Task Scheduling and Parallel Mesh-Sweeps in Transport Computations[1]

Nancy M. Amato
amato@cs.tamu.edu

Ping An
pinga@cs.tamu.edu

Technical Report 00-009
Department of Computer Science
Texas A&M University
College Station, Texas 77843-3112
January 15, 2000

Abstract

In this report, we describe the relationship between task scheduling and deterministic mesh sweeps that arise in particle-transport computations. In particular, we argue that the efficient parallelization of such computations is most accurately viewed as a generalization of the traditional task scheduling problem and not as an application for domain decomposition, as has been generally assumed in the past. In fact, as we show, the transport problem represents an interesting composite task-scheduling problem: given one set of tasks, and multiple dependence graphs for these tasks (one for each distinguishable sweep direction), find an assignment of tasks to processors that minimizes the time required to process all such graphs.

Within this context, our goal is to study and propose scheduling algorithms that are suitable for the particular generalization of the scheduling problem that arises in the context of transport sweeps. This report documents our progress to date and describes future plans. It consists of three parts. First, we define the traditional task scheduling problem and describe relevant related work. Second, we describe our progress in building a C++ task scheduling library that will provide a testbed in which to evaluate and compare the scheduling algorithms we design, including an experimental evaluation of the (traditional) scheduling algorithms implemented to date. Third, we briefly outline our plans for future work.

[1]This work supported in part by NSF CAREER award CCR-9624315, NSF grant ACI-9872126, DOE ASCI ASAP (Level 2 and 3 Programs) grant B347886, and a Hewlett-Packard University Research grant.

1 Introduction

Traditionally, the parallelization of deterministic mesh sweeps that arise in particle-transport computations has been viewed as an application for domain decomposition. Domain decomposition is a technique for partitioning a spatial region, usually represented by a mesh, into roughly equal-sized subdomains, such that the communication between subdomains is minimized. On parallel machines, each processor would be assigned one subdomain.

However, we claim that this problem is more accurately posed as a task scheduling problem. This is because the real goal of the parallelization is to minimize the completion time, not to achieve load balance or minimize communication costs. The latter two are factors which often tend to lead to the first, but they in themselves should not be the primary objective. The difference in these two objectives can be illustrated by comparing the decomposition produced by an 'ideal' domain decomposition and the one resulting from the KBA algorithm [6], which is commonly viewed as the best parallel mesh sweeping transport algorithm known today for regular grids (see Figure 1). For this case, the 'ideal' domain decomposition would be one which partitions the grid into subdomains of equal size while minimizing the surface area of the domains (assuming inter-processor communication is directly related to the surface area of the subdomain). In contrast, the KBA algorithm assigns each processor a *column* of grid cells, which results in tall, skinny subdomains, which increases the surface area of the subdomains relative to the ideal decomposition. Indeed, it can be established theoretically [9] that the KBA algorithm dramatically outperforms an algorithm based on the 'ideal' decomposition.

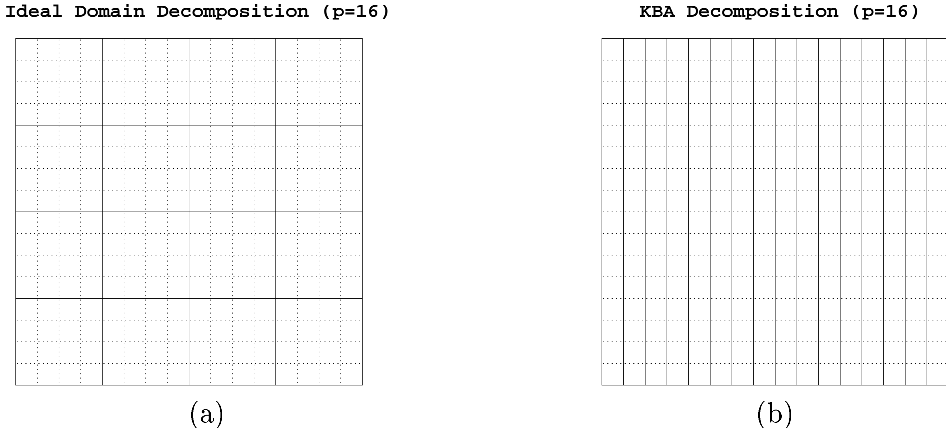

Figure 1: Decompositions from (a) an ideal domain decomposition and (b) the KBA algorithm.

The transport sweep can naturally be modeled as a task scheduling problem as follows. Each direction d considered in the transport equation gives rise to a *weighted dependence graph* $G_d = (V, E_d)$, where each mesh cell represents a task, and adjacent cells $t_i, t_j \in V$ are connected by a directed arc $(t_i, t_j) \in E_d$ if cell t_j needs information from cell t_i in its computation for direction d. Each task has a weight corresponding to the amount of computation it requires, and the arcs are weighted according to the amount of data that must be communicated (communication cost), if the tasks (cells) are assigned to different processors. If the sweep was performed in only one direction, then this problem is precisely the standard task scheduling problem: given a dependence graph for a set of tasks, determine an assignment of tasks to processors that minimizes the total computation time.

The transport problem, however, represents an interesting composite task-scheduling problem:

given one set of tasks, and multiple (distinguishable) dependence graphs for these tasks (one for each sweep direction), find an assignment of tasks to processors that minimizes the time required to process all such graphs. Unfortunately, one cannot hope for efficient algorithms to compute optimal schedules for this composite scheduling problem, since even simple versions of the standard task scheduling problem are known to be NP-complete [10]. Thus, one must turn to heuristic solutions and/or approximation algorithms (see, e.g., [1, 2, 3, 4, 8, 10, 11, 13, 14, 15]).

Within this context, our goal is to study and propose scheduling algorithms that are suitable for the particular generalization of the scheduling problem that arises in the context of transport sweeps. This report documents our progress to date and describes future plans. It consists of three major parts. First, in Section 2, we define the traditional task scheduling problem and describe relevant related work. Second, in Section 3, we describe our progress in building a C++ task scheduling library that will provide a testbed in which to evaluate and compare the scheduling algorithms we design. We present an experimental evaluation of the (traditional) scheduling algorithms implemented to date in our library in Section 4. Finally, in Section 5, we briefly outline our plans for future work.

2 Task Scheduling

In this section, we formally define the (traditional) task scheduling problem. Then we describe how the problem can be modeled with respect to different characteristics, namely, the structure of the computation graph, limitations, if any, assumed regarding the number of available processors, and any assumptions made with respect to the values of the computation and communication costs. Finally, we discuss relevant previous work.

2.1 Problem Definition

The task scheduling problem can be modeled as a weighted directed acyclic graph (DAG) $G = (V, E)$, where V is the set of vertices and E is the set of directed edges. Let $|V| = n$ and $|E| = m$. Vertex t_i represents task t_i, and its weight $w(t_i)$ represents the size of the task (computation). Arc $e_{ij} = (t_i, t_j)$ represents the communication from task t_i to task t_j, and its weight $w(e_{ij})$ reflects the communication cost. The directed edge e_{ij} means task t_j depends on task t_i, i.e., the execution of task t_j cannot be initiated until task t_i completes its computation and the communication to task t_j. Figure 2 shows example computation DAGs.

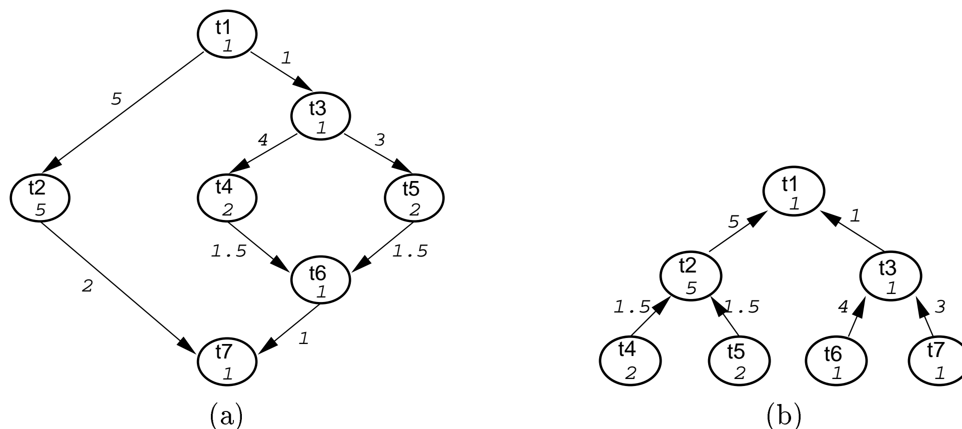

Figure 2: Example computation graphs for task scheduling problems: (a) a DAG, and (b) an ITPG.

Table 1: Model Notation

G: Structure of Graph	
G0	DAG
G1	ITPG
P: Number of Processors	
P0	Limited
P1	Unlimited
C: Communication Constraints	
C0	not constrained
C1	comm. from $t_i \leq w(t_i)$
C2	comm. to/from $t_i \leq w(t_i)$
C3	all comm. $\leq$ smallest task

Table 2: Summary of previous work

Algor.	**Model**	**Time**	**Makespan**
ETF [4]	(G0,P0,C0)	$O(n^2p)$	$(2-\frac{1}{p})OPT(I)+C_l$
KBL [5]	(G0,P1,C0)	$O(n(n+m))$	N/A
SAR [11]	(G0,P1,C0)	$O(m(n+m))$	N/A
JLP/D [1]	(G0,P1,C2)	$O(n^2)$	$OPT(I)$
JLP [1]	(G1,P1,C3)	$O(n)$	$OPT(I)$
DSC [14]	(G0,P1,C3)	$O((m+n)\log n)$	$OPT(I)$

The primary goal of task scheduling is to schedule the n tasks on p processors and minimize the *makespan* of the schedule, i.e., the completion time of the last task relative to the start time of the first task. If two dependent tasks are scheduled on the same processor, the data required by the second task can be retrieved directly from the local memory of the processor, so their communication cost is negligible, i.e., the edge weight can be zeroed. The output of the problem is an assignment of tasks to processors.

2.2 Modeling Task Scheduling Problems

The task scheduling problem can be categorized with respect to different characteristics of the problem. We use a representation proposed in [12] which allows us to compare various algorithms in a unified framework (see Table 1).

The first characteristic is the structure of the computation graph. Generally, a set of dependent tasks can be represented as a *directed acyclic graph (DAG)* (see Figure 2(a)), in which each directed edge describes the dependency between two tasks. If there is at most one out-going edge for each node in a DAG, the graph is an *in-tree precedence graph (ITPG)* (see Figure 2(b)). Task scheduling of a DAG is an NP-complete problem. Scheduling an ITPG is also an NP-complete problem except in very special cases, such as unit task size and zero communication cost [10]. Algorithms designed for ITPG scheduling can often be used to schedule a DAG by duplicating tasks, i.e., the DAG is transformed into an ITPG by task duplication [8].

The second characteristic considered is the number of available processors. Algorithms are usually designed for either a limited (fixed) or unlimited number of processors. The problem can be simplified by assuming there are an unlimited number of processors available. The schedule for unlimited number of processors can be converted to a schedule for a limited number of processors by combining processors with light work loads. However, this usually does not provide the same guarantees on the quality of the resulting schedule as can be obtained by algorithms designed for a limited number of processors.

The last characteristic is the relationship between the communication and computation costs. Different constraints are placed on communication costs. For example, the communication cost might be assumed to be less than any of the computation costs.

2.3 Previous Work

Unfortunately, the standard task scheduling is NP-complete, even for unit task size and unit communication cost [10]. However, various heuristic methods have been proposed that obtain sub-optimal

solutions (schedules) in polynomial time [1, 2, 3, 4, 8, 10, 11, 13, 14, 15].

There are mainly two types of strategies that have been used in task scheduling algorithms: *list scheduling* and *cluster scheduling*. In list scheduling algorithms, a task is assigned to a certain processor when it is scheduled, and in clustering algorithms, a set of tasks is inserted into a cluster, and then the entire cluster is assigned to a processor. Table 2 presents a summary of the relevant previous work.

2.3.1 List Scheduling Algorithms

The *Join Latest Predecessor (JLP)* algorithm [1] is designed for an ITPG with unlimited processors (Model (G1, P1, C3)). The basic idea is to assign a task t_j to the same processor as its 'limiting' predecessor, which is defined to be the last task t_i that finishes both its computation and the communication to task t_j among all immediate predecessors of task t_j. The number of processors used is equal to the number of leaves in the ITPG graph. Under the assumption that communication costs are no longer than the shortest task computation time, the JLP algorithm is guaranteed to find the optimal schedule. It runs in time $O(n)$, where n is the number of tasks. JLP can be extended to produce an optimal schedule on a DAG if task duplication is allowed (*JLP/D*). For a limited number of processors, processors with light work loads can be combined to reduce the total number of processors needed. The results may be sub-optimal compared with those obtained by algorithms which assume a limited number of processors, but the schedule might be computed much faster.

For general problems such as Model$(G0, P0, C0)$, the *Earliest Task First (ETF)* algorithm [4] can be used. ETF attempts to schedule each task as early as possible on each processor as it becomes free. To schedule each task, both a processor queue and a task queue are traversed, yielding a running time of $O(pn^2)$, where p is the number of available processors. The makespan of the schedule produced by ETF is bounded by $(2 - 1/p)OPT(I) + C_l$, where $OPT(I)$ is the makespan of the optimal schedule, and C_l is the total communication requirement on the *critical path* in the DAG (a maximum weight root to leaf path, where the weight is the summation of the all the vertex and edge weights on the path).

Other list scheduling algorithms include *Immediate Predecessor Rescheduling (IPR)* [12], algorithm T and *Task Duplication* (T_{dup}) [8], and a priority scheme proposed by Yen *et al.* [15].

2.3.2 Cluster Scheduling Algorithms

Two simple approaches for cluster scheduling are *critical path merge (CPM)* and *heavy edge merge (HEM)* [11], which can both be applied to Model$(G0, P0, C0)$. CPM is a special case of Kim and Browne's algorithm (*KBL*) [5]. In CPM, the critical path in the graph is recursively clustered, and each cluster is assigned to a different processor. In *HEM*, proposed by Sarkar [11], the edges in the graph are merged in sorted order by edge weight, until the makespan cannot be further reduced.

The *Dominant Sequence Clustering (DSC)* [14] algorithm for Model$(G0, P1, C3)$ produces optimal schedules for special classes of DAGs, and has running time $O((m + n) \log n)$. Other cluster scheduling algorithms include *MCP* [13], *MCCP* [14], and *DCP* [7].

3 A C++ Task Scheduling Library: Testbed Infrastructure

We have implemented a C++ task scheduling library (SCHEDULER) which includes several (traditional) heuristic task scheduling algorithms, and various other scheduling utilities. New task

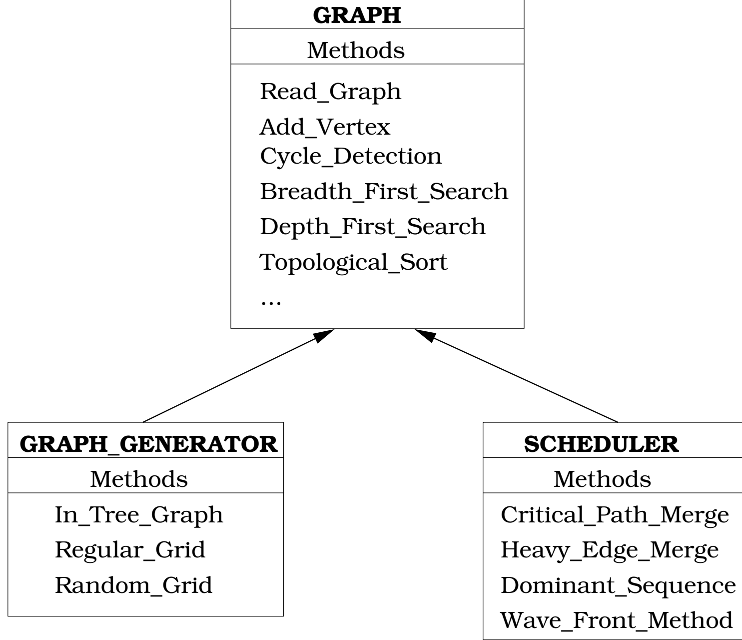

Figure 3: Hierarchy of C++ SCHEDULER, GRAPH_GENERATOR, and GRAPH libraries.

scheduling algorithms can be added conveniently due to the object-oriented design. An auxiliary C++ package (GRAPH_GENERATOR) has also been implemented for generating several classes of dependence graphs, which can be used for testing and evaluating scheduling algorithms. The SCHEDULER and GRAPH_GENERATOR libraries utilize a general graph library (GRAPH), which provides data structures and basic graph algorithms. The hierarchy of the libraries, and their major methods, is shown in Figure 3.

To date, three scheduling algorithms have been implemented in the SCHEDULER, namely CPM, HEM, and DSC (see Section 2). A simple *wave front method (*WFM*)* is also available for comparison purposes.

3.1 GRAPH_GENERATOR

The GRAPH_GENERATOR library can currently be used to generate three types of graphs: in-tree (ITPG), regular grids, and random DAGs. The weights of the vertices (computation) and edges (communication) can be specified by the user.

In an in-tree graph, each vertex has at most one out-going edge. The in-tree graphs constructed by our generator are balanced binary trees (all levels complete except the last), with edges oriented from child to parent. The input to the in-tree generator is n, the number of vertices.

A regular grid is meant to simulate a regular grid in a transport sweep. It is a 2-dimensional $n_1 \times n_2$ grid with the edges oriented in accordance with a given sweep direction. The inputs to the grid generator are n_1 and n_2.

Random DAGs can be viewed as an approximation of an arbitrary grid. The inputs to the random DAG generator are n_1, n_2, d, where n_1 is the maximum number of levels in the graph, n_2 is the maximum number of vertices in each level, and $d \in (0, 1]$ is used to control the vertex degree. The DAG is generated as follows. First, a random integer $\ell \in (0, n_1]$ is generated as the number of levels in the graph. Second, a random integer $v_i \in (0, n_2]$ is generated as the number of vertices

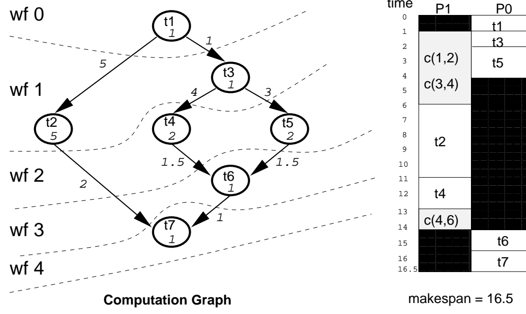

Figure 4: WFM clustering and schedule.

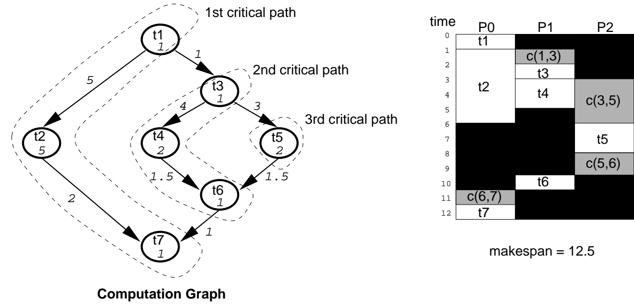

Figure 5: CPM clustering and schedule.

in level i, for all $1 \leq i \leq \ell$. Finally, a random integer $d_{ij} \in [0, v_{i+1} \cdot d]$ is selected as the number of edges from the jth vertex in level i to randomly selected level $i+1$ vertices.

3.2 Scheduler

The input to the scheduling algorithms is a computation DAG for n tasks (adjacency list representation). Each vertex and edge in the DAG has a weight which denotes its associated computation and communication cost, respectively. The output is an assignment of the tasks to p processors, represented in a $p \times n$ array P, such that $P[i, j]$ contains the index of the jth task assigned to processor i, and two arrays $S[i]$ and $F[i]$, $1 \leq i \leq n$, which contain the start and finish times, respectively, for each task. We also output the makespan of the resulting schedule, and the processor and task which lead to that makespan.

The algorithms described below can be applied to any DAG (i.e., they are not restricted to in-tree graphs).

3.2.1 Wave Front Method (WFM)

The wave fronts of the graph are determined according to the level of the vertices in a breadth-first-search traversal of the DAG. For example, the vertices with in-degree zero are contained in the first wavefront, the successors of these vertices comprise the second wavefront, etc. The vertices in each wave front are independent from each other, and are all assigned to different processors (up to the number of available processors).

The running time of WFM is $O(n + m)$. Figure 4 shows the schedule computed by WFM on the example from Figure 2(a).

3.2.2 Critical Path Merge (CPM)

A *critical path* in a DAG is a maximum weight root to leaf path (the path weight is the summation of all vertex and edge weights on the path). CPM computes the critical path P, clusters all tasks in P, assigns them to the same processor, and removes them from the graph. This process is iterated until all tasks are scheduled.

CPM takes $O(n(n + m))$ time, since each critical path computation takes $O(n + m)$, and there are $O(n)$ such paths. Figure 5 shows an example of CPM.

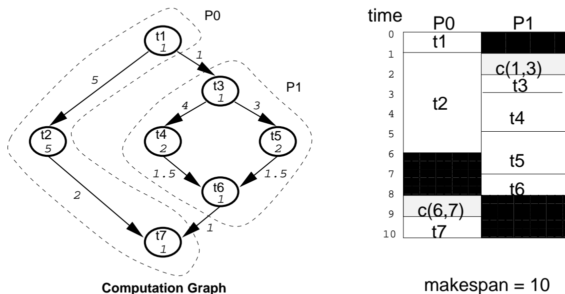

Figure 6: HEM clustering and schedule.

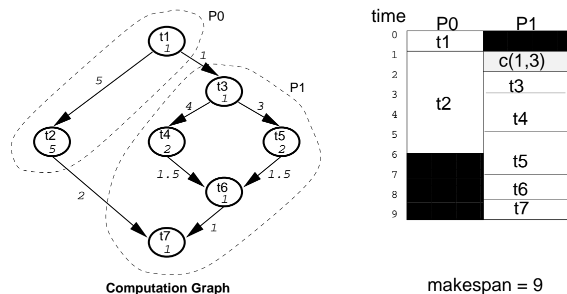

Figure 7: DSC clustering and schedule.

3.2.3 Heavy Edge Merge (HEM)

HEM works by iteratively clustering vertices (tasks) along edges with non-increasing weights. During an initialization stage, the edges are sorted in non-increasing order by edge weight, one task is assigned to each (virtual) processor, and the makespan of this assignment is computed. Then, all edges are processed in sorted order. For each edge, the makespan resulting from merging the tasks associated with the endpoints (perhaps clusters themselves) is computed. If the makespan increases, then the merge is not performed.

The initialization takes $O(m \log m)$ time (sorting), and each edge can be processed in $O(n+m)$ time (computing the makespan). Thus, since there are m edges to process, the total time required by HEM is $O(m(n+m))$. Figure 6 shows an example of HEM.

3.2.4 Dominant Sequence Clustering (DSC)

DSC works by iteratively identifying, and scheduling, so-called dominant sequence tasks which are defined as follows. An unscheduled task is called *free* if all of its predecessors are already scheduled. A *dominant sequence* task is the highest priority free task. The priority of a task is defined as $p(t_x) = top(t_x) + bot(t_x)$, where $top(t_x)$ ($bot(t_x)$) is the length of the longest path from an in-degree 0 (out-degree 0) task to t_x.

DSC initializes $top() = 0$ for every free task and inserts them into a free task list. Next, $bot()$ is computed for each task. The free task t_x with the highest priority is processed first. If $top(t_x)$ cannot be reduced by merging it with a predecessor's cluster, then t_x will be assigned to a new processor. Next, DSC updates the priorities of t_x's successors, and inserts any newly free successors into the free list. The process is repeated until all tasks are scheduled.

The total cost of the algorithm is $O((n+m)\log n)$, using a priority queue for the free list. Figure 7 shows an example of DSC.

4 Task Scheduling Algorithm Performance Study

In this section we examine some experimental results obtained with our prototype task scheduling library. The scheduling algorithms implemented were tested on synthetic graphs generated with GRAPH_GENERATOR, and on some graphs derived from actual sweep grids.

We analyzed the makespans of the schedules obtained and the time required to compute them. As might be expected, it was found that there was a trade-off between the quality of the resulting schedule and the time needed to produce it. In particular, CPM and HEM were seen to produce fairly good schedules on average, but at the cost of large running times. Indeed, the time required

by HEM was so large that it was not a feasible algorithm for large graphs. WFM, which is very fast, performed quite well on the regular grids, but did not do as well on the sweep grids we tested. The makespans of the DSC algorithm were mixed, doing poorly on the synthetic grids, but fairly well on the sweep grids.

Our results suggest that it would be worthwhile to investigate if HEM and CPM can be optimized to achieve faster running times. In addition, we believe it would be useful to study how the performance of DSC can be improved, both in terms of the makespans obtained and the execution costs.

A more detailed analysis of the various cases is presented below.

4.1 Scheduling Synthetic Dependence Graphs

The scheduling algorithms were tested on three classes of graphs created with GRAPH_GENERATOR: in-tree graphs, regular grids, and random grids. For all graphs, the vertex weights (computation) were one, and the edge weights (communication) were random numbers in $(0, 1]$, and $d = 0.7$ for the random DAGs.

The makespans of the schedules created by WFM, CPM, HEM, and DSC are shown in Figure 8(a)-(c) for the three graph classes, respectively, for different sized graphs (n ranged from 100 to 100,000). The cost of a computation-only critical path for each graph is also shown for comparison purposes. It is the summation of the vertex weights on a critical path in the graph (all edge weights are zeroed) and represents a (conservative) lower bound on the best-possible makespan for the graph. For the in-tree graphs (Figure 8(a)), DSC produced the best makespans and WFM the worst, with CPM and HEM falling in between. In contrast, for both regular and random grids (Figure 8(b) and (c)), DSC produced the worst makespans, and CPM and HEM the best (WFM was competitive for the regular grids, but slightly worse for the random grids). In summary, we can conclude that CPM and HEM are relatively good algorithms for all three types of graphs, DSC was only competitive for in-tree graphs, and WFM can produce schedules with high makespans (probably because it disregards communication costs).

The utility of the scheduling algorithms depends on both the quality of the makespan produced and on the time required to compute it. The execution times of the algorithms for regular grids are show in Figure 8(d); the times for the other graph types were similar. The fastest algorithm was WFM, next CPM, and next DSC. HEM is by far the most inefficient algorithm in terms of running time (in fact, it was applied only to graphs with $n \leq 5000$ for this reason). The time of HEM is so high because it re-computes the makespan each time two vertices are merged. Since HEM produces good schedules, it would be useful to see if its running time could be optimized without sacrificing the quality of the resulting makespan too much, e.g., by performing multiple merging steps before recomputing the makespan. Similarly, since DSC allows for some flexibility in terms of its implementation, it would be worthwhile to investigate how to improve the schedules it produces and its execution cost.

4.2 Scheduling Sweep Grids

Seven sweep grids used in real transport computations were scheduled using WFM, CPM, HEM, and DSC.[1] Overall, the results were in accordance with the synthetic grid results.

The makespans of the schedules and the times required to compute them are shown in Figure 9. We see that the schedules produced by CPM and HEM are significantly better than those found by

[1] These graphs were given to us by Will McLendon and Daryl Hawkins.

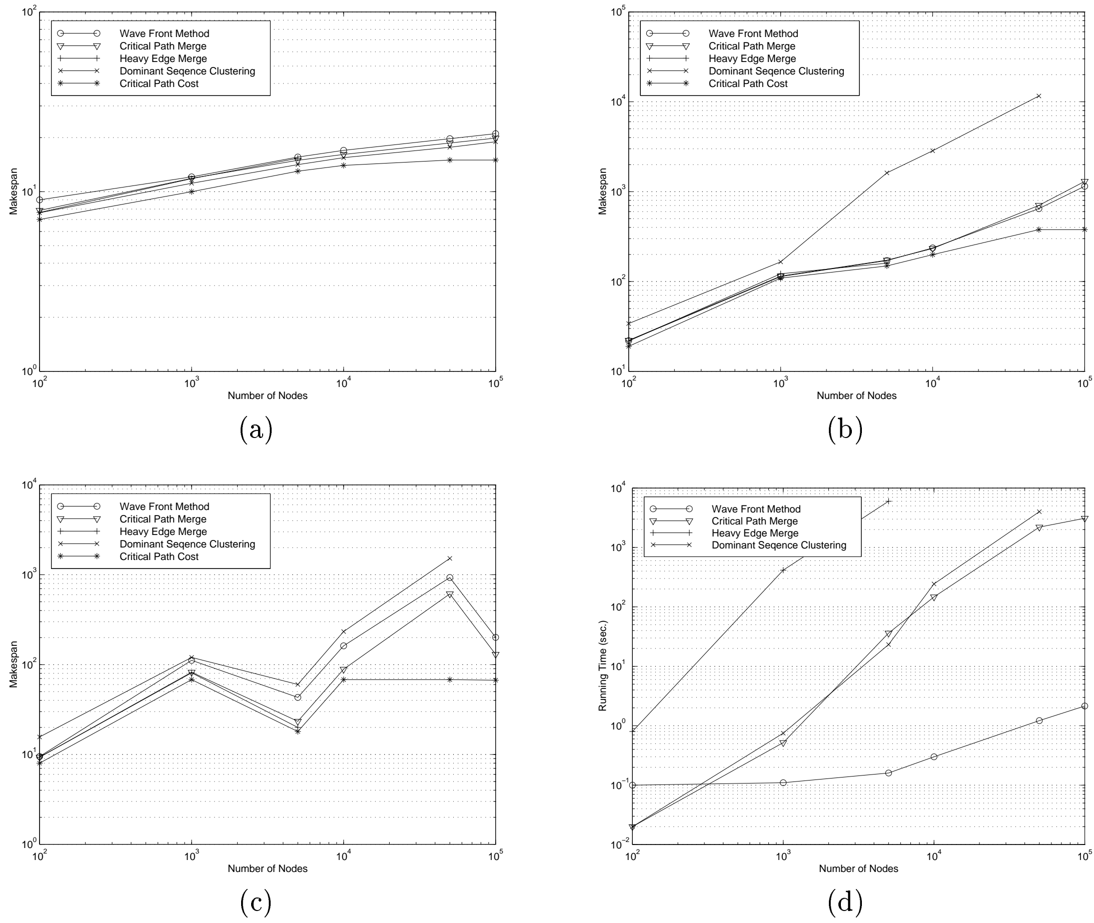

Figure 8: Makespans of Schedules for (a) in-tree graphs, (b) regular grids, and (c) random grids, and (d) the execution time for regular grids (all others were similar).

WFM (HEM was applied only to the four smallest grids due to its large execution time requirements). The makespans of the DSC schedules were usually between the CPM and the WFM makespans. The running times of the algorithms on the sweep grids (Figure 9(b)) was similar to those for the synthetic grids (Figure 8(d)). HEM is slow, WFM is fast, and CPM and DSC are in the middle, with DSC beating CPM in all but one case.

4.3 Cycle Detection

The cycle detection routine in the GRAPH package was tested using both synthetic grids and sweep grids. The average running time on the synthetic grids (in-tree graphs, regular grids, random DAGs, and random graphs with cycles) and the running time of the sweep grid are shown in Figure 10. The time spent to find cycles in a graph with 1 million nodes was around 5 seconds, showing that the cycle detection routine is efficient enough to be used on actual problems.

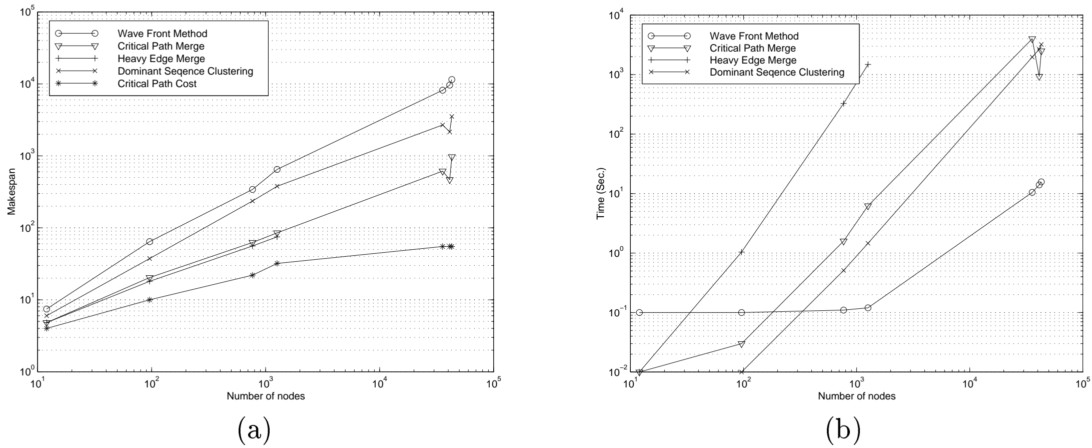

Figure 9: (a) Makespans of schedules and (b) the time required to compute them for sweep grids.

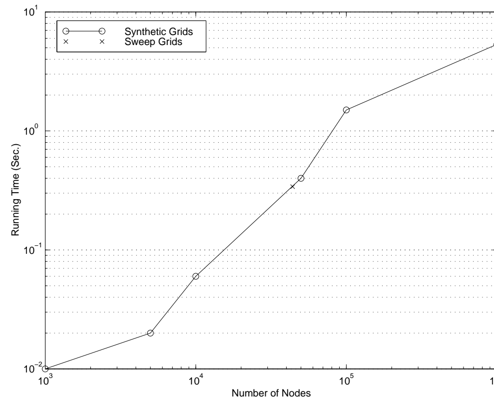

Figure 10: Running Time of Cycle Detection Procedure for synthetic and sweep grids.

5 Future Work

Thus far, our work has led to an increased understanding of the relationship between task scheduling and deterministic transport sweeps. In particular, we suggest that such computations give rise to the following interesting composite task-scheduling problem: given one set of tasks, and multiple dependence graphs for these tasks (one for each distinguishable sweep direction), find an assignment of tasks to processors that minimizes the time required to process all such graphs. This represents a departure from the traditional view which has considered such mesh-sweeps as applications for domain decomposition techniques.

Our current and future work will focus on the following aspects of this problem.

- More scheduling algorithms and hybrid methods will be studied and added to the Scheduler library. Also, we will continue to improve the effectiveness and efficiency of the methods already implemented, and they will be tested on more and larger input graphs.
- We will continue to study the composite scheduling problem posed by the transport sweep computation, and will develop algorithms for it. We believe that such algorithms will need to utilize spatial information related to the problem (as opposed to algorithms for the traditional

problem which simply view the problem as a graph problem). This is because the relationships among the various dependence graphs are intimately related with the geometry of the problem.

- We also intend to study rescheduling/dynamic scheduling methods, which could incrementally improve previous schedules using information from past computations, and by incorporating new information about the computation graph into the scheduling process.
- Finally, due to the large computational requirements of scheduling algorithms, we will study methods for parallelizing them.

References

[1] Frank D. Anger, Jing-Jang Hwang, and Yuan-Chieh Chow. Scheduling with sufficiently loosely coupled processors. *J. Par. Dist. Comp.*, 9:87–92, 1990.

[2] A. Gerasolis and Tao Yang. A comparison of clustering heuristics for scheuling directed acyclic graphs on multiprocessors. *J. Par. Dist. Comp.*, pages 276–291, 1992.

[3] R. L. Graham. Bounds on multiprocessing timing anomalies. *SIAM J. Appl. Math.*, pages 416–429, 1969.

[4] J.-J. Hwang, Y.-C. Chow, F. D. Anger, and C.-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times. *SIAM J. Comput.*, 18(2):244–257, 1989.

[5] S. J. Kim and J. C. Browne. A general approach to mapping of parallel computation upon multiprocessor architectures. In *Int. Conf. Parallel Processing (ICPP)*, pages 1–8, 1988.

[6] K. R. Koch, R. S. Baker, and R. E. Alcouffe. Solution of the first-order form of the 3D discrete ordinates equation on a massively parallel processor. *Transactions of the American Nuclear Society*, 65:198–199, 1992. 1992 Annual Meeting, Boston, MA.

[7] Y. K. Kwok and I. Ahmad. A static scheduling algorithm using dynamic critical path for assigning parallel algorithms onto multiprocessors. In *Int. Conf. Parallel Processing (ICPP)*, pages 155–159, 1994.

[8] D. R. Lopez. Model and algorithms for task allocation in a parallel environment. Technical report, Texas A&M University, 1992. Doctoral Dissertation.

[9] M. M. Mathis, N. M. Amato, and M. L. Adams. A general performance model for parallel sweeps on orthogonal grids for particle transport calculations. In *Proc. ACM Int. Conf. Supercomputing (ICS)*, 2000. To appear.

[10] V. J. Rayward-Smith. Uet scheduling with interprocessor communications delays. Internal Report SYS-C86-06, School of Information Systems. University of East Anglia, Norwich.

[11] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* MIT Press, Massachusetts, 1989.

[12] Y. Wang, N. M. Amato, and D. K. Friesen. Hindsight helps: Deterministic task scheduling with backtracking. In *Int. Conf. Parallel Processing (ICPP)*, pages 170–173, 1997.

[13] M. Wu and D. D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Trans. Par. Dist. Sys.*, 1:330–343, 1990.

[14] T. Yang and A. Gerasoulis. A fast static scheduling algorithm for DAGs on a unbounded number of processors. In *Proc. Supercomputing(SC)*, pages 633–640. IEEE Computer Society Press, November 1991.

[15] C. Yen, S. S. Tseng, and C.-T. Yang. Scheduling of precedence constrained tasks on multiprocessor systems. In *Proc. IEEE Int. Conf. Alg. Arch. Paral. Proc. (ICAAPP)*, pages 379–382, April 1995.