel scheduling technique is used. We have demonstrated a scalable scheduler that uses the global load information to optimize load balancing. Its overhead is comparable to the low-overhead randomized allocation. Parallel incremental scheduling is a synchronous approach which eliminates the stability problem and is able to balance the load quickly and accurately. It applies to a wide range of applications, from slightly irregular ones to highly irregular ones.

Parallel incremental scheduling is a new approach. It has a good potential to provide high-quality load balancing. Parallel incremental scheduling is still under development. Major research topics include design of parallel scheduling algorithms and incremental scheduling methodologies. Currently, some parallel scheduling algorithms have been developed. Although the incremental scheduling methodology has not been well-developed yet, one for dynamic problems used in RIPS has been tested. Many open problems need to be solved. High-quality parallel scheduling algorithms with low complexity are to be developed. It can be achieved by parallelizing the existing sequential scheduling algorithms or by designing new parallel scheduling algorithms. Runtime overhead depends on not only the complexity of the scheduling algorithm, but also the methodologies of incremental scheduling. New incremental scheduling algorithms are to be developed. Finally, a system needs to be developed that can test different methodologies and algorithms.

Acknowledgements

We are very grateful to Feeley for the elegant N-Queens program. The performance data was gathered on the CM-5 at NPAC, Syracuse University.

References

1. I. Ahmad and Y.K. Kwok. A parallel approach to multiprocessor scheduling. In *Int'l Parallel Processing Symposium*, 1995.
2. M.J. Berger and S. Bokhari. A partitioning strategy for non-uniform problems on multiprocessors. *IEEE Trans. Computers*, C-26:570–580, 1987.
3. H. Berryman, J. Saltz, and J. Scroggs. Execution time support for adaptive scientific algorithms on distributed memory machines. *Concurrency: Practice and Experience*, 1991.
4. Y.C. Chung and S. Ranka. Applications and performance analysis of a compile-time optimization approach for list scheduling algorithms on distributed memory

multiprocessors. In *Supercomputer '92*, November 1992.

5. G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *J. of Parallel Distrib. Comput.*, 7:279–301, 1989.
6. K. M. Dragon and J. L. Gustafson. A low-cost hypercube load balance algorithm. In *Proc. of the 4th Conf. on Hypercube Concurrent Computers and Applications*, pages 583–590, 1989.
7. D. L. Eager, E. D. Lazowska, and J. Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Software Eng.*, SE-12(5):662–674, May 1986.
8. D. L. Eager, E. D. Lazowska, and J. Zahorjan. A comparison of receiver-initiated and sender-initiated adaptive load sharing. *Performance Eval.*, 6(1):53–68, March 1986.
9. H. El-Rewini and T. G. Lewis. Scheduling parallel program tasks onto arbitrary target machines. *Journal of Parallel and Distributed Computing*, June 1990.
10. G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors*, volume I. Prentice-Hall, 1988.
11. A. Gerasoulis, J. Jiao, and T. Yang. Experience with graph scheduling for mapping irregular scientific computation. In *First Workshop on Solving Irregular Problems on Distributed Memory Machines in conjunction with International Parallel Processoing Symposium*, pages 1–8, April 1995.
12. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, September 1985.
13. F. C. H. Lin and R. M. Keller. The gradient model load balancing method. *IEEE Trans. Software Engineering*, 13:32–38, January 1987.
14. J.K. Salmon. Parallel hierarchical N-body methods. Technical report, Tech. Report, CRPC-90-14, Center for Research in Parallel Computing, Caltech, 1990., 1990.
15. Niranjan G. Shivaratri, Phillip Krieger, and Mukesh Singhal. Load distributing for locally distributed systems. *IEEE Computer*, 25(12):33–44, December 1992.
16. W. Shu and M. Y. Wu. Runtime Incremental Parallel Scheduling (RIPS) for large-scale parallel computers. In *Proceedings of the 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 456–463, April 1995.
17. W. Shu and M.Y. Wu. Runtime parallel scheduling for distributed memory compupters. In *Int'l Conf. on Parallel Processing*, August 1995.
18. Marc Willebeek-LeMair and Anthony P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans. Parallel and Distributed System*, 9(4):979–993, September 1993.
19. M. Y. Wu. On runtime parallel scheduling. Technical Report 95-34, Dept. of Computer Science, State University of New York at Buffalo, April 1995.
20. M. Y. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Trans. Parallel and Distributed Systems*, 1(3):330–343, July 1990.
21. T. Yang and A. Gerasoulis. PYRROS: Static task scheduling and code generation for message-passing multiprocessors. *The 6th ACM Int'l Conf. on Supercomputing*, July 1992.