LA-UR-00-5993
Approved for public release; distribution is unlimited.

Title: **An Algorithm for Parallel Sn Sweeps on Unstructured Meshes (UNC)**

Author(s): Shawn D. Pautz

Submitted to:

http://lib-www.lanl.gov/la-pubs/00796284.pdf

An Algorithm for Parallel S_n Sweeps on Unstructured Meshes (UNC)

Shawn D. Pautz
Los Alamos National Laboratory

We develop a new algorithm for performing parallel S_n sweeps on unstructured meshes. The algorithm uses a low-complexity list ordering heuristic to determine a sweep ordering on any partitioned mesh. For typical problems and with "normal" mesh partitionings we have observed nearly linear speedups on up to 126 processors. This is an important and desirable result, since although analyses of structured meshes indicate that parallel sweeps will not scale with normal partitioning approaches, we do not observe any severe asymptotic degradation in the parallel efficiency with modest ($\leq$ 100) levels of parallelism. This work is a fundamental step in the development of parallel S_n methods. (UNC)

Keywords: radiation transport, parallel processing, sweeps, scheduling

Introduction

The standard iterative technique for solving discretized transport equations is source iteration, in which one alternates between solving for the local scattering source and inverting the global streaming-plus-collision operator. In the case of discrete ordinates (S_n) equations derived from the first-order form of the transport equation the streaming-plus-collision operator is usually directly inverted by the method of "sweeping." During a sweep this operator is locally solved for each spatial cell in the mesh in a specified order for a single direction in the discrete ordinates set. This order is constrained by the interaction of the discrete ordinates set with the spatial mesh; a cell cannot be solved for a particular direction until its "upstream" neighbors have been solved. Because of these constraints the solution order often resembles a plane wave "sweeping" across the mesh along the ordinate direction.

Because of the constraints placed on the sweep ordering, parallelization of the sweep process is difficult. Tasks (cell-angle pairs) assigned to a processor cannot begin until cells upstream from them have been solved; the processor may be idle for some time if upstream tasks are on other processors. The assignment of tasks to processors and the ordering of that work must be carefully coordinated in order to obtain good parallel efficiencies and speedups. Sweep problems are a subset of the class of problems known in computer science as "scheduling" problems, which are difficult to solve in general.

For the special case of sweeps on orthogonal hexahedral meshes the KBA (Koch-Baker-Alcouffe) sweep algorithm has been developed [Koch et al., 1992, Baker et al., 1995, Baker and

Alcouffe, 1997, Baker and Koch, 1998]. The KBA algorithm uses a special columnar domain decomposition and a particular sweep ordering to obtain very high parallel efficiencies. This algorithm is difficult to generalize to unstructured meshes; it is not obvious what a "columnar" decomposition on an unstructured mesh is or what the corresponding ordering should be.

The purpose of this paper is to develop an algorithm for efficient parallel sweeps on unstructured meshes. Our focus is on obtaining a good sweep ordering; we defer the problem of obtaining a specialized decomposition and use instead a more traditional decomposition. Although this limits the scalability of our algorithm (one needs both the "right" decomposition and a good ordering for scalability) we are able to obtain near-ideal efficiencies and speedups with over 100 processors. Furthermore, our ordering algorithms are designed to work with any decomposition and should be able to exploit favorable properties of specialized partitions as they become available in the future.

The rest of the paper is organized as follows. First we discuss the nature of the sweep process and relate it to the general class of scheduling problems. We also describe the KBA approach for structured meshes. Next we develop an algorithm for parallel sweeps on unstructured meshes. We then present theoretical performance estimates of our algorithm and we compare these estimates to computational results. Finally, we make some conclusions and recommendations for future work.

Sweeps and the General Scheduling Problem

As mentioned in the introduction, the problem of efficiently parallelizing sweeps is a subset of the class of scheduling problems. Scheduling concerns itself with the distribution and ordering of tasks among processors, particularly when there are data dependencies between tasks. In this section we will discuss S_n sweeps and show how their parallelization can be described in terms of scheduling problems. We will discuss scheduling problems in general and the approaches that have been developed to solve them. Finally we will discuss the KBA algorithm for scheduling sweeps on structured meshes.

The Nature of S_n Sweeps

Discretizations of the Boltzmann transport equation are generally solved by means of source iteration, which we present here for the first-order form of the equation:

$$[\mathbf{\Omega} \cdot \nabla + \sigma_t]\psi^{(l+1)} = M\Sigma\phi^{(l)} + q, \tag{1a}$$

$$\phi^{(l+1)} = D\psi^{(l+1)}, \tag{1b}$$

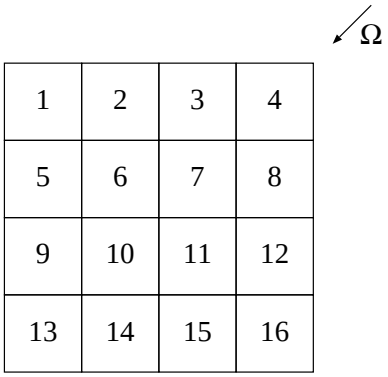
where ψ is the angular flux distribution, ϕ is the set of angular flux moments, l is the iteration index, $\mathbf{\Omega}$ is the direction of particle travel, D, M, and Σ are the discrete-to-moments, moments-to-discrete, and scattering operators, respectively, and σ_t is the total cross section. We assume that Eq. (1) has been discretized in the angular variable by the method of discrete ordinates (S_n) and that some spatial discretization method has also been applied. In order to solve Eq. (1a) it

is necessary to numerically invert $[\mathbf{\Omega} \cdot \nabla + \sigma_t]$, the streaming-plus-collision operator. Although one could globally invert this operator by iterative methods, with first-order S_n methods we may directly invert this operator by the method of sweeping, as described below.

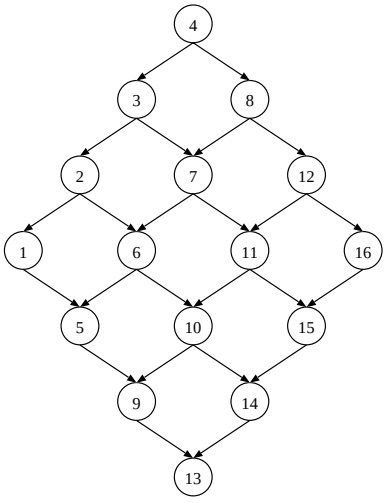
In Figure 1(a) we depict a simple rectangular mesh such as those used in many discrete transport calculations; each of the cells have been numbered. We have also depicted a direction $\mathbf{\Omega}$. In order for the streaming-plus-collision operator to be numerically inverted for $\mathbf{\Omega}$ for a given cell by the method of sweeping, the incoming fluxes for that cell must be known, i.e. the fluxes along cell faces for which $\mathbf{\Omega} \cdot \mathbf{n}$ is negative, where $\mathbf{n}$ is the outward unit normal on the face. These incoming fluxes are determined either from boundary conditions or from "upstream" cells previously solved by the sweep. For example, we cannot solve for $\psi^{(l+1)}(\mathbf{\Omega})$ in cell 3 until we have solved for $\psi^{(l+1)}(\mathbf{\Omega})$ in cell 4, since angular fluxes cross over the right-hand face of cell 3 from cell 4. On the other hand, cell 4 can be solved before any other cells, since the only incoming fluxes it has along $\mathbf{\Omega}$ are from external boundaries, which we assume to be known. We will restrict our attention, without loss of generality, to problems without reflecting boundary conditions; such conditions add dependencies between different directions, which can be treated as a group in a manner analogous to what we are describing here for a single direction.

The situation described above and depicted in Figure 1(a) induces the dependency graph shown in Figure 1(b). Each vertex in the graph represents the work to be done for direction $\mathbf{\Omega}$ for the corresponding cell, and the directed edges represent the dependencies between the cells. A vertex cannot be solved until all of its immediate predecessors have been solved. There is a dependency graph associated with every direction in the angular quadrature set. Since the angular fluxes in different directions are coupled only during the calculation of the scattering source, the sweep dependency graph for a direction is independent of those for other directions. We note that the graph in Figure 1(b) is a directed acyclic graph (DAG); there is no directed path from some vertex to another and back again. This is equivalent to saying that there is no group of two or more vertices that need to be solved simultaneously rather than sequentially. This is also equivalent to saying that the sweep matrix can be put into block lower triangular form and is therefore well suited to direct inversion via forward substitution.

Figure 1 depicts the situation for sweeps on a structured mesh. For unstructured meshes the situation is similar, albeit more complex. In Figure 2(a) we depict a small unstructured mesh of triangular and quadrilateral elements. We also show the associated dependency graph for $\mathbf{\Omega}$ in Figure 2(b). As in the structured case there will be a dependency graph for every direction in the angular quadrature set. As illustrated in Figure 2(b) the structure of the graphs obtained from unstructured meshes can be much more complicated and irregular than those from structured meshes. Furthermore, some meshes will induce graphs that do contain cycles; in Figure 2(b) there is a cycle between cells 8 and 9 and another cycle between cells 16, 17, and 18. Although these cycles are induced by concave (reentrant) cells, cyclic dependencies can also occur among groups of cells in an unstructured mesh even if there are no concave cells. Such situations have already been encountered in serial unstructured codes and have generally been handled by artificially removing one or more dependencies and using information from previous iterations. This procedure may affect the convergence rate of the source iteration process, but it does not affect the converged solution [Wareing et al., 1999, Wareing et al., 2000]. The result of such an approach is to convert a graph with cycles into a DAG; throughout the rest of our analysis we will assume that the sweep graphs are DAGs.

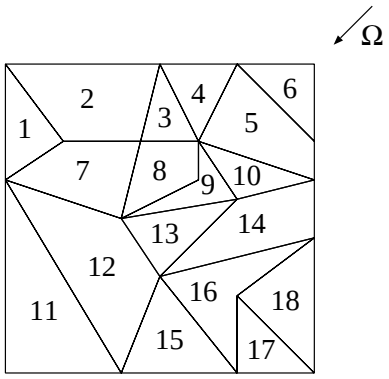


(a) Structured mesh.

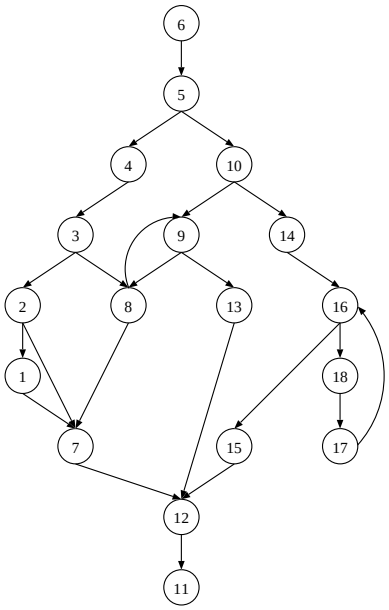


(b) Dependency graph for Ω.

Figure 1: Structured mesh and dependency graph for Ω.



(a) Unstructured mesh.



(b) Dependency graph for Ω.

Figure 2: Unstructured mesh with concave elements and dependency graph for Ω.

For serial codes the ordering of tasks in the sweep is fairly straightforward. In the case of structured meshes a simple expression for the ordering can be obtained since the basic structure of the dependency graph is always the same. For unstructured meshes the sweep graph needs to be analyzed each time since it varies from problem to problem, but it is relatively easy to determine an ordering that satisfies the dependency constraints; any such ordering is acceptable.

The situation is more complicated for parallel implementations. We now must decide how to distribute the tasks among processors; this does not determine the sweep ordering per se, but it does eliminate many otherwise valid orderings, including possibly some very efficient ones. Among the remaining technically valid orderings we somehow must choose one that makes efficient use of the available processors; we want to minimize processor idle time. Finally, wherever there is a dependency (i.e. a need for data) between sweep tasks on different processors there will need to be communication; communication costs must be handled in some manner. To deal with these issues in a systematic way we now turn our attention to the general scheduling problem.

The General Scheduling Problem

In order to describe scheduling problems in general let us first introduce some terms and notation. A directed graph is a tuple $G = (V, E)$, where V is a set of vertices and E is a set of directed edges between vertices in V. A vertex $v_i \in V$ represents a task to be computed; there are $v = |V|$ vertices. An edge $e_{i,j} \in E$ represents a data communication from v_i to v_j (and hence a dependency of v_j on v_i); there are $e = |E|$ edges. Associated with each vertex v_i is a weight τ_i representing the execution time for that task. Associated with each edge $e_{i,j}$ is a weight $c_{i,j}$ representing the communication time between task v_i and v_j; this weight is set to zero (edge zeroing) if tasks v_i and v_j are computed by the same processor.

The scheduling problem is defined as the assignment of tasks to processors and the assignment of start times of tasks (which induces an ordering of tasks) such that dependency constraints are satisfied and such that some objective is met [Gerasoulis and Yang, 1992, Kwok and Ahmad, 1999]. The objective is usually to minimize the solution time, or *makespan*, of the graph, although other objectives such as minimum communication costs are sometimes used. The task execution model generally used is the "static macro-dataflow" model; a processor may not begin work on a task until all necessary data from parent vertices have been received by the processor, once work on a task begins it must continue without preemption until completed, and output data are sent immediately to child tasks in parallel when the task completes. The architecture assumed in most problems is a completely connected set of identical processors.

The general scheduling problem has been shown to be NP-complete [Garey and Johnson, 1979]; no polynomial-time algorithm has ever been found that solves an NP-complete problem [Cormen et al., 1990]. Even under severe restrictions the scheduling problem usually remains NP-complete; only a very few limited scheduling problems have been found to be polynomial-time solvable [Kwok and Ahmad, 1999]. As an illustration of the difficulty of NP-complete problems let us compare a non-polynomial optimal scheduling algorithm of complexity $O(v!)$ with a (hypothetical) polynomial optimal scheduling algorithm of complexity $O(v^2)$. For a graph with only 100 vertices the non-polynomial algorithm would require $O(10^{158})$ operations to determine the optimal schedule, whereas the polynomial algorithm would require only $O(10^4)$ operations. As this example demonstrates the problem of finding an optimal schedule for a given graph is intractable

except for extremely small graphs.

Although optimal scheduling solutions may be effectively impossible to compute, numerous polynomial-time algorithms have been constructed that yield acceptable, albeit suboptimal, solutions [Gerasoulis and Yang, 1992, Kwok and Ahmad, 1999]. These heuristics vary in their assumptions about the graphs and processor architectures that they use. Despite this variety most scheduling algorithms make use of a list scheduling approach [Kwok and Ahmad, 1999]. In list scheduling one first assigns numerical priorities to each task in the graph. Different scheduling algorithms use a number of different prioritization schemes for this process. After the assignment of priorities the algorithm performs a sequence of steps; in each step the algorithm assigns the highest priority ready task (a task that has all input data available) to a processor according to another algorithm-specific heuristic. It may also assign a start time. These assignments continue until the entire graph has been scheduled.

There are some other common characteristics of existing scheduling algorithms. Most have been developed for relatively small graphs, containing tens or occasionally hundreds of vertices. As a consequence many have fairly high time-complexities, for example $O(v^2)$ or $O(ev^2 \log(v))$, since higher complexity algorithms generally produce better solutions and the graphs are usually small enough that the algorithms are not too expensive. Most algorithms also assume the existence of a relatively large number of processors (even if bounded) and use as many as necessary to minimize the solution time. We will observe in the next section that these and other properties are important considerations as we attempt to solve the sweep scheduling problem.

The KBA Scheduling Algorithm for Orthogonal Hexahedral Meshes

As mentioned in the preceding subsection, a number of scheduling algorithms have been developed for special types of graphs. One such algorithm, the KBA algorithm, has been designed to schedule sweeps on orthogonal hexahedral meshes. Although it does not necessarily produce an optimal schedule, it does produce a schedule that yields nearly ideal parallel efficiencies [Koch et al., 1992, Baker et al., 1995, Baker and Alcouffe, 1997, Baker and Koch, 1998].

A typical orthogonal hexahedral mesh is depicted in Figure 3; element boundaries are indicated with thin lines. The mesh has I, J, and K elements along the x-, y-, and z-axis, respectively. We assume that I, J, and K are all $O(N)$. The KBA algorithm uses a spatial domain decomposition consisting of $P_x \times P_y \times 1$ domains, as indicated by bold lines in the figure; P_x and P_y are $O(N)$. Each domain has at most $I_C \times J_C \times K$ elements, where $I_C = [I/P_x]$, $J_C = [J/P_y]$, and $[\cdot]$ is the ceiling function. To simplify the rest of our discussion we will assume that $[I/P_x] = I/P_x$ and $[J/P_y] = J/P_y$. Note that KBA does not decompose in angle; the sweep work for any mesh cell for every direction in the quadrature set is assigned to the same processor.

For a given direction KBA orders the work as depicted in Figure 4. First the processor that has been assigned work at the top of the directed graph for that direction (in this case at the top front right corner) solves an $I_C \times J_C \times K_C$ block of elements, where K_C is $O(1)$ and $I_C J_C K_C$ is the block or "chunk" size. The ordering within the block is irrelevant, as long as it satisfies the dependency constraints. The solution of this block is indicated by its removal from the mesh in Figure 4(a). The processor then communicates newly computed partition boundary fluxes to the processors that have been assigned the partitions to the left and to the back of its partition. Note that it does not need to communicate data downward since all of these cells are in its own

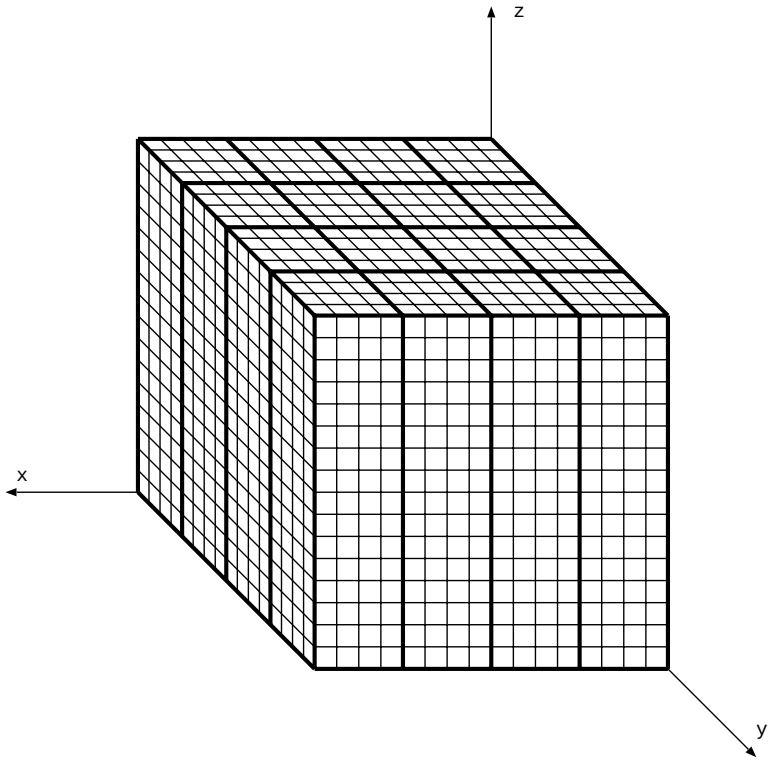


Figure 3: Orthogonal hexahedral mesh with KBA decomposition.

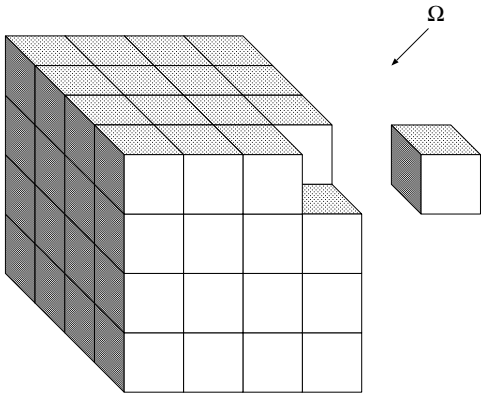
partition. In the next step, as indicated in Figure 4(b), both the original processor and the two to which it communicated solve another set of blocks and then communicate to the left and back. This continues as shown in the figure until most or all processors are performing computations at each step. Eventually one or more processors, starting with the original one, complete their work for that direction, and when all processors have finished we repeat the process for every other direction in the quadrature set in succession.

In order to describe the quality of the KBA schedules let us first define a few terms. The *parallel efficiency* (η) of a parallel algorithm is defined as the ratio of the serial CPU time and the total CPU time:

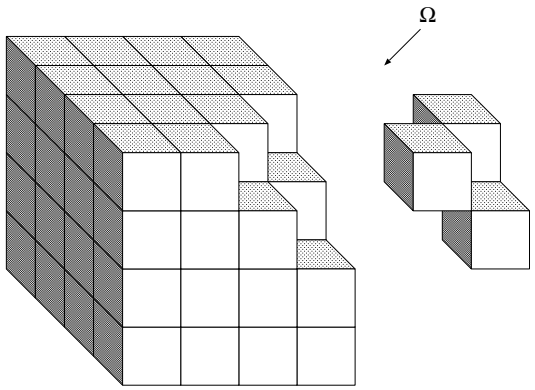
$$\eta = \frac{T_s}{PT_p}, \tag{2}$$

where T_s is the serial execution time, T_p is the parallel execution time, and P is the number of processors. The *parallel computational efficiency* (PCE) of an algorithm is the parallel efficiency in the absence of communication costs. The PCE is an upper bound on the parallel efficiency when communication costs are not negligible; a high PCE is a necessary, albeit insufficient, condition for high actual efficiencies. Finally, the *scalability* of an algorithm describes the trends in the efficiency as both the problem size and the number of processors increase; if the efficiency can be kept asymptotically fixed the parallel algorithm is said to scale [Kumar et al., 1994].

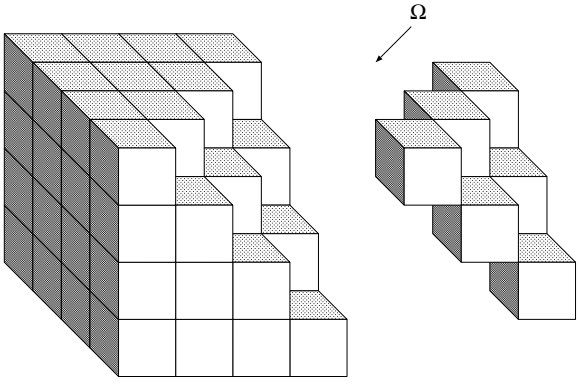
The exact efficiency of the KBA algorithm depends on the communication costs. If com-



(a) Step 1 of sweep.



(b) Step 2 of sweep.



(c) Step 3 of sweep.

Figure 4: KBA sweep ordering.

munication costs are zero, then the efficiency is given by [Koch et al., 1992, Baker et al., 1995, Baker and Alcouffe, 1997, Baker and Koch, 1998]

$$PCE = \frac{K}{K + K_C(I/I_C + J/J_C - 2)}. \quad (3)$$

Note that since I, J, and K are $O(N)$ and the other quantities in Eq. (3) are $O(1)$, the PCE is $O(1)$; the KBA scheduling algorithm scales with N. In the case of an asymptotically large cubic mesh with $I_C = J_C = K_C$ the PCE is 33%. This efficiency can be increased by decreasing K_C with respect to I_C and J_C.

A variant of the simple KBA scheme presented above obtains much higher efficiencies by pipelining the work for several directions. In the ordering that we described previously a processor that had completed its work for a direction would wait for all processors to finish that direction before repeating the process for a new direction. In the alternative scheme when a processor completes all of the work for some direction, it can immediately begin work on the next direction in the same quadrant without waiting. With pipelining we are able to eliminate most of the processor idle time, and the efficiency becomes [Koch et al., 1992, Baker et al., 1995, Baker and Alcouffe, 1997, Baker and Koch, 1998]

$$PCE = \frac{2MK}{2MK + K_C(I/I_C + J/J_C - 2)}, \quad (4)$$

where M is the number of directions in an octant. For an S_8 level-symmetric quadrature on a large cubic mesh with $I_C = J_C = K_C$ the PCE is 91%.

There are some significant differences between KBA and most scheduling algorithms. KBA scheduling may be viewed as a two-step process: first assign domains and then establish the task ordering. Most other algorithms perform these two steps simultaneously. The KBA algorithm deviates somewhat from the static macro-dataflow task execution model by delaying communication until a processor has solved an entire block of cells. Finally, KBA has very low complexity; the scheduling solution is effectively in closed form because of the simple fixed structure of the sweep graph and needs merely to be evaluated with specific parameter values from a given problem.

We have noted that KBA obtains scalable efficiencies, which can be nearly ideal in the pipelined variant. There are two main factors that contribute to KBA's success: the nature of its decomposition and its particular ordering strategy. The combination of these factors yields a scalable scheduling algorithm. We discuss each of these two factors in turn.

We have analyzed the use of a $P_x \times P_y \times P_z$ block decomposition of an $I \times J \times K$ orthogonal hexahedral mesh, where I, J, and K are all $O(N)$. Our analysis shows that it is impossible to obtain a scalable schedule with $O(N^2)$ processors with this decomposition unless one of the processor dimensions (say, P_z) is $O(1)$ and the other two are both $O(N)$. This is a consequence of the fact that it is impossible to construct an ordering such that in some of the steps there are $O(N^2)$ processors with tasks (cells) ready to be computed, which is necessary for scalability, unless the above conditions are met. The KBA decomposition satisfies these conditions, with $P_z = 1$ and with P_x and P_y both $O(N)$. We remark that the above analysis assumes that the number of directions in the angular quadrature set does not scale with the number of processors or the size of the mesh, i.e. it is $O(1)$. We also note that the above conditions are necessary, but

they may not be sufficient, to guarantee the existence of a scalable schedule. Of course, the fact that the efficiencies given by Eqs. (3) and (4) are $O(1)$ regardless of problem size demonstrates that at least one scalable schedule exists for the KBA decomposition.

Even if a decomposition has at least one scalable ordering associated with it, there may be other orderings that do not scale. For example, if KBA were to divide each columnar partition into several narrower columns within the partition and then order the columns sequentially during the sweep, the resulting ordering would have $O(N^2)$ steps, which does not scale. It is important therefore to carefully construct an ordering for a given decomposition that exploits the properties of the decomposition. In the case of KBA we see that tasks are performed in such an order by a processor that a minimum number of tasks needs to be completed in order to generate data needed by other processors. This allows other processors to begin work as soon as possible.

In summary, the parallelization of sweeps is a special case of a scheduling problem. Although optimal schedules are extremely difficult to obtain for general scheduling problems, many suboptimal scheduling algorithms have been developed for a variety of scheduling problems that often yield near-optimal solutions. One specialized scheduling algorithm, the KBA algorithm, constructs sweep orderings on orthogonal hexahedral meshes that yield high parallel efficiencies; it relies on a particular spatial decomposition and a special ordering to obtain scalable schedules.

Development of a Parallel Unstructured Mesh Sweep Algorithm

In this section we develop an algorithm suitable for scheduling parallel sweeps on unstructured meshes. We first show that existing scheduling algorithms are not suitable for this problem. We therefore construct a new list scheduling algorithm that is suitable for modest levels of parallelism. We also introduce several prioritization heuristics for use with the list scheduling algorithm.

Unsuitability of Existing Scheduling Algorithms

Previously we referred to an extensive literature on existing scheduling algorithms. In order to determine whether any of these are suitable for parallelizing the sweeps on unstructured meshes we must first determine what properties a sweep scheduling algorithm should have. We have identified several such properties that we discuss in this subsection.

First, the algorithm should have low complexity, since we expect the sweep graphs to be quite large. For example, a typical large three-dimensional transport calculation may use a mesh with 100,000 elements and an S_8 quadrature (80 directions), yielding a set of sweep graphs with nearly 10^7 vertices (and a similar number of edges). Even an algorithm with a complexity of only $O(v^2)$ would be considered prohibitively expensive for this problem.

Second, the algorithm should schedule on a small, bounded number of processors. By "small" we mean that the number of processors should be much less than the number needed for maximum possible speedup. In typical problems such speedups could in theory require tens or hundreds of thousands of processors; we do not want to restrict the use of a sweep scheduling algorithm to such large systems.

Finally, we would prefer an algorithm that distributes work in the spatial dimension only. One reason for this is that after every set of sweeps we need to recalculate the scattering source. If the calculation of new angular fluxes has been decomposed in angle then we may need to communicate the fluxes from every cell in order to recompute the scattering source, whereas with a purely spatial decomposition we need only communicate angular fluxes that exist on partition boundaries during the sweeps. Such an approach also would make it easier to couple a transport calculation with other parallel applications that use the same mesh.

The above considerations effectively eliminate existing general scheduling algorithms as potential candidates for scheduling sweeps. As we noted previously, almost all general scheduling algorithms have high time-complexity and would therefore be too expensive for the problems we wish to solve. Our restriction on the number of processors is quite severe in comparison to what most scheduling algorithms were designed for, even those that schedule on a bounded number of processors. Finally, our desire for a pure spatial decomposition imposes the constraint that if the task associated with one cell-angle pair is assigned to a particular processor, then the tasks for every other angle and that cell need to be assigned to the same processor. Existing general scheduling algorithms are free to assign these other tasks to other processors, and it is not clear how to impose this constraint on them. For these reasons we elect to develop a new algorithm tailored specifically for our problem.

We note that the KBA algorithm satisfies the above constraints, although it can only be used with certain meshes. Given its success at achieving high efficiencies in these special cases, however, we would like to determine whether it could be generalized to unstructured meshes. Our goal in the rest of this section is to develop such a generalized sweep algorithm.

Overview of the New Algorithm

Earlier we observed that KBA scheduling could be viewed as two separate phases: a partitioning phase and an ordering phase. We will adopt this strategy since it simplifies the problem somewhat. Nevertheless, both subproblems are still challenging for unstructured meshes.

Our analysis of structured meshes shows that for scalable sweeps on these meshes a decomposition similar to KBA's is necessary. In order to scalably use $O(N^2)$ processors with unstructured meshes we expect that we will need a decomposition that is a generalization in some sense of the KBA structured decomposition. This assumes that the sweep graphs from unstructured meshes share some basic properties with the graphs from structured meshes (e.g. $O(N^3)$ cells yield $O(N)$ graph levels). Unfortunately, the KBA decomposition (and any generalization to unstructured meshes) differs from the partitions produced by established or "traditional" partitioning approaches. Existing partitioning algorithms attempt to minimize the "edgecut" (the number of element faces on partition boundaries), which yields decompositions that are three-dimensional in character rather than the two-dimensional character of the KBA decomposition. Developing a new partitioning approach for unstructured meshes that duplicates KBA properties is a very difficult problem; our attempts so far have been largely unsuccessful.

Rather than attempting to address all scalability issues such as partitioning needs in our current study, we opt to strive for a more modest goal initially. We will make use of established partitioning algorithms and develop special ordering algorithms that yield relatively high efficiencies on a small or intermediate (≤ 100) number of processors. Two factors lead us to believe that we can achieve

high efficiencies with a non-KBA-like decomposition, assuming we have a KBA-like ordering and a modest number of processors. First, no matter how we perform the partitioning we will have $O(1)$ processors along any given axis when we use $O(1)$ processors rather than the maximum $O(N^2)$ processors; this satisfies the requirements of our analysis of structured meshes. Second, the number of quadrature directions will generally be comparable to (or even greater than) the number of processors, so pipelining of the directions should yield meaningful efficiency gains. One advantage of this approach is that one could simply use the decomposition provided by another application when it is coupled to a transport code. Another advantage is that, assuming some good heuristics are developed for the ordering phase, we can easily use more optimal partitions as they become available to achieve greater efficiencies.

Before we discuss the ordering phase we will list some additional assumptions and constraints. Like KBA we will assume that all vertex weights (task computation times) are equal for the sake of simplifying the problem; this should be true if the same spatial differencing is used throughout the problem and if certain effects like cache misses can be ignored. In addition, we will assume that communication costs are small in comparison to computational costs. Our reasoning is that in practice there is much more overhead involved in executing an unstructured mesh code than a structured mesh code, so relatively expensive spatial differencings will be used in unstructured mesh codes to amortize these extra costs. Thus we expect that in parallel unstructured mesh codes the computational costs are more likely to dominate communication costs. We will, however, introduce some KBA-like features to our ordering algorithm to mitigate the effects of non-zero communication costs.

Our approach for the ordering phase is to use a list scheduling heuristic. An outline of this algorithm is given in Figure 5. We first assign priorities to every cell-angle pair according to one of several heuristics we will describe later. We then initialize a priority queue for each processor with tasks that are entry nodes in the sweep graphs and that have been implicitly assigned to the processor by the partitioning phase. Note that we differ here from most list scheduling heuristics, which maintain a single global priority queue that is used both for task-to-processor assignment and for task ordering. Next we enter a loop that repeats until all tasks have been completed. For now let us assume that the parameter `maxCellsPerStep` is unity. At the top of the loop we remove a task from the top of the priority queue of each processor (if it is not empty) and perform that task. In doing so we may generate output data that enables other tasks lower in the graph. Let us classify these new tasks according to whether or not they have been assigned to the same processor. Those tasks that are on the same processor as the parent task we immediately place into the priority queue; it is possible to perform them without any communication. We then send any partition boundary data that has been computed to the appropriate processors and receive any data that other processors may send. It is this data that enables the other group of tasks; we now place these tasks into the priority queue. We repeat this process until the sweeps have completed.

We have assumed until now that communication costs are negligible. This of course will not generally be the case in practice. The multiple communication steps of the sweep process could result in numerous tiny messages; when `maxCellsPerStep = 1` the messages will contain data from only one cell. If communication costs are significant we can usually reduce them by decreasing the number of messages and increasing their size through buffering. We can accomplish this by increasing the value of `maxCellsPerStep`. Then each processor may perform up to

```
Assign priorities to every cell-angle pair
Place all initially ready tasks in priority queue
While (uncompleted tasks)
     For i=1,maxCellsPerStep
          Perform task at top of priority queue
          Place new on-processor tasks in queue
     Send new partition boundary data
     Receive new partition boundary data
     Place new tasks in queue
```

Figure 5: Pseudocode for list scheduling.

maxCellsPerStep tasks before sending and receiving data; larger values for maxCellsPerStep will result in larger average message sizes and fewer messages. An increase in maxCellsPerStep in our algorithm is equivalent to increasing the block or "chunk" size in the KBA algorithm. Such increases, however, also have the effect of reducing the PCE, since idle processors have to wait longer on average to receive needed data. We must balance these two effects in any given problem in order to maximize the processor efficiency.

We note that in *Yang and Gerasoulis (1994)* it is shown that the complexity of list scheduling with static priorities is $O(v \log(v))$. This will be the complexity of our overall algorithm if the prioritization heuristics are no more expensive. We believe such complexity is acceptably low.

Prioritization Heuristics

The determining factor in the performance of the list scheduling algorithm is the assignment of priorities to tasks. The priorities represent the relative importance of completing one available task before some other available task. If the prioritization phase is done poorly we will obtain a low PCE, but if it is done well we may obtain a PCE that is near-optimal for the given decomposition and the selected value of maxCellsPerStep. Indeed, any ordering that satisfies the execution model of the list scheduling algorithm (each processor must perform up to maxCellsPerStep tasks, if available, between communication steps) can be encoded by an appropriate assignment of priorities. The goal, then, is to determine prioritization heuristics that maximize the performance of the list scheduling algorithm.

The above problem is a difficult one. We need to determine some measure of the "importance" of completing a task early in the sweep process. We expect that a good prioritization scheme will account for the global structure of the sweep graph as well as the decomposition. However, we also require our scheduling algorithm to have low complexity. Therefore we will search for relatively simple prioritization heuristics that somehow encode at least some global information that can impact performance.

For some insight into this problem we revisit the KBA algorithm. KBA scheduling does not use any explicit prioritization scheme, but one could duplicate its efficient ordering by means of a list scheduling technique with some set of priorities. Given this conceptual "list scheduled" KBA algorithm, we would like to understand what types of prioritizations yield the KBA ordering,

what their common characteristics are, and how we can generalize them to an arbitrary mesh and partitioning.

We observed earlier that one property of the KBA algorithm is that each processor performs tasks in such an order that it generates partition boundary data in as few steps as possible. Obviously if other processors are idle then they can begin performing tasks earlier if data for which they are waiting is delivered to them earlier, which will improve the efficiency of the parallel calculation. There are other observations we can make about KBA. When a processor has two or more tasks from the same sweep direction that reside at different levels of the sweep graph and that are ready to be solved, the ordering resulting from selecting the highest-level task will roughly approximate the KBA ordering. Furthermore, the partition boundary data that a processor generates at any given step in the sweep is input data for tasks on other processors that are generally higher in the graph than other remaining tasks for that direction. Uncompleted tasks that are lower in the graph may be descendants of uncompleted tasks higher in the graph, in which case a processor could not begin calculations until it had the input data for the higher level tasks.

The situation for general graphs is, of course, more complex. It may not be possible to maintain several properties such as those listed above simultaneously. Furthermore, we may encounter graph characteristics in unstructured meshes that are never observed in structured meshes. We will need to develop heuristics that account for these various cases. In order to keep the algorithmic complexity low we will need to make simplifying assumptions and compromises.

One simplifying assumption we will make is that the only information a processor needs about parts of the graph assigned to other processors are the b-levels of tasks to which it sends data [Kwok and Ahmad, 1999]. The b-levels encode information about the global graph in a local manner and represent the maximum time to solve a task and its descendants, given unlimited processors. Our prioritization heuristics will generally attempt to preferentially generate input data for off-processor tasks with higher b-levels than other tasks. We note that the b-levels for all tasks can be computed in $O(v+e)$ time. Therefore this assumption should help produce low-complexity algorithms that do take into account global graph information that can impact performance.

Given the above assumptions and observations we have developed several prioritization heuristics. Details about these heuristics are available in *Pautz, (2000)*. The new heuristics we have developed are the BFDS, the DFDS, and the DFHDS heuristics. For comparison we also use the "random" heuristic, in which all priorities are randomly assigned, and the b-level heuristic, which is used in some existing general scheduling algorithms.

We note that all of the above heuristics can be implemented with algorithms with complexity equal to or less than $O(v+e)$, so we have satisfied our requirement for low-complexity heuristics. With the exception of random priorities all of the heuristics use information about the global graph as encoded in the b-levels, as desired. Finally, our new heuristics use information about the graph partition, which we also expect to help improve parallel efficiencies.

In summary, our sweep scheduling algorithm will initially use a traditional approach for generating a spatial domain decomposition. Although this limits our scalability, we hope to obtain high efficiencies on a modest number of processors, and we allow for the use of better decompositions in the future. We will order the sweep tasks by means of a list scheduling algorithm. This algorithm will use one of several prioritization schemes we have developed to order the tasks; these

Table 1: Meshes used in sweep studies.

Mesh	Number of elements	Description
adjalum	4097	Simple two-region box for regression testing.
nneut	43012	Neutron well-logging tool and surrounding media.
silc	51963	Computer chip and packaging for radiation shielding.
reac	165530	Reactor pressure vessel and surrounding cavity structures.
contest2	768	Cube divided into approximately equal-sized elements.
contest3	6140	Cube divided into approximately equal-sized elements.
contest4	32546	Cube divided into approximately equal-sized elements.
contest5	168356	Cube divided into approximately equal-sized elements.

prioritization heuristics are low-complexity but use information we expect will yield intelligent orderings.

Analysis of the Sweep Scheduling Algorithm

In this section we present theoretical performance estimates for our sweep scheduling algorithm. We will construct sweep schedules for several different meshes using the various prioritization heuristics we developed in the previous section. We will evaluate these heuristics based on the PCEs they produce.

We will use tetrahedral meshes in our studies. The meshes we will use are described in Table 1. These meshes vary in size from several hundred elements to over 160,000 elements. They also vary in their structure. The "contest" meshes consist of elements of approximately the same size, whereas the other ("irregular") meshes have a wider distribution of element dimensions. Note that our scheduling algorithm may be applied to any type of mesh, not just tetrahedral meshes. We will partition these meshes with Metis©, a "traditional" mesh partitioner that attempts to minimize the edge cut (the number of partition boundary faces) while preserving load balance [Karypis and Kumar, 1998].

There are many different combinations of parameters we can study. Our base cases will use an S_8 level-symmetric quadrature and a value of 50 for `maxCellsPerStep`. We feel that these choices should be representative of many calculations.

Our calculation of the PCE is performed as follows. In each computation-plus-communication step of our list scheduling heuristic we determine the maximum number of tasks performed by any processor. Assuming that all processors remain synchronized this is the step length; it will be less than or equal to `maxCellsPerStep`. The sum of the step length over all steps is the parallel computation time, T_p, in the absence of communication costs and assuming that synchronization is preserved. The serial computation time, T_s, is the product of the number of mesh cells and the number of quadrature directions, i.e. the total number of tasks to perform. The PCE is then given by Eq. (2), where we replace η with PCE.

We present the PCEs resulting from list scheduling with different prioritization heuristics as a function of the number of processors in Figures 6 and 7 for the nneut and reac meshes, respectively.

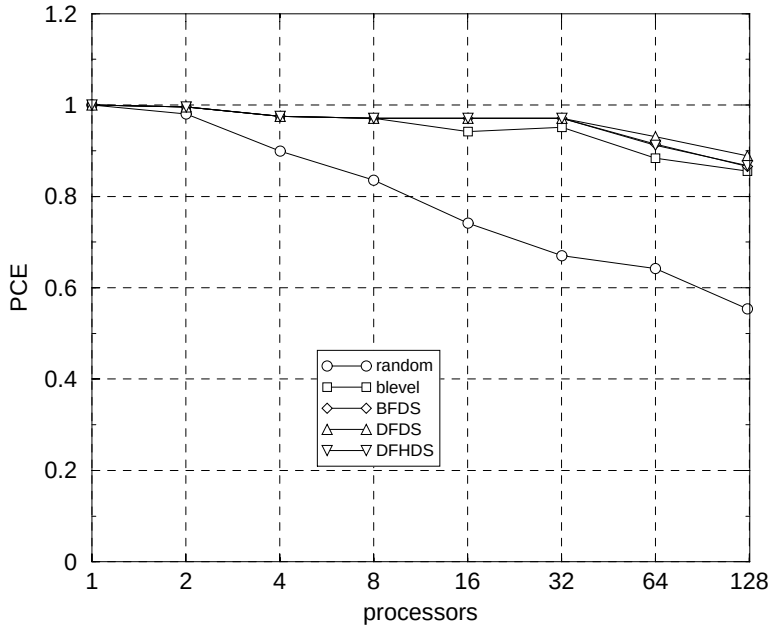


Figure 6: PCE vs. processors for nneut mesh (S_8 quadrature, `maxCellsPerStep` = 50).

There are several characteristics common to these and other studies we have conducted. The random heuristic results in noticeably worse schedules than the other heuristics, demonstrating the need for (and our ability to develop) intelligent prioritization schemes. The b-level heuristic also results in PCEs somewhat lower than the others we have developed, although it is still better than the random heuristic. The remaining heuristics (BFDS, DFDS, and DFHDS) yield very similar performance, with DFDS producing slightly better schedules than the other two. In general, for a small number of processors we can obtain schedules yielding nearly 100% efficiency; this efficiency gradually degrades to 80-90% as we increase the number of processors to over 100. Since we are not using specialized decompositions we expect continued degradation of the efficiency as we increase the number of processors further.

Next we demonstrate the effects of quadrature order on the PCE. In Figure 8 we plot the highest PCEs we could obtain (using all combinations of heuristics) for various quadrature orders for the nneut mesh. For this test we use `maxCellsPerStep = 1`; higher values tend to obscure the trends for this mesh. From this plot we see that increasing the quadrature order almost always increases the PCE; our scheduling algorithm takes advantage of some pipelining effects.

The final theoretical behavior we will explore is the effect of the value of `maxCellsPerStep` on the PCE. In Figure 9 we plot the best PCEs we could obtain for each of several values of `maxCellsPerStep` for the nneut mesh and S_8 quadrature. We see that an increase in `maxCellsPerStep` leads to some reduction in the PCE, sometimes large. If the value of `maxCellsPerStep` is small in comparison to the number of elements per partition, then the degradation in PCE will be small when `maxCellsPerStep` is altered. Hence we expect that for large meshes and/or a relatively small number of processors we can use fairly large values for `maxCellsPerStep` without substantially degrading the theoretical performance of the sweeps.

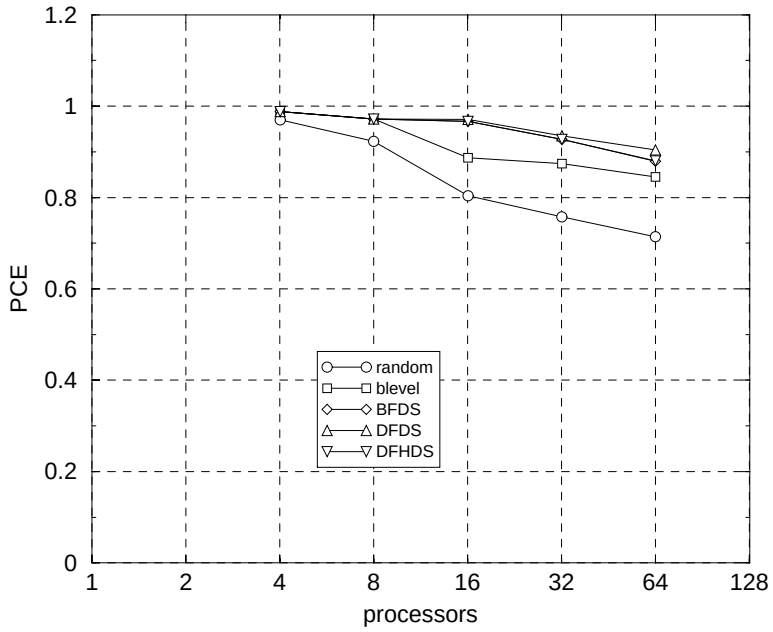


Figure 7: PCE vs. processors for reac mesh (S_8 quadrature, `maxCellsPerStep` = 50).

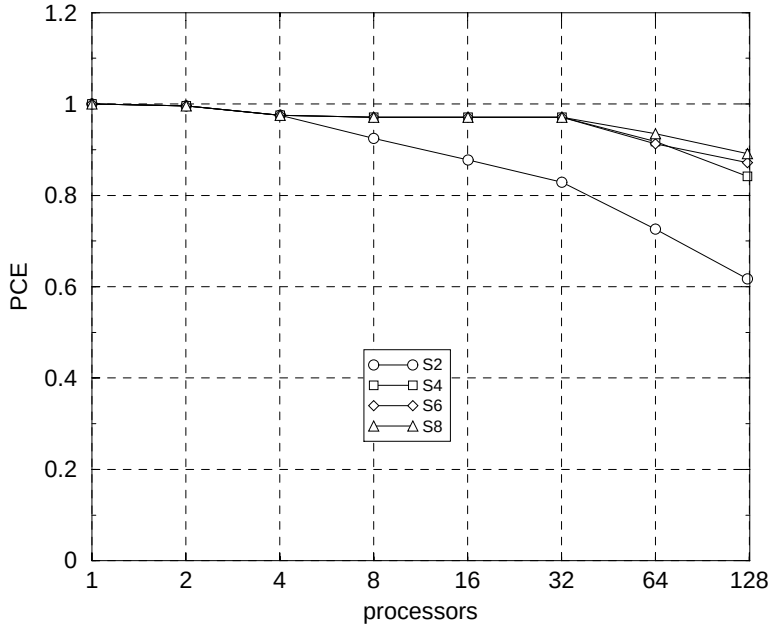


Figure 8: PCE vs. processors for nneut mesh (`maxCellsPerStep` = 1, best schedules).

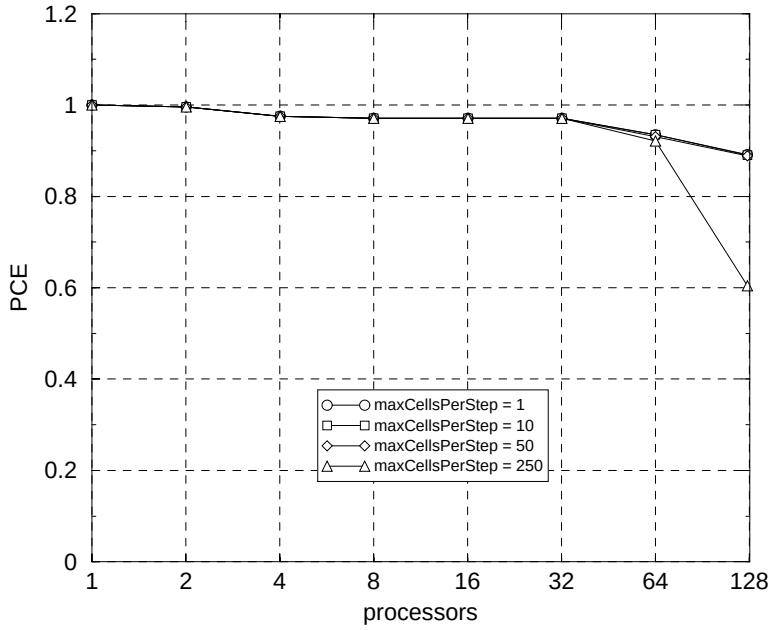


Figure 9: PCE vs. processors for nneut mesh (S_8 quadrature, best schedules).

In summary, our theoretical analyses reveal that our new heuristics (BFDS, DFDS, and DFHDS) should perform quite well, yielding fairly high PCEs on a variety of meshes and with about 100 processors or less. The DFDS scheme consistently performs slightly better than the other schemes. Our algorithm is able to obtain some pipelining effect as quadrature orders are increased. Finally, depending on the size of the mesh and the number of processors, we can increase the value of `maxCellsPerStep` from its "ideal" value of unity without decreasing the PCE significantly. This may prove useful when we consider how to reduce communication costs in a real calculation.

Computational Results

In this section we report actual run-time results for a parallel transport code that uses our sweep scheduling algorithm. We describe the major trends and compare them to the theoretical predictions from the previous section. Where the predictions differ from the computational results we offer possible explanations.

We have implemented our sweep scheduling algorithm in a new parallel S_n code, Tycho, which is under development at Los Alamos National Laboratory (LANL). Tycho currently uses linear discontinuous finite element differencing of the first-order form of the transport equation on tetrahedral elements. Its differencing is identical to that used in Attila and Pericles, serial research codes also developed at LANL [Wareing et al., 1996, Morel, 1999]. We have verified with several numerical tests that parallel Tycho calculations yield the same results (to several significant digits) as Attila and Pericles, thus confirming that we have correctly implemented a

parallel sweep capability.

Our timing studies were conducted on Blue Mountain, a large cluster of SGI Origin2000 (O2K) computational servers at LANL. Each O2K, or "box", consists of 128 250 MHz R10000 processors organized as a non-uniform shared memory machine; each processor has a local memory that is directly accessible by any other processor in the box [Rafiei, 1998]. Instead of exploiting the shared memory capability of the machine by means of multithreading, we have elected to create a separate process for each processor and to share data with explicit MPI calls; we use a vendor supplied version of MPI. In our studies we will use up to one full box, with the exception of two processors kept in reserve for system overhead.

In each of our timing studies we will perform at least five complete source iterations, more if the problem is small enough. We note that the measured CPU time to invert the streaming-plus-collision operator for a single cell-angle pair during a sweep in our code is about 40 μs, which is several times higher than the same calculation in Attila or Pericles; we expect to reduce this computational cost in the future with coding optimizations. Since we are addressing the general question of efficiently parallelizing sweeps on unstructured meshes and not specifically the optimizing of a particular code, we will not concern ourselves at present with any potential shifts in efficiency curves that result from future modifications to Tycho.

We present the measured computational efficiencies for calculations on the nneut and reac meshes in Figures 10 and 11, respectively. These calculations use S_8 quadrature and `maxCellsPerStep = 50`; they correspond to the problems examined in Figures 6 and 7. There are general similarities between the PCEs we calculated and the measured efficiencies. The random heuristic yields noticeably lower efficiencies both in theory and in practice. We also observe relatively high efficiencies for a small number of processors and a reduction in efficiencies when much larger numbers of processors are used. There are also some noticeable differences between our theoretical predictions and these computational results. For a small number of processors we observe superlinear scaling effects, especially for large problems. This we attribute to the amount of memory required and the architecture of the O2K; memory allocated by a processor probably has greater locality as the number of processors is increased, since each processor is assigned a smaller part of the problem. For large numbers of processors the measured efficiency curves drop off more rapidly than the predicted efficiencies, probably because communication costs become increasingly severe as we increase the number of processors. Also, we observe that the b-level heuristic appears to yield efficiencies as good as or sometimes even better than our specialized heuristics, despite our predictions that it would produce slightly less optimal schedules. We do not yet understand this behavior, but we note that the variance in the run-time results seems to be larger than the difference between our predictions for these heuristics. Overall, though, it appears that our PCE calculations provide a reasonable guide to the performance characteristics of our parallel sweep scheduling heuristics.

The next result we will present shows the effect of varying `maxCellsPerStep`. In Figure 12 we show the run-time efficiencies of calculations on the nneut mesh. The parameters used in these calculations are the same as those used to generate Figure 9. There is a clear difference between the behavior of the predicted and measured efficiencies as we vary `maxCellsPerStep`, which is not surprising. An increase in `maxCellsPerStep` leads to some reduction in the PCE, but it can yield marked improvements in the run-time efficiencies, due presumably to reduced communication costs. Only when there is a substantial drop in the PCE as we increase `maxCellsPerStep` do

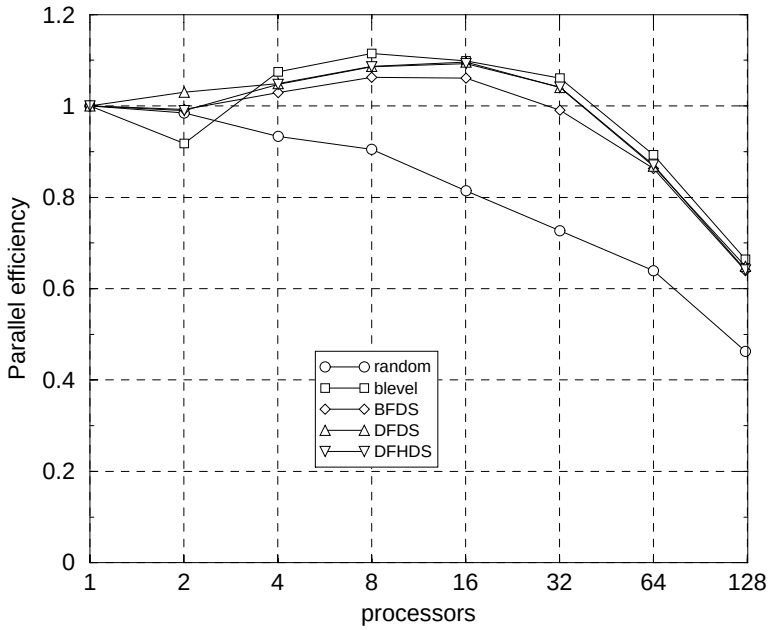


Figure 10: Parallel efficiency vs. processors for nneut mesh (S_8 quadrature, `maxCellsPerStep` = 50).

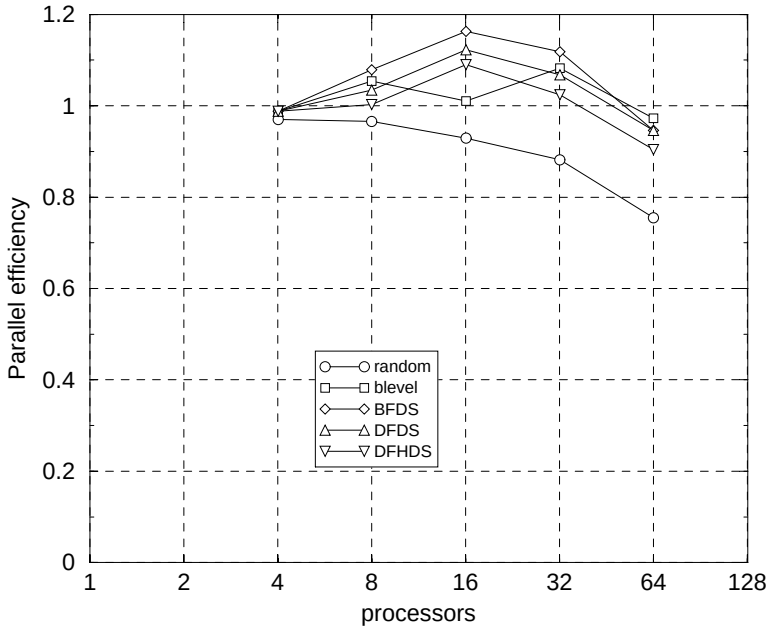


Figure 11: Parallel efficiency vs. processors for reac mesh (S_8 quadrature, `maxCellsPerStep` = 50).

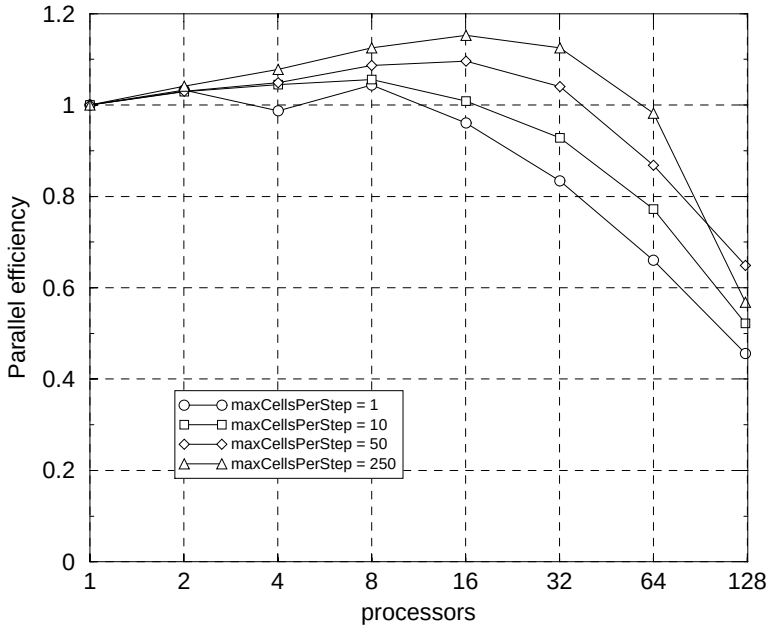


Figure 12: Parallel efficiency vs. processors for nneut mesh (S_8 quadrature, DFDS heuristic).

we observe a drop in the run-time efficiency. To maximize the run-time efficiency for a given problem we recommend using a value for `maxCellsPerStep` as high as possible without severely degrading the PCE.

The final results we will show are scaling studies in the form of log-log timing results. In Figures 13 and 14 we show the CPU time per source iteration for calculations on contest and irregular meshes, respectively. We also show the CPU times that we would obtain with perfectly linear speedups. For these calculations we use S_8 quadrature, `maxCellsPerStep = 50`, and DFDS prioritizations. In these plots we observe almost linear speedups on up to 126 processors, but the changing slopes of the curves suggest that we will lose this scaling on even larger numbers of processors. Thus we see that our approach to partitioning and scheduling is sufficient for low or modest parallelism.

In summary, our performance predictions for our various sweep scheduling heuristics roughly correspond to actual run-time results. With our heuristics we can obtain high efficiencies for a small or modest number of processors, but with a larger number of processors the efficiencies drop off. Some of this loss of efficiency is due to the intrinsic nature of the sweep problem, but much of it in the cases we have examined is caused by the non-ideal aspects of our code and computer system. By increasing the value of `maxCellsPerStep` we can improve the run-time efficiencies, but excessive increases in this parameter eventually decrease the efficiencies. In general, our sweep scheduling algorithms may be used to obtain high speedups on about 100 or fewer processors.

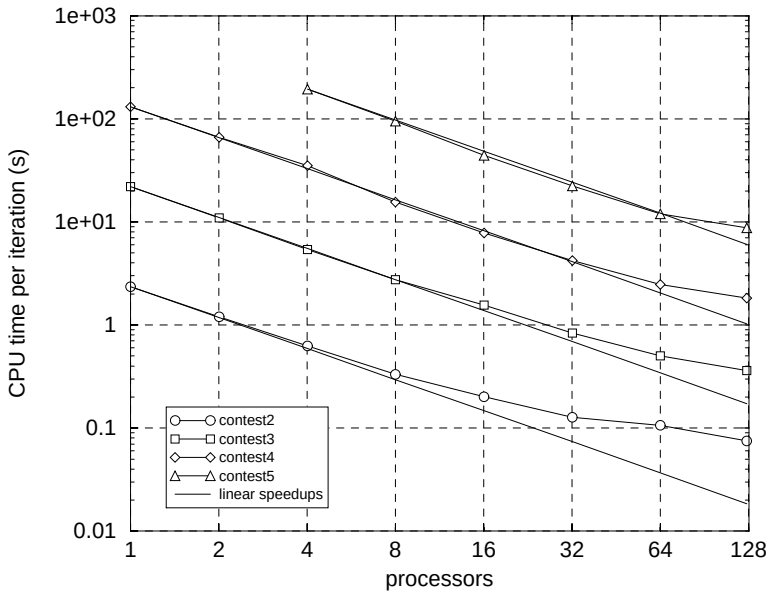


Figure 13: CPU time per iteration vs. processors for contest meshes (S_8 quadrature, `maxCellsPerStep` = 50, DFDS heuristic).

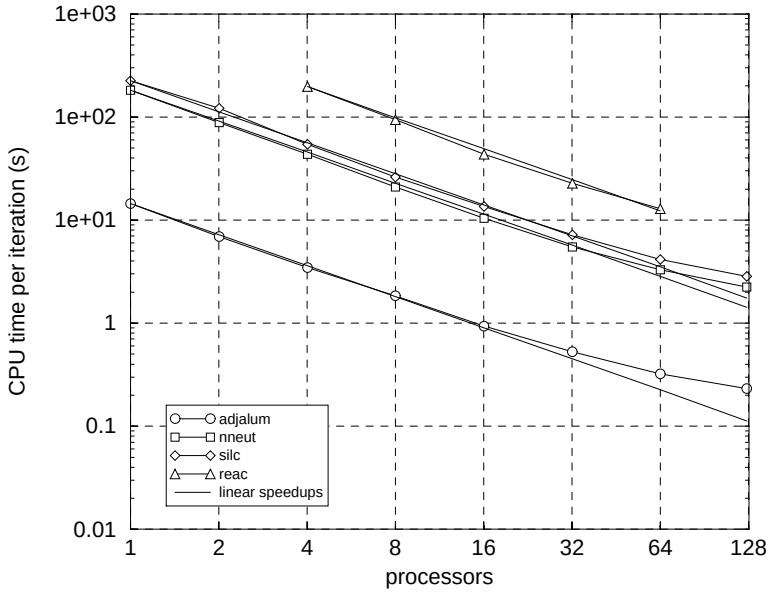


Figure 14: CPU time per iteration vs. processors for irregular meshes (S_8 quadrature, `maxCellsPerStep` = 50, DFDS heuristic).

Conclusions

We have developed a new algorithm for performing parallel sweeps on unstructured meshes. This algorithm uses a low-complexity list scheduling scheme with one of several prioritization heuristics we have developed to determine the parallel sweep ordering on a spatially decomposed unstructured mesh. With this new scheme in conjunction with traditional mesh decompositions we have obtained nearly linear speedups on up to 126 processors. This is an important step in the development of parallel first-order S_n codes.

Our ability to obtain high parallel efficiencies with traditional mesh partitionings is an important result. We have shown that it is impossible to obtain scalable schedules on structured meshes unless the decomposition has a two-dimensional nature; the KBA algorithm has this property. We believe that a similar constraint applies to unstructured meshes. Nevertheless, for typical quadrature orders and for modest levels of parallelism ($\leq$ 100 processors) we can obtain fairly high efficiencies; more processors are apparently needed to observe any severe asymptotic degradation in efficiency.

There are two factors that are key to our algorithm's ability to obtain high theoretical efficiencies for unstructured mesh sweeps. First, we have developed specialized prioritization heuristics that yield much better orderings than a random one; these heuristics attempt to generate and propagate data needed by other processors as rapidly as possible. Second, when the number of quadrature directions is comparable to or greater than the number of partitions we benefit from a pipelining effect. Both of these factors are generalizations of the approach that the KBA algorithm uses to determine the sweep ordering. A third factor, the use of a parameter that delays communications until a "sufficient" amount of computation has been performed, helps to produce run-time efficiencies that are close to the theoretical efficiencies. This too is a generalization of the KBA approach.

There are two main research areas in which we would like to see continuing work. First, we need to develop algorithms that can produce better spatial decompositions. Analysis of structured meshes leads us to believe that decompositions with a two-dimensional or columnar nature will be necessary in order to scale to thousands of processors or more. Second, continued development of prioritization heuristics may produce low-complexity ones that are even better than the ones we have developed. Work with other meshes and better decompositions may yield insights that assist in this process.

References

[Koch et al., 1992] K.R. Koch, R.S. Baker, and R.E. Alcouffe, "A Parallel Algorithm for 3D S_N Transport Sweeps," LA-CP-92-406, Los Alamos National Laboratory (1992).

[Baker et al., 1995] R.S. Baker, C. Asano, and D.N. Shirley, "Implementation of the First-Order Form of the Three-Dimensional Discrete Ordinates Equations on a T3D," *Trans. Am. Nucl. Soc.*, **73**, 170 (1995).

[Baker and Alcouffe, 1997] R.S. Baker and R.E. Alcouffe, "Parallel 3-D S_N Performance for DANTSYS/MPI on the Cray T3D," *Proc. Joint Int. Conf. Mathematical Methods and*

Supercomputing for Nuclear Applications, Saratoga Springs, New York, October 5-9, 1997, Vol. 1, p. 377 (1997).

[Baker and Koch, 1998] R.S. Baker and K.R. Koch, "An S_n Algorithm for the Massively Parallel CM-200 Computer," *Nucl. Sci. Eng.*, **128**, 312 (1998).

[Wareing et al., 1999] T.A. Wareing, J.M. McGhee, J.E. Morel, and S.D. Pautz, "Discontinuous Finite Element S_N Methods on 3-D Unstructured Grids," *Proc. Int. Conf. Mathematics and Computations, Reactor Physics and Environmental Analysis in Nuclear Applications*, Madrid, Spain, September 27-30, 1999, Vol. 2, p. 1185 (1999).

[Wareing et al., 2000] T.A. Wareing, J.M. McGhee, J.E. Morel, and S.D. Pautz, "Discontinuous Finite Element S_N Methods on 3-D Unstructured Grids," *Nucl. Sci. Eng.* (accepted for publication).

[Gerasoulis and Yang, 1992] A. Gerasoulis and T. Yang, "A Comparison of Clustering Heuristics for Scheduling Directed Acyclic Graphs on Multiprocessors," *J. Parallel and Distributed Computing*, **16**, 276 (1992).

[Kwok and Ahmad, 1999] Y.K. Kwok and I. Ahmad, "Benchmarking and Comparison of the Task Graph Scheduling Algorithms," *J. Parallel and Distributed Computing*, **59**, 381 (1999).

[Garey and Johnson, 1979] M.R. Garey and D.S. Johnson, *Computers and Intractibility: A Guide to the Theory of NP-Completeness*, Freeman, San Francisco, California (1979).

[Cormen et al., 1990] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms*, MIT Press, Cambridge, Massachusetts (1990).

[Kumar et al., 1994] V. Kumar, A. Grama, A. Gupta, and G. Karypis, *Introduction to Parallel Computing: Design and Analysis of Algorithms*, Benjamin/Cummings, Redwood City, California (1994).

[Yang and Gerasoulis, 1994] T. Yang and A. Gerasoulis, "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors," *IEEE Trans. Parallel and Distributed Systems*, **5**, 951 (1994).

[Pautz, 2000] S.D. Pautz, "An Algorithm for Parallel S_n Sweeps on Unstructured Meshes," (submitted for publication).

[Karypis and Kumar, 1998] G. Karypis and V. Kumar, "METIS 4.0: Unstructured Graph Partitioning and Sparse Matrix Ordering System," Technical Report, Department of Computer Science, University of Minnesota (1998).

[Wareing et al., 1996] T.A. Wareing, J.M. McGhee, and J.E. Morel, "ATTILA: A Three-Dimensional, Unstructured Tetrahedral Mesh Discrete Ordinates Transport Code," *Trans. Am. Nucl. Soc.*, **75**, 146 (1996).

[Morel, 1999] J.E. Morel, "Deterministic Transport Methods and Codes at Los Alamos," *Proc. Int. Conf. Mathematics and Computations, Reactor Physics and Environmental Analysis in Nuclear Applications*, Madrid, Spain, September 27-30, 1999, Vol. 1, p. 25 (1999).

[Rafiei, 1998] M. Rafiei, "Origin2000 Application Development & Optimization," Technical Report O2KAPPL-1.0-6.2/3/4-S-SD-W, Silicon Graphics, Inc. (1998).

Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy by the University of California Los Alamos National Laboratory under contract No. W-7405-Eng-36.