

# Provable Algorithms for Parallel Sweep Scheduling on Unstructured Meshes

V.S. Anil Kumar

Los Alamos National Laboratory  
MS M997, Los Alamos, NM 87545  
anil@lanl.gov

Madhav V. Marathe

Virginia Bio-informatics Institute and  
Dept. of Computer Science  
Virginia Tech, Blacksburg, VA  
mmarathe@vbi.vt.edu

Srinivasan Parthasarathy<sup>1</sup>

Department of Computer Science  
University of Maryland  
College Park, MD 20742  
sri@cs.umd.edu

Aravind Srinivasan<sup>1</sup>

Department of Computer Science and UMIACS  
University of Maryland  
College Park, MD 20742  
srin@cs.umd.edu

Sibylle Züst

Los Alamos National Laboratory  
MS M997, Los Alamos, NM 87545  
sibylle.zuest@alumni.ethz.ch

## Abstract

We present provably efficient parallel algorithms for sweep scheduling on unstructured meshes. Sweep scheduling is a commonly used technique in Radiation Transport problems, and involves inverting an operator by iteratively sweeping across a mesh. Each sweep involves solving the operator locally at each cell. However, each direction induces a partial order in which this computation can proceed. On a distributed computing system, the goal is to schedule the computation, so that the length of the schedule is minimized.

Several heuristics have been proposed for this problem; see [14, 15] and the references therein; but none of the heuristics have worst case performance guarantees. Here we present a simple, almost linear time randomized algorithm which (provably) gives a schedule of length at most  $O(\log^2 n)$  times the optimal schedule for instances with  $n$  cells, when the communication cost is not considered, and a slight variant, which coupled with a much more careful analysis, gives a schedule of (expected) length  $O(\log m \log \log \log m)$  times the optimal schedule for  $m$  processors. These are the first such provable guarantees for this problem. We also design a priority based list schedule using these ideas, with the same theoretical guarantee, but much better performance in practice.

We complement our theoretical results with extensive empirical analysis. The results show that (i) our algorithm performs very well and has significantly better performance guarantee in practice and (ii) the algorithm compares favorably with other natural and efficient parallel algorithms proposed in the literature [14, 15].

## 1. Introduction

Radiation transport methods are commonly used in the simulation and analysis of a wide variety of physical phenomena. As described by Pautz [14], these methods often involve inverting an operator<sup>2</sup>, and a frequently used technique for inverting this operator is called sweep scheduling. During a sweep, the operator is locally solved for each spatial cell in the mesh in a specified order for each direction from a specific set of directions (see Figure 1). Thus the computation on each cell requires knowledge of the incoming fluxes, which are determined by either the boundary conditions, or from *upstream* cells, solved in the previous iteration. From a computational standpoint, this means that each direction induces a set of precedence constraints captured by a dependency graph. An example is shown in Figure 1(a): for instance, in direction  $i$ , we cannot start the computation on cell 6 unless we have completed it on

---

<sup>1</sup> Research Supported in part by NSF Award CCR-0208005.

---

<sup>2</sup> Referred to in [14] as the “streaming-plus-collision”

cell 3. In each direction, the dependency graph is different, but is induced on the same set of mesh elements. As in Pautz [14], without loss of generality, we will assume that the dependency graph in each direction is a directed acyclic graph (DAG).

Thus, the sweep along any direction is essentially the classical precedence constrained scheduling problem (e.g. [8]), where a partial order is defined on the cells (based on the direction), and the computation on the cells must be performed in any order consistent with the partial order. However, when this problem is solved in a distributed system, a very significant constraint that must be satisfied [14] is: each cell must be processed on the same processor along each direction. This constraint makes this problem different from the standard precedence constrained scheduling problem; we call this problem the *Sweep Scheduling Problem*. This problem does not fall into any of the categories defined by Graham et al. [2] for scheduling problems.

Designing efficient algorithms for sweep scheduling on a set of  $m$  parallel processors is challenging because of the constraints mentioned above. In fact, if all the cells in some direction form a chain, the computation has to proceed sequentially. Thus, one can never expect to obtain linear scaling in the efficiency with increasing number of processors, for arbitrary instances. There are two important performance measures: the length of the schedule while ignoring communication costs (the *makespan*), and the total communication cost (the cost of the messages that must be exchanged between processors), and the goal is to obtain a schedule that minimizes both of these. While there are no known algorithms with provable performance guarantees for the sweep scheduling problem, a lot of empirical work has been done. Pautz [14] describes some heuristics, such as *Depth First Descendant Seeking (DFDS)*, which work very well in practice, but has no worst case guarantees.

## 2. Our Contributions

**Analytical Results: Provable Approximation Algorithms.** The main focus of this paper is to develop algorithms for sweep scheduling with provable guarantees, in contrast to existing heuristics such as DFDS [14], which work well on real meshes, but have no worst case guarantees. For our theoretical bounds, we only focus on the problem which ignores the communication costs between processors - even this simplified version generalizes the well known precedence constrained scheduling problem [2], and is  $\mathcal{NP}$ -complete; no provable results are known so far for this simplified case.

We design the *Random Delay* algorithm, and analyze it rigorously to show that it gives an  $O(\log^2 n)$  approximation<sup>3</sup> ( $n$  is the number of mesh elements). We

then show that a modification of this algorithm, coupled with an improved analysis actually leads to an  $O(\log m \log \log \log m)$ -approximation, where  $m$  is the number of processors. To our knowledge, these are the first algorithms with provable performance guarantees; in contrast, for all the heuristics studied by Pautz [14] there are worst case instances, admittedly not mesh like, where the schedule length could be  $\Omega(m)$  times the optimal. We then modify these algorithms to an algorithm that has the same performance guarantee in theory but performs significantly better in practice. While the running time is usually not that crucial, it is interesting to note that our algorithms run in time almost linear in the length of the schedule. All the algorithms we consider assume no relation between the DAGs in different directions, and thus are applicable even to non-geometric instances. In addition, these algorithms are randomized, suggesting the use of randomness for such scheduling problems (for other such instances, see [5]).

**Empirical Performance.** We also study the empirical performance of our algorithms, and since communication cost is very important in practice, we consider two extreme models of communication cost in this paper; we expect the real communication cost to be in between the costs captured by these two extremes. In the first model, we assume that it costs one unit whenever a message has to be exchanged between two processors, even though these messages might be exchanged in parallel. In the second model, we assume that after each step of computation, the amount of time required to do all the communication equals the maximum number of messages any processor has to send (see Section 5 for details). We adapt our algorithms to lower the communication cost by a suitable partitioning. We also compare the performance of our algorithm with the DFDS heuristic [14], and other natural heuristics. Our main results are the following.

1. Although the worst case guarantee we prove is only  $O(\log^2 n)$ , the performance of our algorithms seems to be within a small constant (usually less than 3) times the optimal on all the real mesh instances we tried. The priority based implementation works much better than the basic Random Delays algorithm.
2. Our first algorithm views each cell as a separate block, and chooses a different processor assignment for it. The makespan for this was very low, but the communication overhead (number of interprocessor edges) was very high. We modified it to first compute a partition of the mesh into blocks, and choose a processor for each block; this did not increase the

<sup>3</sup> An approximation factor of  $\alpha$  implies that the length of the schedule obtained by this algorithm is at most  $\alpha$  times the length

of the optimal schedule, for *any* instance of this problem- note that we do not know the value of the optimal solution, but our proof will not require this knowledge.

makespan too much, but the communication overhead came down significantly.

3. For all the real mesh instances we tried, with varying number of directions, block size and processors, the length of our schedule was always at most  $3nk/m$ , where  $n$  is the number of cells,  $k$  is the number of directions and  $m$  is the number of processors. Since  $nk/m$  is the average load, and an absolute lower bound on the length of the optimal scaling, this observation implies that we get linear speedup in performance for up to 128 processors (and in some instance even more).
4. We compare the performance of our algorithm with the DFDS heuristic of Pautz [14], and other natural heuristics. Our algorithm compares very well, in spite of its simplicity. We also observe that combining our random delays technique with some of these heuristics performs even better in practice.

In this work, we only focus on simulations of our algorithms on real meshes, instead of actual implementation on a parallel cluster, in order to determine whether our theoretical algorithms, with provable performance guarantees, can be adapted to match the best heuristics for this problem, and to obtain good lower bounds on the quality of schedules. Because of space limitations, several details and proofs are omitted; the reader is referred to [7] for the full version of the paper.

**Related Work.** Because of its general applicability, there has been a lot of work on the sweep scheduling problem. When the mesh is very regular, the KBA algorithm [6] is known to be essentially optimal. However, when the mesh is irregular, or unstructured, it is not so easy to solve. There are a lot of heuristics, which have worked quite well in practice for a lot of real meshes, for instance by Pautz [14] and Plimpton et al. [15]. However, *none* of the heuristics has been analyzed rigorously, and *worst case* guarantees on their performance (relative to the optimal schedule) are not known. The  $S_n$ -sweeps application has a very high symmetry, arising from the directions being spread out evenly, but in other applications where this problem is relevant, such symmetry might not exist. In such scenarios, it is not clear how the heuristics of Pautz [14] would work.

Scheduling problems in general have a rich history and much work has gone into theoretical algorithmic and hardness results, as well as the design of heuristics tailor-made for specific settings; see Karger et al. [5] for a comprehensive survey of the theoretical work on scheduling. The precedence constrained scheduling problem was one of the first problems for which provable approximation algorithms were designed, and is described as  $P|prec, p, c|C_{max}$  in the notation of Graham et al. [2] who also give a simple  $2 - \frac{1}{m}$  approximation algorithm, for the case of no communication cost. Lenstra and Rinnooy Kan [10] showed that one cannot approximate this

problem better than  $\frac{4}{3}$ , unless  $\mathcal{P} = \mathcal{NP}$ . Very few results are known for the problem in the presence of communication costs. Hoogeveen, Lenstra and Veltman [4] showed that it is  $\mathcal{NP}$ -complete to get an approximation better than  $\frac{5}{4}$ . Munier and Hanen [13] give an algorithm with a performance guarantee of  $\frac{7}{3} - \frac{4}{3m}$  for this version.

### 3. Preliminaries

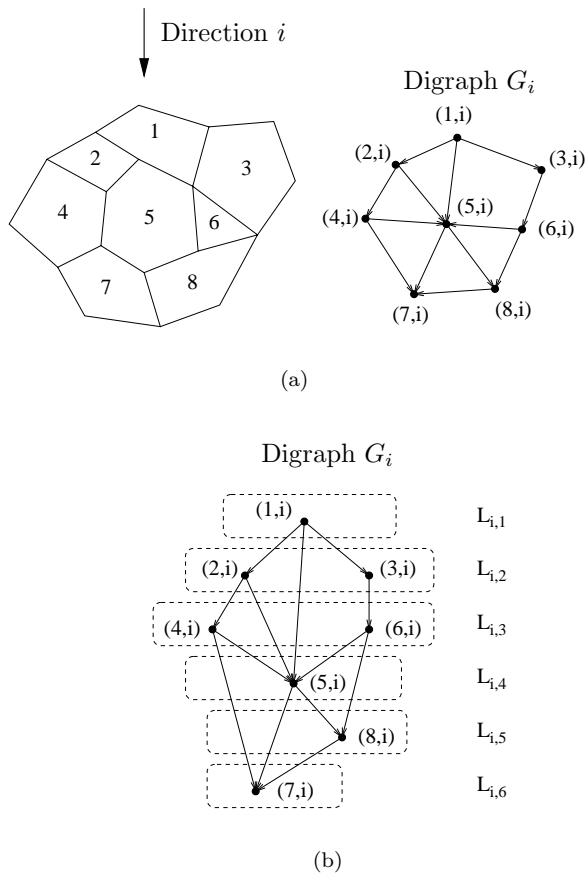
We are given an unstructured mesh  $\mathcal{M}$  consisting of a collection of  $n$  cells, a set of  $k$  directions and  $m$  processors. The mesh induces a natural graph  $G(V, E)$ : cells of the mesh correspond to the vertices and edges between vertices correspond to adjacency between mesh elements. A direction  $i$  induces a directed graph with the vertex set being identical to  $V$ , and a directed edge from  $u$  to  $v$  is present iff  $u$  and  $v$  are adjacent in  $G$  and the sweep in direction  $i$  requires  $u$  to be done before  $v$ . Figure 1(a) illustrates how a digraph is induced in an irregular, 2-dimensional mesh: for example, vertex 5 cannot be solved before its upstream neighbor 2 is solved, which induces a directed edge from 2 to 5 in the corresponding digraph. We assume in the following that the induced digraphs are acyclic (otherwise we break the cycles) and call them *directed acyclic graphs (DAG)*.

Thus, there is a copy of each vertex  $v$  for each direction; we will denote the copy of vertex  $v$  in direction  $i$  by  $(v, i)$  and call this (cell, direction) pair a *task*. The DAG in direction  $i$  will be denoted by  $G_i(V_i, E_i)$ , where  $V_i = \{(v, i) \mid v \in V\}$ .

An *instance* of a sweep scheduling problem is given by a vertex set  $V$  (the cells),  $k$  DAGs  $G_i(V_i, E_i)$ ,  $i = 1, \dots, k$  (the precedence constraints), and  $m$  processors. A *feasible solution* to the sweep scheduling problem is a schedule that processes all the DAGs, so that the following constraints are satisfied.

1. The precedence constraints for each DAG  $G_i(V_i, E_i)$  must be satisfied. That is, if  $((u, i), (v, i)) \in E_i$ , then task  $(u, i)$  must be processed before task  $(v, i)$  can be started.
2. Each processor can process one task at a time and a task cannot be pre-empted.
3. Every copy of vertex  $v$  must be processed on the same processor for each direction  $i$ .

Our overall *objective* is to minimize the computation time of all sweeps subject to the above constraints. We will assume that each task takes uniform time  $p$  to be processed, and there exists a communication cost of uniform time  $c$  between processors. In reality, inter-processor communication will increase the time until all tasks are processed in a way that is hard to model. We will therefore consider the following two objectives separately: (i) the *makespan* of the schedule assuming no



**Figure 1. (a) Example of mesh and digraph induced by direction  $i$ . (b) Levels of the digraph shown in (a).**

communication cost, that is, the time it takes to process all tasks on  $m$  processors according to a certain schedule without taking communication cost into account, and, (ii) the communication cost. For the communication cost, we consider two measures, which represent two extremes (see also Section 5 for details)- the first is the number of interprocessor edges, which is the number of edges  $((u, i), (v, i))$  in all digraphs, for which the tasks  $(u, i)$  and  $(v, i)$  are scheduled on different processors, and the second is the communication delay incurred if after each step of computation, all the processors exchange the messages needed to finish all the communication (this is elaborated in Section 5). Also, note that this model assumes that communication and computation do not overlap, which is clearly a simplifying assumption.

*Levels.* Given  $k$  DAGs  $G_i(V_i, E_i), i = 1, \dots, k$ , we can form levels (also called layers) as follows: for DAG  $G_i(V_i, E_i)$ , layer  $L_{i,j}$  is the set of vertices with no predecessors after vertices  $L_{i,1} \cup \dots \cup L_{i,j-1}$  have been deleted.

We define  $D$  as the maximum number of layers in any direction. In Figure 1(b) we show how levels are formed for the example in Figure 1(a). Note that if we completely process all the cells in one level in arbitrary order before we start processing cells in the next level, we have processed the cells in an order that satisfies the precedence constraints. We will sometimes call a vertex  $(u, i)$  a leaf (or a sink) if the out-degree is 0. Similarly a node with in-degree 0 is called root (or a source).

*List Scheduling.* Throughout the paper, we will use list scheduling at various places. In list scheduling, we may assign a priority to each task. If no priorities are assigned to the tasks, all tasks are assumed to have the same priority. List scheduling can be used after tasks have been assigned to processors or without such an assignment.

A task is said to be *ready*, if it has not been processed yet, but all its ancestors in the dependence graph have been processed. At each timestep  $t$ , let us denote by  $R(t) \subset V \times \{1, \dots, k\}$  the subset of tasks that are ready. We further denote by  $R_P(t) \subset R(t)$  the subset of tasks that are ready and allowed to be processed by processor  $P$ . If nodes have been assigned to certain processors, the subsets  $R_P(t)$  will contain the ready tasks that are assigned to processor  $P$ . If tasks can be processed by an arbitrary processor,  $R_P(t)$  will be equal to  $R(t)$  for all processor  $P$ . The list scheduling algorithm now proceeds such that for each timestep  $t$ , it assigns to each processor  $P$  the task of highest (or lowest) priority in  $R_P(t)$ . Ties are broken arbitrarily. If  $R_P(t)$  is empty, processor  $P$  will be idle at time  $t$ .

## 4. Provable Approximation Algorithms

In this section, we will assume that all processing costs are uniform and there are no communication costs (i.e.,  $p = 1$  and  $c = 0$ ). We first present two randomized approximation algorithms, both with an approximation guarantee of  $O(\log^2 n)$ . The underlying intuition behind both these algorithms is simple and is as follows. We first combine all the DAGs  $G_i$  into a single DAG  $G$  using the “random delays” technique. Next, we assign each vertex to a random processor. Each randomization serves to do contention resolution: the random assignment ensures that each processor roughly gets the same number of mesh elements, the random delay ensures that at each layer of the combined DAG, we do not have too many tasks corresponding to any given cell. Thus the two randomized steps taken together ensure the following property: at a particular level  $l$  of the combined DAG  $G$ , there are “relatively few” tasks to be scheduled in a particular processor. We now expand each level into appropriate time slots to obtain a valid sub-schedule for this level. The final schedule can be constructed by merging the sub-schedules for each of the levels.

In Section 4.3 we present a slightly modified algorithm and a much more careful analysis, which gives an approximation guarantee of  $O(\log m \log \log m)$ .

In Section 5.1, we outline an approach for relaxing the second assumption about the communication costs and obtaining schedules which trade-off communication and processing costs. We note that although our theoretical analysis yields the same approximation guarantee for the first two algorithms presented here, experimental studies indicate that the second algorithm performs significantly better than the first (see Section 5).

#### 4.1. Random Delays Algorithm

---

##### Algorithm 1 Random Delay

---

- 1: For all  $i \in [1, \dots, k]$ , choose  $X_i \in \{0, \dots, k-1\}$  uniformly at random.
  - 2: Form a combined DAG  $G$  as follows:  $\forall r \in \{1, \dots, D+k-1\}$ , define  $L_r = \bigcup_{\{i: X_i < r\}} L_{i, r-X_i}$ . The edge  $((u, i), (v, i))$  is present in  $G$ , if and only if there exists an edge  $((u, i), (v, i))$  in  $G_i$ .
  - 3: For each vertex  $v \in V$ , choose a processor uniformly at random from  $\{1, \dots, m\}$ .
  - 4: Construct a schedule by processing layers  $L_1, L_2, \dots$  sequentially in that order:
    - Layer  $L_{r+1}$  is processed only after all tasks in  $L_r$  have been processed.
    - Within each layer  $L_r$ , process the tasks assigned to each processor in any arbitrary order.
- 

We now present our first algorithm for the sweep scheduling problem, called “Random Delay” (see Algorithm 1). In the first step, we choose a random delay  $X_i$  for each DAG  $G_i$ . In the second step, we combine all the DAGs  $G_i$  into a single DAG  $G$  using the random delays chosen in first step. Recall that  $L_{i,j}$  denotes the set of tasks which belong to the level  $j$  of the DAG  $G_i$ . Specifically, for any  $i$  and  $j$ , the tasks in  $L_{i,j}$  belong to the level  $r$  in  $G$ , where  $r = j + X_i$ . The edges in  $G$  between two tasks are induced by the edges in the original DAGs  $G_i$ : if the edge  $((u, i), (v, i))$  exists in  $G_i$  then it also exists in the combined DAG  $G$ . It is easy to see that all the edges in  $G$  are between successive levels, and all the original precedence constraints are captured by the new DAG  $G$ . The third step involves assigning a processor chosen uniformly at random for each vertex  $v$  (and hence for all its copies in  $G$ ). The fourth and the final step involves computing the schedule. This is done by computing a sub-schedule for each of the layers separately and merging these schedules. Within each layer, the tasks are scheduled using a greedy approach: tasks assigned a particular processor are scheduled in an arbitrary sequence. In the final schedule, all tasks in level

$L_r$  are processed before any task in level  $L_{r+1}$  is processed.

We now analyze the performance of the above algorithm. We first state the following basic facts from probability theory.

**Lemma 1. (The Chernoff-Hoeffding Bound and its variants [1, 3])** Given independent r.v.s  $X_1, \dots, X_t \in [0, 1]$ , let  $X = \sum_{i=1}^t X_i$  and  $\mu = \mathbf{E}[X]$ .

- a. For any  $\delta > 0$ ,  $\Pr[X \geq \mu(1 + \delta)] \leq G(\mu, \delta)$ , where  $G(\mu, \delta) = \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu$ . In particular, for any sufficiently large  $c \geq 0$ ,

$$\Pr[X > c \log n (\max\{\mu, 1\})] < \frac{1}{n^{O(1)}}. \quad (1)$$

- b. There exists a constant  $a > 0$  such that the following holds. Given  $\mu > 0$  and  $p \in (0, 1)$ , suppose a function  $F(\mu, p) \geq \mu$  is defined as follows:

$$F(\mu, p) = \begin{cases} a \cdot \frac{\ln(p^{-1})}{\ln(\ln(p^{-1})/\mu)} & \text{if } \mu \leq \ln(p^{-1})/e \\ \mu + a \cdot \sqrt{\frac{\ln(p^{-1})}{\mu}} & \text{otherwise} \end{cases} \quad (2)$$

Then, defining  $\delta = F(\mu, p)/\mu - 1 \geq 0$ , we have  $G(\mu, \delta) \leq p$ ; in particular,  $\Pr[X \geq F(\mu, p)] \leq p$ .

Let  $\mathcal{S}$  be the schedule produced by our algorithm. In the following analysis, unless otherwise specified, level  $L_r$  refers to level  $r$  of DAG  $G$ . The following lemmas follow by the application of the Chernoff-Hoeffding bounds stated in Lemma 1.

**Lemma 2.** For all  $v \in V$ , and for each layer  $L_r$ , with high probability, the number of copies of  $v$  in  $L_r$  is at most  $\alpha \log n$  with high probability, where  $\alpha > 0$  is a constant. Specifically, this probability is at least  $1 - \frac{1}{n^\beta}$ , where  $\beta$  is a constant which can be made suitably large by choosing  $\alpha$  appropriately.

For each layer  $L_r$ , define the set  $V_r = \{v \mid \exists i \text{ such that } (v, i) \in L_r\}$ . The following lemma holds.

**Lemma 3.** For any level  $L_r$  and any processor  $P$ , the number of tasks that are assigned to  $P$  from  $L_r$  is at most  $\alpha' \max\{\frac{|V_r|}{m}, 1\} \log^2 n$  with high probability where  $\alpha' > 0$  is a constant. Specifically, this probability is at least  $1 - \frac{1}{n^{\beta'}}$ , where  $\beta'$  is a constant which can be made suitably large by choosing  $\alpha'$  appropriately.

**Lemma 4.** Let  $OPT$  denote the length of the optimal schedule. Schedule  $\mathcal{S}$  has length  $O(OPT \log^2 n)$  with high probability.

*Proof.* Let  $R$  be the number of levels in  $G$ . Lemma 3 implies that any level  $L_r$  has a processing time of  $O(\max\{\frac{|V_r|}{m}, 1\} \log^2 n)$  with high probability. Hence, the total length of schedule  $\mathcal{S}$  is at most

$$\sum_{r=1}^R O((\max\{\frac{|V_r|}{m}, 1\} \log^2 n)) \leq \sum_{r=1}^R O((\frac{|L_r|}{m} + 1) \log^2 n),$$

which is  $O((\frac{nk}{m} + R) \log^2 n)$ , where  $R \leq k + D$ . We observe that  $OPT \geq \max\{\frac{nk}{m}, k, D\}$ . Hence the length of schedule  $\mathcal{S}$  is  $O(OPT \log^2 n)$ .  $\square$

**Theorem 1.** *Algorithm 1 computes in polynomial time a schedule  $\mathcal{S}$  which has an approximation guarantee of  $O(\log^2 n)$  with high probability.*

In a schedule produced by Algorithm 1, each layer in  $G$  is processed sequentially. This might result in the following scenario: there may be time instants  $t$  during which a processor  $P$  remains idle, even though there are ready tasks assigned to processor  $P$ . Clearly, idle times needlessly increase the makespan of the schedule. One way to eliminate idle times is to “compact” the schedule obtained through Algorithm 1. We now describe this approach in detail.

## 4.2. Random Delays with Compaction: A Priority based List Schedule

Motivated by the need to eliminate idle times from the schedule, we present Algorithm 2, which is called “Random Delays with Priorities”. Algorithm 2 first defines a priority  $\Gamma(v, i)$  for each task  $(v, i)$  and uses these priorities to create a schedule by list scheduling, as follows: at any given time  $t$ , for any processor  $P$ , among the set of all yet to be processed tasks which are ready and which are assigned to  $P$ , Algorithm 2 schedules the task with the least  $\Gamma$  value. It is easy to see that this algorithm results in a schedule such that there are no idle times. Let  $\mathcal{S}'$  denote the schedule produced by this algorithm. The following theorem gives the performance guarantees on Algorithm 2; its proof is omitted here.

---

### Algorithm 2 Random Delays with Priorities

---

- 1: For all  $i \in [1, \dots, k]$ , choose  $X_i \in \{0, \dots, k-1\}$  uniformly at random.
  - 2: For each task  $(v, i)$ , if it lies in level  $r$  in  $G_i$ , define  $\Gamma(v, i) = r + X_i$ .  $\Gamma(v, i)$  is the priority for task  $(v, i)$ .
  - 3: For each vertex  $v \in V$ , choose a processor uniformly at random from  $\{1, \dots, m\}$ .
  - 4:  $t = 1$ .
  - 5: **while** not all tasks have been processed **do**
  - 6:   **for all** processors  $P = 1, \dots, m$  **do**
  - 7:     (i) Let  $(v, i)$  be the task with lowest priority assigned to  $P$  (i.e.,  $\Gamma(v, i)$  is the smallest) that is ready to be processed (with ties broken arbitrarily).
  - 8:     (ii) Schedule  $(v, i)$  on  $P$  at time  $t$ .
  - 9:   **end for**
  - 10:    $t \leftarrow t + 1$ .
  - 11: **end while**
- 

**Theorem 2.** *Let  $G(V, E)$  be an unstructured mesh, with  $|V| = n$  and  $D_1, \dots, D_k$  be the sweep directions. Let  $OPT$*

*be the length of the optimal schedule and  $m$  be the total number of processors. Algorithm 2 runs in time  $O((mk + nk) \log nk)$  and produces an assignment of mesh elements to the  $m$  processors and a schedule  $\mathcal{S}'$  whose makespan is at most  $O(OPT \log^2 n)$  with high probability.*

## 4.3. An Improved Analysis: An $O(\log m \log \log m)$ -Approximation

We now show that a slight modification of the earlier algorithm, along with a more careful analysis leads to a  $O(\log m \log \log \log m)$ -approximation of the makespan. The new algorithm is called “Improved Random Delay” and is presented in Algorithm 3. In contrast with Theorem 1, which shows a high probability bound, we will only bound the expected length of the schedule. The basic intuition for the improved analysis comes from corollary 2 below: if we consider the standard “balls-in-bins” experiment, the maximum number of balls in any bin is at most the average, plus a logarithmic quantity. The idea now is to consider the scheduling of each layer in the combined DAG as such an experiment. One complication comes from the dependencies - the events that tasks  $(v, i)$  and  $(w, i)$  end up in the same layer in the combined DAG are not independent. This can be mitigated by the preprocessing step, which informally reduces the “width” of the layers, i.e., modifies each DAG such that there are at most  $m$  nodes in each layer.

**Analysis.** For the tighter analysis, we need to look at the time taken to process all the tasks in any layer  $L_t''$ . Due to space limitations, we omit several proofs, and sketch some of them; the full details are available in [7]. Let  $Y_t$  denote the time required to process the tasks in  $L_t''$ . Our main result will be the following.

**Theorem 3.** *For any  $t$ ,  $\mathbf{E}[Y_t] \leq O(\mu_t/m + (\log m) \log \log \log m)$ , where  $\mu_t = \mathbf{E}[|L_t''|]$ .*

Let  $\rho = (\log m) \log \log \log m$ . Theorem 3 implies that we get an  $O(\rho)$ -approximation in expectation, by observing that the makespan  $T$  after the preprocessing step is within a small factor of the optimal.

**Corollary 1.** *Algorithm 3 gives a schedule of expected length  $O(\rho)$  times the optimal.*

We start with some observations on the expected maximum load in a balls-in-bins experiment. Motivated by Lemma 1, we define a function  $H(\mu, p)$ , for  $\mu > 0$  and  $p \in (0, 1)$  as follows; the constant  $C$  will be chosen large enough.

$$H(\mu, p) = \begin{cases} C \cdot \frac{\ln(p^{-1})}{\ln(\ln(p^{-1})/\mu)} & \text{if } \mu \leq \ln(p^{-1})/e; \\ Ce\mu & \text{otherwise.} \end{cases} \quad (3)$$

Note that for any fixed  $p$ ,  $H$  is continuous and has a valid first derivative for all values of  $\mu$  - to see this, we just need to check these conditions for  $\mu = \ln(p^{-1})/e$ .

---

**Algorithm 3** Improved Random Delay
 

---

- 1: **Preprocessing:** Construct a new set of levels  $L'_i$  for each direction  $i$  in the following manner.
  - First construct a new DAG  $H(\cup_i V_i, \cup_i E_i)$  by combining all the  $G_i$ 's, and viewing all the copies  $(v, i)$  of a vertex  $v$  as distinct.
  - Run the standard greedy list scheduling algorithm on  $H$  with  $m$  identical parallel machines [2]; let  $T$  be the makespan of this schedule.
  - Let  $L'_{ij} = \{(v, i) \in V_i | (v, i) \text{ done at step } j \text{ of above schedule}\}$ .
- 2: For all  $i \in [1, \dots, k]$ , choose  $X_i \in \{0, \dots, k-1\}$  uniformly at random.
- 3: Form a combined DAG  $G''$  as follows:  $\forall r \in \{1, \dots, T+k-1\}$ , define  $L''_r = \bigcup_{\{i: X_i < r\}} L'_{i, r-X_i}$ . The edge  $((u, i), (v, i))$  is present in  $G''$ , if and only if there exists an edge  $((u, i), (v, i))$  in  $G_i$ .
- 4: For each vertex  $v \in V$ , choose a processor uniformly at random from  $\{1, \dots, m\}$ .
- 5: Construct a schedule by processing layers  $L''_1, L''_2, \dots$  sequentially in that order:
  - Layer  $L''_{r+1}$  is processed only after all tasks in  $L''_r$  have been processed.
  - Within each layer  $L''_r$ , process the tasks assigned to each processor in any arbitrary order.

---

**Corollary 2.** (a) If we fix  $p$ , then  $H(\mu, p)$  is a concave function of  $\mu$ . (b) Suppose the constant  $C$  in the definition of  $H$  is chosen large enough. Then, if we assign some number  $t$  of objects at random to  $m$  bins, the expected maximum load on any bin is at most  $H(t/m, 1/m^2) + t/m$ .

**Lemma 5.** For any constant  $a \geq 3$ , the function  $\phi_a(x) = x^a e^{-x}$  is convex in the range  $0 \leq x \leq 1$ .

*Proof of Theorem 3:* Fix  $t$  arbitrarily. For  $j \geq 0$ , let  $Z_j = \{v | \{(v, i) \in L''_t\} \in [2^j, 2^{j+1})\}$ , i.e.,  $Z_j$  is the set of nodes  $v$  such that the number of copies of  $v$  that end up in layer  $L''_t$  lies in the range  $[2^j, 2^{j+1})$ . We first present some useful bounds on  $\mathbf{E}[|Z_j|]$  and on  $\mu_t$ , whose proofs are omitted.

**Lemma 6.** (a)  $\sum_{j \geq 0} 2^j \mathbf{E}[|Z_j|] \leq \mu_t$ ; and (b)  $\mu_t \leq m$ .

**Lemma 7.** For  $j \geq 2$ ,  $\mathbf{E}[|Z_j|] \leq (e/2^j)^{2^j} \cdot \mu_t$ .

Now, consider step (3) of Algorithm 3, and fix  $Z_j$  for some time  $t$ . Next, schedule the jobs in  $Z_j$  in the following manner in step (5) of the algorithm: we first run all nodes in  $Z_0$  to completion, then run all nodes in  $Z_2$  to completion,  $\dots$ . Clearly, our actual algorithm does no worse than this. Recall that we condition on some given values  $Z_j$ . We now bound the expected time to process all jobs in  $Z_j$ , in two different ways (this expectation is only w.r.t. the random

choices made by  $P_1$ ): (a) first, by Corollary 2, this expectation is at most  $2^{j+1} \cdot (H(|Z_j|/m, 1/m^2) + |Z_j|/m)$ ; and (b) trivially, this expectation is at most  $2^{j+1} \cdot |Z_j|$ . Thus, conditional on the values  $Z_j$ , the expected makespan for level  $t$  is:  $\mathbf{E}[Y_t | (Z_0, Z_1, \dots)] \leq [\sum_{j=0}^{\ln \ln m} 2^{j+1} \cdot (H(|Z_j|/m, 1/m^2) + |Z_j|/m)] + [\sum_{j > \ln \ln m} 2^{j+1} \cdot |Z_j|]$ , and therefore,  $\mathbf{E}[Y_t] \leq [\sum_{j=0}^{\ln \ln m} 2^{j+1} \cdot (\mathbf{E}[H(|Z_j|/m, 1/m^2)] + \mathbf{E}[|Z_j|/m])] + [\sum_{j > \ln \ln m} 2^{j+1} \cdot \mathbf{E}[|Z_j|]] \leq [\sum_{j=0}^{\ln \ln m} 2^{j+1} \cdot (H(\mathbf{E}[|Z_j|/m, 1/m^2] + \mathbf{E}[|Z_j|/m]) + \mathbf{E}[|Z_j|])] + [\sum_{j > \ln \ln m} 2^{j+1} \cdot \mathbf{E}[|Z_j|]]$ , since  $H$  is concave by Corollary 2(a). (We are using Jensen's inequality: for any concave function  $f$  of a random variable  $T$ ,  $\mathbf{E}[f(T)] \leq f(\mathbf{E}[T])$ .) Consider the first sum in the last inequality above. By Lemma 6(a), the term " $\sum_{j=0}^{\ln \ln m} 2^{j+1} \cdot \mathbf{E}[|Z_j|/m]$ " is  $O(\mu_t/m)$ . Next, we can see from (3) that if  $p$  is fixed, then  $H(\mu, p)$  is a non-decreasing function of  $\mu$ . So, Lemmas 6 and 7 show that there is a value  $\alpha \leq O((\ln m)/\ln \ln m)$  such that  $H(\mathbf{E}[|Z_j|/m, 1/m^2]) \leq \alpha$  for  $j = 0, 1$ . So,  $\sum_{j=0}^{\ln \ln m} 2^{j+1} \cdot H(\mathbf{E}[|Z_j|/m, 1/m^2]) \leq O(\alpha) + \sum_{j=2}^{\ln \ln m} 2^{j+1} \cdot H(\mathbf{E}[|Z_j|/m, 1/m^2]) \leq O(\alpha) + \sum_{j=2}^{\ln \ln m} 2^{j+1} \cdot H((e/2^j)^{2^j}, 1/m^2) \leq O(\alpha) + O\left(\sum_{j=2}^{\ln \ln m} 2^{j+1} \cdot \frac{\ln m}{\ln \ln m + j2^j}\right)$ . The second inequality above follows from Lemmas 6(b) and 7. We split this sum into two parts. As long as  $2^j \leq \ln \ln m / \ln \ln \ln m$ , the term " $\frac{\ln m}{\ln \ln m + j2^j}$ " above is  $\Theta(\frac{\ln m}{\ln \ln m})$ ; for larger  $j$ , it is  $\Theta(\frac{\ln m}{j2^j})$ . Thus, the sum in the first part is dominated by its last term, and hence equals  $O((\log m)/\log \log \log m)$ . The sum in the second part is bounded by

$$O\left(\sum_{j=2}^{\ln \ln m} (\ln m)/j\right) = O((\log m) \cdot \log \log \log m).$$

Summarizing, the first sum above is  $O(\mu_t/m + (\log m) \log \log \log m)$ . Now consider the second sum above. Recalling Lemma 7, we get

$$\sum_{j > \ln \ln m} 2^{j+1} \cdot \mathbf{E}[|Z_j|] \leq \mu_t \cdot \sum_{j > \ln \ln m} 2^{j+1} \cdot (e/2^j)^{2^j} = O(\mu_t/m), \quad (4)$$

since the second sum in the earlier expression for  $E[Y_t]$  is basically dominated by its first term. This completes the proof of Theorem 3.

## 5. Experiments with Random Delay Algorithms

We implemented the algorithms described in Section 4.1 and 4.2 and tested them on various meshes. All of the meshes used were unstructured tetrahedral meshes, varying in form and size. The smallest mesh, called `tetonly`, has 31481 cells, the mesh called `well_logging`

has 43012 cells, the mesh called `long` has 61737 cells, and the largest mesh, called `prismtet`, has 118211 cells. Throughout our experiments, the qualitative results have been very similar across the different meshes. We therefore usually show plots for one mesh only.

*Objective functions.* As mentioned earlier, we will *simulate* the sweeps, instead of actually running them on a distributed machine, and identify machine independent optimizations. For the makespan, we only compute the number of steps taken, in the absence of communication. For the communication cost, we consider two different measures. The first (denoted by  $C_1$ ) is a static measure of the communication cost: the total number of edges that go across processors, for the given processor assignment. For the second measure (denoted by  $C_2$ ), we assume that after each step of computation, there is a round of communication, and the amount of time taken for this equals the maximum number of messages any processor has to send (or, the maximum degree). Performing all the communication within this time is not trivial, and requires some extra coordination. One way this can be done in a distributed manner is to use an edge coloring algorithm, such as [11]. Note that  $C_2$  is a very optimistic measure- in reality, it might take much more time to perform the communication than the maximum degree.

*Lower Bound of the Makespan.* The makespan cannot be lower than  $\frac{nk}{m}$ , the number of tasks divided by the number of processors. In most of our experiments, we compare the makespan to this lower bound.

## 5.1. Approximation Guarantees in Practice

In Section 4 we showed that the random delays approach can be shown to yield an  $O(\log^2 n)$  approximation guarantee. We now study its performance in practice. We implemented the initial random delays heuristic, and ran it on the tetrahedral meshes described above. Our main observations are:

1. The random delays algorithm gives a much better performance guarantee than  $O(\log^2 n)$ , when compared to the lower bound (Figure 2(a)). But the communication cost turns out to be very high: the fraction of edges that are on different processors is more than  $\frac{m-1}{m}$ .
2. Instead of choosing a processor for each cell, if we first partition into blocks, and choose a processor for each block, the number of interprocessor edges,  $C_1$ , comes down significantly, while the makespan does not increase too much (Figure 2(b)); also cost  $C_2$  seems to behave differently- it seems to be much smaller than  $C_1$ , and does not seem to be affected significantly with the block partitioning.
3. In practice, the “Random Delays with Priorities” algorithm performs much better than the original

“Random Delays” algorithm, especially when we use a higher number of processors, giving an improvement of up to a factor of 4 (Figure 2(c)). Also, in all our runs, the total makespan was at most  $3nk/m$ . Since  $nk/m$  is a lower bound on the optimal makespan, this shows a good scaling with the number of processors, even up to 500 processors.

*Partitioning into Blocks.* In an attempt to bring down the number of interprocessor edges ( $C_1$ ), we partition the cells of the mesh into blocks, using METIS, which is a popular freely available graph partitioning software [12]. Instead of choosing a processor for each individual cell (as in Algorithm 2), we choose the processor for each block. METIS forms these blocks so that the number of crossing edges is small; this automatically reduces  $C_1$ . Also, with increasing block size,  $C_1$  decreases. When we assign blocks of cells randomly to processors, the proof of our theoretical approximation guarantee does not hold anymore. However, in practice we observed that the makespan increases only slightly (Figure 2(a)), while the number of interprocessor edges decreases significantly (Figure 2(b)); also, the cost  $C_2$  seems to be much smaller than the cost  $C_1$ , and does not seem to reduce too much with the block partitioning.

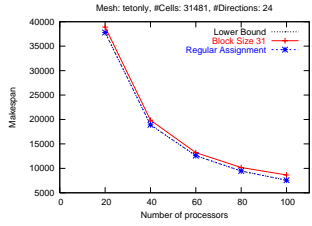
## 5.2. Comparison of Different Algorithms

In this section, we compare the empirical performance of the algorithm “Random Delays with Priorities” with several different heuristics. We show that we get even better performance by combining ideas from these different heuristics. In all of these implementations, we first do the same block assignment (which gives the same number of interprocessor edges, or communication cost ( $C_1$ )); therefore we will compare only the makespans of all the heuristics. We have not yet studied the behavior of the cost  $C_2$  for these instances.

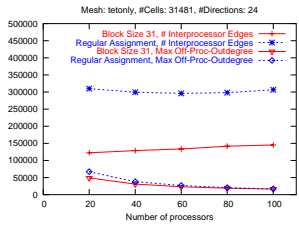
The heuristics consist of different prioritizations of the tasks and optionally giving random delays to directions.

*Priorities.* A priority is given to each task  $(v, i)$ . The schedule will then be computed by using prioritized list scheduling. Examples of different priorities are:

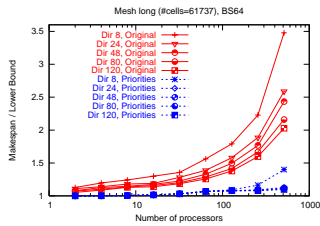
- *Level Priorities:* Let task  $(v, i)$  belong to level  $L_{i,j}$  in DAG  $G_i$ . Then this task will get priority  $j$ . Smaller priorities will be preferred over higher priorities.
- *Descendant Priorities:* This priority scheme is chosen since it is very similar to the one suggested in [15]. Every task  $(v, i)$  gets as its priority the number of descendants it has in DAG  $G_i$ . Higher priorities will be preferred over smaller priorities.
- *Depth-First Descendant-Seeking (DFDS) Priorities:* This prioritization was introduced by Pautz [14]. In



(a)

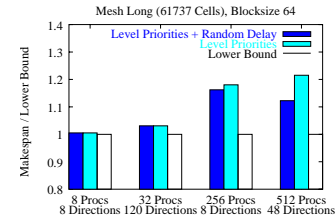


(b)

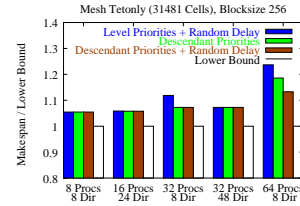


(c)

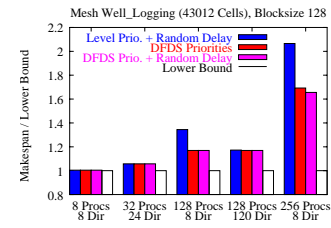
Figure 2. (a)&(b) Random delay scheduling on the mesh tetonly with 24 directions. Plotted is (a) the makespan, and (b) the number of inter-processor edges  $C_1$ , and the cost  $C_2$  (labeled as Max Off-Proc-Outdegree in the plots), for a regular assignment of cells to processors as well as for a block assignment of cells to processors. (c) “Random Delays” versus “Random Delays with Priorities” for different numbers of directions and increasing numbers of processors for mesh long.



(a)



(b)



(c)

Figure 3. Approximation Ratios for various algorithms: (a) The effect of the random delays. (b) Random delays algorithm compared to descendant priorities without and with random delays. (c) Random delays algorithm compared to DFDS priorities without and with random delays.

his paper, the  $b$ -level of a task  $(v, i)$  is the number of nodes in the longest path from  $(v, i)$  to a leaf, so the levels are numbered from bottom up instead of top down as we do it. In the DFDS prioritization, a priority equal to the highest  $b$ -level of its children plus some constant greater than or equal to the numbers of levels in the graph is first assigned to each task with off-processor children. To each task with no off-processor children a priority one unit less than the highest priority of its children is assigned. If a task has no off-processor descendants, it is assigned a priority of zero. Here also higher priorities will be preferred over smaller ones.

*Performance of Level Priorities.* In the bar chart shown in Figure 3(a) we compared our random delays algo-

rithm to the algorithm where we just give level priorities to the tasks, without adding random delays, and then perform list scheduling. We plotted the ratio of the makespan to the lower bound for the mesh `long` with a block partitioning of block size 64. While the algorithms perform equally well for a small numbers of processors independent of the number of directions, the random delays improve the makespan for higher number of processors.

*Performance of Descendant Priorities.* In the bar chart shown in Figure 3(b) we compared our random delay algorithm to the algorithm where we give descendant priorities to the tasks (green) and to the algorithm where we give descendant priorities to the tasks and in addition randomly delay directions (brown). We plotted the ratio of the makespan to the lower bound for the `tetonly` mesh with a block partitioning of block size 256. Also here the random delays algorithm and the descendant priorities algorithm perform equally well for a small numbers of processors independent of the number of directions. For a higher number of processors and a small number of directions, the descendant priorities seem to do better than the random delay algorithm. However, even for a high number of processors, if we use a higher number of directions, the algorithms perform equally well again. Adding a random delay to directions improves the performance of the descendant priorities for a very high number of processors and a low number of directions.

*Performance of DFDS Priorities.* In the bar chart shown in Figure 3(c) we compared our random delay algorithm to the algorithm where we give DFDS priorities to the tasks (red) and in addition to DFDS priorities add random delays (pink). We plotted the ratio of the makespan to the lower bound for the mesh `well_logging` with a block partitioning of block size 128. Once again, our algorithms performs equally well as the DFDS priorities algorithm for a small number of processors. Once we increase the number of processors, the DFDS algorithm has a lower makespan than the random delay algorithm for a low number of directions. For a higher number of directions, they produce the same makespan even for a high number of processors. We can further observe that the random delays do not improve the performance of DFDS priorities except for a high number of processors and a low number of directions.

**Acknowledgments.** We thank Jim Morel (Los Alamos National Laboratory) and Shawn Pautz (Sandia National Laboratory) for introducing the Sweep Scheduling problem to us. We also thank Jim Morel, Shawn Pautz, Jim Warsa, and Kent Budge for valuable discussions throughout the course of this research and also providing us with the meshes used in the experimental analysis. We also thank the anonymous referees for very valuable comments.

## References

- [1] H. Chernoff. A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations. *Annals of Mathematical Statistics*, 23:493–509, 1952.
- [2] R.L. Graham, E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Ann. Discrete Math.*, 5, 287-326, 1979.
- [3] W. Hoeffding. Probability Inequalities for Sums of Bounded Random Variables. *American Statistical Association Journal*, 58:13–30, 1963.
- [4] J.A. Hoogeveen, J.K. Lenstra and B. Veltman. Three, four, five, six, or the complexity of scheduling with communication delays. *Operations Research Letters*, 16, 129-137, 1994.
- [5] D. Karger, C. Stein, J. Wein. Scheduling Algorithms. Chapter in the *Handbook of Algorithms*, CRC Press, 1997.
- [6] K.R. Koch, R.S. Baker and R.E. Alcouffe. A Parallel Algorithm for 3D  $S_N$  Transport Sweeps. Technical Report *LA-CP-92406*, Los Alamos National Laboratory, 1992.
- [7] V.S. Anil Kumar, M. Marathe, S. Parthasaraty, A. Srinivasan and S. Züst. Provable Algorithms for Parallel Sweep Scheduling on Unstructured Meshes, Los Alamos National Laboratory Technical Report LA-UR-04-2811.
- [8] E. Lawler, J. Lenstra, A.H.G. Rinooy Kan and D. Shmoys. Sequencing and Scheduling: Algorithms and Complexity. In S.C. Graves, A.H.G. Rinooy Kan and P.H. Zipkin, ed. *Handbooks in Operations Research and Management Science, Vol 4, Logistics of Production and Inventory*, 445-552, North-Holland, 1993.
- [9] T. Leighton, B. Maggs and S. Rao. Packet Routing and Job Shop Scheduling in  $O(\text{Congestion} + \text{Dilation})$  Steps. *Combinatorica* 14(2): 167-186 (1994).
- [10] J.K. Lenstra and A.H.G. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26, 22-35, 1978.
- [11] M. Marathe, A. Panconesi, L. Risinger. An experimental study of a simple, distributed edge coloring algorithm. *ACM Symposium on Parallel Algorithms and Architectures*, 166-175, 2000.
- [12] METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0 (1995). <http://www-users.cs.umn.edu/~karypis/metis/index.html>
- [13] A. Munier and C. Hanen. An approximation algorithm for scheduling unitary tasks on  $m$  processors with communication delays. Internal Report LITP 12, Université P. et M. Curie.
- [14] S. Pautz. An Algorithm for Parallel  $S_n$  Sweeps on Unstructured Meshes. *Nuclear Science and Engineering*, 140, 111-136, 2002.
- [15] S. Plimpton, B. Hendrickson, S. Burns and W. McLendon. Parallel Algorithms for Radiation Transport on Unstructured Grids, *Super Computing*, 2001.