Provable Algorithms for Parallel Generalized Sweep Scheduling ⋆

V.S. Anil Kumar [a], Madhav V. Marathe [b], Srinivasan Parthasarathy [c,1,*], Aravind Srinivasan [d,1], Sibylle Zust [e]

[a] *Virginia Bioinformatics Institute and Dept. of Computer Science, Virginia Tech, Blacksburg, VA. Email: akumar@vbi.vt.edu*

[b] *Virginia Bioinformatics Institute and Dept. of Computer Science, Virginia Tech, Blacksburg, VA. Email: mmarathe@vbi.vt.edu*

[c] *Department of Computer Science, University of Maryland, College Park, MD 20742. Email: sri@cs.umd.edu*

[d] *Department of Computer Science and UMIACS, University of Maryland, College Park, MD 20742. Email: srin@cs.umd.edu*

[e] *Email: sibylle.zuest@alumni.ethz.ch*

Abstract

We present provably efficient parallel algorithms for *sweep scheduling*, which is a commonly used technique in Radiation Transport problems, and involves inverting an operator by iteratively *sweeping* across a mesh from multiple directions. 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. Due to efficiency and coupling considerations, we have an additional constraint, namely, a mesh cell must be processed on the same processor along each direction. Problems similar in nature to sweep scheduling arise in several other applications, and here we formulate a combinatorial generalization of this problem that captures the sweep scheduling constraints, and call it the *generalized sweep scheduling problem.*

Several heuristics have been proposed for this problem; see (22; 23) and the references therein; but none of these have provable worst case performance guarantees. Here we present a simple, almost linear time randomized algorithm for the generalized sweep scheduling problem that (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. The algorithm can be extended with an additional multiplicative factor in the case when we have inter-processor communication latency, in the models of Rayward-Smith(26), and Hwang et al. (11). Our algorithms are extremely simple, and use no geometric information about the mesh; therefore, these techniques are likely to be applicable in more general settings. We also design a priority based list schedule using these ideas, with the same theoretical guarantee, but much better performance in practice; combining this algorithm with a simple block decomposition also lowers the overall communication cost significantly. Finally, we perform a detailed experimental analysis of our algorithm. Our results indicate that the algorithm compares favorably with the length of the schedule produced by other natural and efficient parallel algorithms proposed in the literature (22; 23).

1 Introduction

Radiation transport methods are commonly used in the simulation and analysis of a wide variety of physical phenomena. In its generality, this process involves computing the propagation of radiation flux across an unstructured mesh, and this can be modeled as solving a discrete form of the Boltzmann transport equation (see (22; 23) for details). As described by Pautz (22) and Plimpton et al. (23), this problem involves inverting an operator [2], and a frequently used technique for inverting this operator is called *Sweep Scheduling (SS)*. 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. Figure 1 shows an instance of a mesh partitioned into cells, and a direction i. The computation on a cell, say cell 4, requires information from either the boundary conditions (along edge $\bar{ab}$) or from other cells "upstream" of this cell (along edge $\bar{bc}$); no information is needed from cells "downstream" of this cell, e.g. cells 5 and 7, in this example. These terms (i.e., "upstream" and "downstream") can be made precise, by looking at the angle between the direction i and the normals to these edges, but we refer the reader to (22; 23) for technical details about this. From a computational standpoint, this means that each direction induces a set of precedence constraints captured by a dependency graph; for instance, in Figure 1, cell 4 depends on cell 2 for information, and so there is a precedence $(2, i) \rightarrow (4, i)$ in the corresponding DAG. In each direction, the dependency graph is different, but is induced on the same set of mesh elements - this is captured by the notation (v, i), which stands for the copy of cell v in direction i. As in Pautz (22; 23), there could be cyclic dependencies in

⋆ A preliminary version of this work appeared in Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2005.
* Corresponding author
[1] Research supported in part by NSF Award CCR-0208005.
[2] Referred to in (22) as the "streaming-plus-collision"

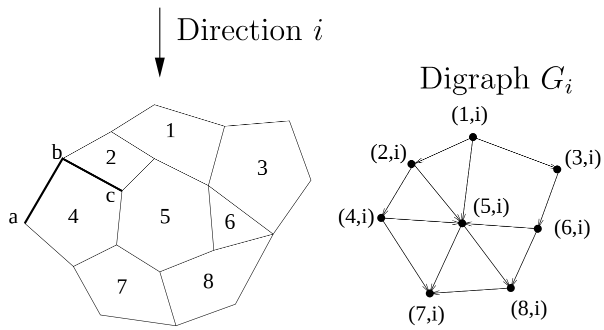


Fig. 1. Example of a mesh and the digraph induced by direction i. The heavy edges of cell 4 show the edges from which information is needed before processing it. The edge $\bar{ab}$ is a boundary edge, whereas the edge $\bar{bc}$ shares an edge with cell 2, which is upstream to cell 4, w.r.t. this direction. The figure on the right shows the DAG corresponding to the mesh for direction i, and the nodes in the DAG are labeled as tuples (v, i), for each mesh cell v. There is an edge from (v, i) to (w, i) in the DAG if v is upstream of w w.r.t. this direction.

the case of unstructured meshes, but there are known algorithms for detecting and removing cycles (see, e.g. (24)). Therefore, without loss of generality, we will assume that the dependency graph in each direction is a directed acyclic graph (DAG). The objective of the Sweep Scheduling problem is to assign the mesh cells to processors, and construct a schedule of the *smallest length* (or *makespan*), that respects all the precedence constraints. Messages that need to be sent from one processor to another incur communication delays, which could increase the makespan - in the models of Rayward-Smith (26) and Hwang et al. (11), for each edge $((v, i), (w, i))$ in the precedence graph, task (w, i) can start only certain time T after task (v, i) is completed if (v, i) and (w, i) are assigned to different processors - here, T corresponds to the time it takes for the information about (v, i) to be sent to the processor that is going to process (w, i). We also study two other communication cost models empirically; these are all defined formally in Section 3. In this paper, we will assume that the schedule is completely determined upfront; in reality, this is not strictly necessary, and some steps of the schedule could be changed locally.

Generalized Sweep Scheduling (GSS) problem. While the traditional applications that motivate the sweep scheduling problem are fundamentally geometric, we now discuss a more general non-geometric setting of this problem which is more suitable for a wider class of applications, that we call the *Generalized Sweep Scheduling (GSS)* problem. GSS generalizes the sweep scheduling problem and is applicable in more general settings that place co-location constraints on certain jobs to be scheduled on a parallel machine. An instance of the GSS problem consists of a directed acyclic graph (DAG) $G(V, E)$, and a partition of V into sets $V_1, V_2, \ldots$. The set V denotes the set of all tasks to be done, and $G(V, E)$ denotes the precedence constraints on V - The goal of the GSS problem is to construct a schedule that minimizes the maximum completion time on any processor (makespan) subject to the following constraints: (i) satisfy all precedence constraints as given by G, and (ii) satisfy the following *collocation* constraints - for each V_j, all the tasks in it must be performed on the same processor.

The special case of the GSS problem, where $|V_j| = 1$ for each j, corresponds to the precedence constrained scheduling (PCS) problem (e.g. (16)), that has been studied extensively. It can be verified that GSS subsumes SS by noting the following: (i) each computation on a mesh cell v, in each direction i corresponds to a task and $V_j = \{(v_j, 1), \ldots, (v_j, k)\}$ for some mesh cell v_j, i.e., V_j contains the tasks corresponding to cell v_j in the k different directions, and (ii) each cell v_j is required to be processed on a fixed processor for all the directions. However, both the SS and GSS problems seem much harder than the PCS problem, and do not fall into any of the categories in the classification of machine scheduling problems, defined by Graham et al. (8). The SS problem is a special case of the GSS problem, where the precedence constraints and the collocation constraints are decomposable, i.e., the DAG G can be expressed as a collection of distinct DAGs $G_1, \ldots, G_k$ (corresponding to the directions), with each V_j containing exactly one node from any G_i; an additional property of the SS problem in radiation transport is that the precedence constraints $G(V, E)$ are geometric, since they arise out of cells in a three dimensional mesh, and all the known heuristics, such as those in (22; 23), are carefully tied to this geometric structure.

The GSS problem is of much wider interest than radiation transport - it frequently arises in problems related to efficient parallel simulation of interdependent socio-technical systems, and large grid applications. For example, simulations such as *EpiSims* (7), *Transims* (25; 3) and *Simfrastructure* (4; 5). Each of the simulation involve the same set of individuals (agents) engaged is different activities. For example, in *EpiSims*, agents interact and transmit infectious diseases, in *Transims* they drive vehicles and in a telecommunication simulation they might be involved in use of wireless devices. These simulations interact and exchange data generated during the course of running the simulations. For example, the location data of an individual is used in the epidemiological and telecommunication simulations. For purposes of efficiency (both in terms of total space and the communication cost) when such simulations are run concurrently, it is highly desirable to keep a single copy of the agent even though the agent participates in different simulations. Moreover, the precedence constraints in such computations form arbitrary DAGs and do not necessarily have any geometric structure to them. It is easy to see how the scheduling problem for these concurrent simulations can be cast as GSS problem. The CHAMPS system of Keller et al. (13) and the Streamline system of Agarwalla et al. (1) are instances of large grid based applications, where constraints similar to GSS arise. Thus, the GSS problem is potentially of broader interest. In this paper, we take the first step towards understanding this problem, by developing provable algorithms for the sweep scheduling problem, that do not use any geometric information about the precedence constraints.

The rest of the paper is organized as follows. In Section 2, we present the main contributions of this work. In Section 3, we present the formal problem statement as well as the basic terms and definitions used in the rest of the paper. In section 4, we present our three main randomized algorithms for the sweep scheduling problem along with the analysis of their running times and performance guarantees. In Section 5, we present the experimental evaluation of these algorithms as well as our heuristics for minimizing the communication cost. We conclude with a discussion of future research directions in Section 6.

2 Our Contributions

Analytical Results: Provable Approximation Algorithms. We develop the first known algorithm for the sweep scheduling problem with provable performance guarantees. Our algorithm does not require any geometric assumptions about the precedence constraints, and therefore, works in more general settings than radiation transport. For our theoretical bounds, we focus only on the problem which ignores the communication costs between processors - even this simplified version generalizes the well known precedence constrained scheduling problem (8), and is $\mathcal{NP}$-complete. In contrast, none of the known heuristics for sweep scheduling (22; 23) have provable performance guarantees, and it is not clear how to use them in the presence of more general, non-geometric precedence constraints.

We design the *Random Delay* algorithm, and analyze it rigorously to show that it gives an $O(\log^2 n)$ approximation [3] (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 (22) 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 one that has the same analytical performance guarantee but performs significantly better in practice. While the running time is usually not crucial, it is interesting to note that our algorithms run in time almost

[3] An approximation factor of α implies that the length of the schedule obtained by this algorithm is at most α 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.

linear in the length of the schedule. Our algorithms assume no relation between the DAGs in different directions and are thus 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 (12)).

The algorithms can also be extended for certain communication models proposed by Rayward-Smith (26) and Hwang et al. (11). The extended algorithm yields a $O(C_{\max} \log m \log\log\log m)$ bound on the makespan in the communication latency model of (26; 11). Here $C_{\max}$ denotes the maximum interprocessor communication latency and is defined formally in Section 4.4.

Empirical Performance. We also study the empirical performance of our algorithms on unstructured meshes (more details about these meshes are give in Section 5). In addition to the makespan, we also look at two models of communication cost, which are described in more detail in Section 5. We also compare the performance of our algorithm with the DFDS heuristic (22), and other natural heuristics. Our main results are the following.

(1) Our algorithm has a much better performance in practice (in all the meshes we tried, we obtained schedules of length at most three times the optimal), than the logarithmic worst case bound we are able to prove in general. 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 is very low, but the communication overhead (number of interprocessor edges) is 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 makespan too much, but the communication overhead came down significantly.
(3) We compare the performance of our algorithm with the DFDS heuristic of Pautz (22), and other natural heuristics, using the same block decomposition. Our algorithm compares well in terms of the makespan, in spite of its simplicity. Since we use the same block decomposition, the overall communication cost is the same for all the algorithms. Significantly, we also observe that combining our random delays technique with some of the existing heuristics improves the performance in some cases.

In this work, we only focus on simulations of our algorithms on real meshes, instead of actual implementation on a parallel cluster - this means that we generate the actual schedules, but do not perform all the steps of the radiation transport calculations at each step. While such a simulation does not give the *real* cost of the sweep, it gives us a qualitative comparison between various algorithms, and gives us good lower bounds for the schedules. As mentioned in Theorem 2, the schedule can be computed very fast, and such a simulation allows us to make many runs.

Related Work. Because of its general applicability, sweep scheduling has been an active area of research. When the mesh is very regular, the KBA algorithm (14) is known to be essentially optimal. However, when the mesh is irregular, or unstructured, it is not easy to solve. There has been a lot of research in developing efficient algorithms for sweep scheduling, by exploiting the geometric structure, for instance by Pautz (22) and Plimpton et al. (23; 24). The results of Amato et al. (2) and Mathis et al. (19) develops a theoretical model for finding good decomposition techniques that can be used along with other sweep scheduling algorithms. However, *none* of these 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 as in the parallel implementation of *EpiSims* (7) and *Transims* (25), as discussed earlier), such symmetry does not exist. In such scenarios, it is unclear how the heuristics of (22; 23; 19) 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. (12) 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 denoted as $P|prec|C_{max}$ in the notation of Graham et al. (8) who also give a simple $2 - \frac{1}{m}$ approximation algorithm; there has been an enormous amount of subsequent research on other variants of this problem (see (12)). However, there has been very little work that includes communication cost - the only model studied in this context was introduced by Rayward-smith (26), and is denoted $P|prec, p, c|C_{max}$. This model involves communication latencies of the following form: for any edge (v, w), if v is assigned to processor i and w is assigned to processor $i' \neq i$, then w can be processed only $c_{ii'}$ time units after v has been completed, with $c_{ii'}$ denoting the time needed to send a message from processor i to processor i'; the goal is to minimize the makespan, with this communication latency incorporated. All the known results are about the special case of uniform communication delays, i.e., $c_{ii'} = c, \forall i, i'$, and usually, $c = 1$. Hoogeven, Lenstra and Veltman (10) showed that it is $\mathcal{NP}$-complete to get an approximation better than $\frac{5}{4}$. Rayward-Smith (26) and Hwang et al. (11) give constant factor approximation algorithms in this model, which was improved by Munier and Hanen (21) to $\frac{7}{3} - \frac{4}{3m}$. A generalization of Rayward-Smith's model is proposed by Hwang et al. (11); the model is described in Section 4.4; our results hold for this model.

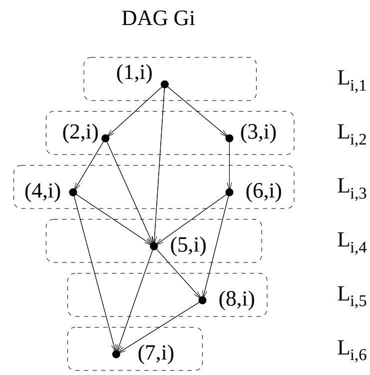


Fig. 2. Levels of the digraph shown in Figure 1.

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 if and only if u and v are adjacent in G *and* the sweep in direction i requires u to be done before v. Figure 1 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 using the algorithm of (24)) 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 $G_i(V_i, E_i), i = 1, \ldots, k$ (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, interprocessor communication will increase the makespan 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 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.

3.0.0.1 Levels. Given k DAGs $G_i(V_i, E_i), i = 1, \ldots, k$, 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 $L_{i,1} \cup \cdots \cup L_{i,j-1}$ have been deleted. We define D as the maximum number of layers in any direction. In Figure 2 we show how levels are formed for the example in Figure 1. 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).

3.0.0.2 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.

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 $R(t) \subset V \times \{1, \ldots, k\}$ 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. 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. Note, however, that both the above randomized steps are likely to lead to huge communication costs, as confirmed in our experiments in Figure 5 in Section 5.1. This can be improved significantly by first doing a decomposition into blocks and then doing the random assignment on the blocks, as described in Section 5.1.

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

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

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.

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 (6; 9))*** *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 $c \geq 0$,

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

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:

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

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

The following corollary follows.

Corollary 1 *Let $X_1, \ldots, X_n \in [0,1]$ be independent random variables and let $X = \sum_{i=1}^{n} X_i$. Let $\mathbf{E}[X] \leq \mu$. Then, for any sufficiently large $c \geq 0$,*

$$\Pr[X > c \log n(\max\{\mu, 1\})] < \frac{1}{n^c} . \tag{3}$$

PROOF. It can be checked that for $\delta \geq 4$, $\left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\mu} \leq e^{-\delta\mu}$. Now let $\delta = c \log n$. We then get that $\Pr[X > c \log n\mu] < e^{-c \log n\mu}$. For $\mu \geq 1$ we further get $e^{-c \log n\mu} = 1/n^{c\mu} \leq 1/n^c$. $\square$

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.

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 β is a constant which can be made suitably large by choosing α appropriately.*

PROOF. Let $Y_{r,v,i}$ be the indicator random variable which is 1 if task (v,i) is in layer L_r and 0 otherwise. Since we choose X_i randomly from $\{0, \ldots, k-1\}$, we have $\Pr[Y_{r,v,i} = 1] \leq \frac{1}{k}$. Let $N_{r,v} = \sum_i^k Y_{r,v,i}$ be the random variable that denotes the number of copies of v in layer L_r. By linearity of expectation, we have $\mathbf{E}[N_{r,v}] = \sum_i^k \mathbf{E}[Y_{r,v,i}] = \sum_i^k \Pr[Y_{r,v,i} = 1] \leq \frac{k}{k} = 1$. Applying Lemma 1(a), we have $\Pr[N_{r,v} > c \log n] < \frac{1}{n^{O(1)}}$. Let $\mathcal{E}$ denote the event that there exists a vertex u and a layer l such that the number of copies of u in l is $> c \log n$. By the union bound, we have $\Pr[\mathcal{E}] \leq \sum_{v,r} \Pr[N_{r,v} > c \log n] < \sum_{v,r} \frac{1}{n^{O(1)}} \leq \frac{n^2}{n^{O(1)}} \leq \frac{1}{n^{\beta}}$, by choosing c suitably large.

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 β' is a constant which can be made suitably large by choosing α' appropriately.*

PROOF. Consider any level L_r and a processor P. Let $Y_{P,v}$ be the indicator variable which is one if vertex v is assigned to processor P and zero otherwise. Due to the random assignment, we have $\Pr[Y_{P,v} = 1] = \frac{1}{m}$. Let $N_{P,r} = \sum_{v \in V_r} Y_{P,v}$ be the random variable which denotes the number of vertices in V_r

that are assigned to P. By linearity of expectation, we have $\mathbf{E}[N_{P,r}] = \frac{|V_r|}{m}$. By Lemma 1(a), we have $\Pr[N_{P,r} > c \log n(\max\{\frac{|V_r|}{m}, 1\})] < \frac{1}{n^{O(1)}}$, for a sufficiently large c. By Lemma 2, with high probability, there are at most $\alpha \log n$ copies of any vertex v in L_r, where α is a constant.

Let $\mathcal{F}_{P,r}$ denote the event that the total number of tasks assigned to processor P from level L_r is greater than $c' \cdot \max\{\frac{|V_r|}{m}, 1\} \cdot \log^2 n$, where c' is a constant. The above two arguments imply that

$$\Pr[\mathcal{F}_{P,r} > c' \max\{\frac{|V_r|}{m}, 1\} \log^2 n] < \frac{1}{n^\gamma},$$

where γ is a constant which can be made sufficiently large by choosing the value of c' appropriately. Let $\mathcal{F}$ denote the event that there exists a processor P and level L_r such that event $\mathcal{F}_{P,r}$ holds. By the union bound, we have

$$\Pr[\mathcal{F}] = \sum_{P,r} \Pr[\mathcal{F}_{P,r}] \leq \sum_{P,r} \frac{1}{n^\gamma} \leq \frac{n^2}{n^\gamma} \leq \frac{1}{n^{\gamma-2}} \ ,$$

where γ can be made suitably large. Hence, the lemma follows.

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 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 $O(OPT \log^2 n)$.

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.*

PROOF. The approximation guarantee follows from Lemma 4. It is easy to see that the algorithm runs in time $O(k + kn^2 + n + mnk)$. Since $k = O(n)$ and $m = O(n)$, the algorithm runs in polynomial time of the input size n.

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 Γ 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 guarantee and the running time of Algorithm 2.

Algorithm 2 Random Delays with Priorities

For all $i \in [1, \ldots, k]$, choose $X_i \in \{0, \ldots, k-1\}$ uniformly at random.
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).
For each vertex $v \in V$, choose a processor uniformly at random from $\{1, \ldots, m\}$.
$t = 1$.
while not all tasks have been processed **do**
  for all processors $P = 1, \ldots, m$ **do**
    (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).
    (ii) Schedule (v, i) on P at time t.
  end for
  $t \leftarrow t + 1$.
end while

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

PROOF.

Running time: To prove the claimed running time, we use a priority queue data structure which supports the operations: (i) Build Priority Queue, in $O(N \log N)$ time, where N is the total number of items, (ii) Find Min, in $O(1)$ time, (iii) Delete Min, in $O(1)$ time and (iv) Update Priority, in $O(\log N)$ time. Each item has a key, and the items are ordered based on this key. The Find Min operation returns the item with the smallest key value. In our case $N = mk$, since we have at most m edges in each DAG and a total of k distinct DAGs. For meshes arising in practice, $m = O(n)$.

To improve the efficiency, we define the key value of task (v, i) as $key(v, i) = \Gamma(v, i) + W \cdot indegree(v, i)$. Here, W is a large number (e.g. $10nk$), and $indegree(v, i)$ is the (current) number of immediate predecessors of task (v, i). As the schedule proceeds, $indegree(v, i)$ will reduce, and the key values will reduce. After the random delays are determined, $\Gamma(v, i)$, and therefore $key(v, i)$ is fixed for each task (v, i); we use these key values to construct a separate priority queue for each processor.

At each step, each processor looks at the task with smallest key value in its heap. If its indegree is 0, it performs it in the current step, and for each child w of this task, it reduces the indegree of w by 1; the key values of such tasks also need to be updated, using the Update Priority operation. Thus, whenever a task w is completed, it requires $O(outdegree(w) \log nk)$ time, which gives the bound in the lemma. Also, because of the definition of the key value, it follows that nodes of indegree i will have priorities much lower than nodes of priority $i + 1$, for any i. This ensures that only ready tasks are picked at any time.

Performance analysis: Recall the sets L_r defined in Algorithm 1. Let $\mathcal{S}$ and $\mathcal{S}'$ be the schedules produces by Algorithms 1 and 2 respectively. Let $t(v, i)$ and $t'(v, i)$ be the times at which task (v, i) got completed in the schedules $\mathcal{S}$ and $\mathcal{S}'$, respectively. We will show that for each r, $\max_{(v,i)\in L_r}\{t'(v, i)\} \leq \max_{(v,i)\in L_r}\{t(v, i)\}$. The claim then follows. Observe that for each $(v, i) \in L_r$, the priority $\Gamma(v, i) = r$. We now prove the above statement by induction on r.

Base Case: For $r = 1$, all nodes in L_1 have the lowest priority of 1, which is lower than the priority of any other node. Therefore, each processor P will schedule tasks in L_1, as long as there are any tasks in L_1 assigned to it. So, we have $\max_{(v,i)\in L_1}\{t'(v, i)\} \leq \max_{(v,i)\in L_1}\{t(v, i)\}$.
Induction hypothesis: Assume that the claim holds for all $r \leq l$.
Induction step: We now show the claim for $r = l + 1$. In schedule $\mathcal{S}$, the tasks in L_{l+1} are started only after those in L_l are completed; let t denote the time when the last task in L_l got completed in $\mathcal{S}$. By the induction hypothesis, applied to L_l, all tasks in L_l are already completed in $\mathcal{S}'$, by this time. Also, the priority of any task in L_{l+1} is lower than that of any task in L_j for $j > l+1$.

Therefore, in $\mathcal{S}'$, each processor P will first complete the tasks assigned to it in L_{l+1}, and only then would it pick tasks with higher $\Gamma()$ value. Therefore, $\max_{(v,i)\in L_r}\{t'(v,i)\} \leq \max_{(v,i)\in L_r}\{t(v,i)\}$.

4.3 An Improved $O(\log m \log\log\log m)$-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 3 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, as a result of which a lot of tasks from some direction could be assigned to the same layer of the combined DAG. The new algorithm handles this problem by the pre-processing step, which is the essential difference between this and the previous random delay algorithms. The pre-processing step transforms the original instance, so that there are at most m tasks in each layer in each direction, and also guarantees that, in expectation, at most m tasks are assigned to each layer of the combined DAG.

Analysis. For the tighter analysis, we need to look at the time taken to process all the tasks in any layer L''_t. 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 $\mu_t = E[|L''_t|]$.*

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

PROOF. ¿From the analysis in (8), it follows that $T \leq 2OPT$; therefore, $OPT = \Omega(nk/m+T+k)$. Next, $\sum_t |L''_t| = nk$ and thus, $\sum_t \mu_t = nk$. Summing the bound of Theorem 3 over all t, we get that the expected final makespan is $O(nk/m + (T+k)\rho)$, which gives an $O(\rho)$–approximation.

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 (8); let T be the makespan of this schedule.
   - Let $L'_{ij} = \{(v,i) \in V_i | (v,i)$ done at step j of above schedule$\}$.

2: For all $i \in [1,\ldots,k]$, choose $X_i \in \{0,\ldots,k-1\}$ uniformly at random.

3: Form a combined DAG G'' as follows: $\forall r \in \{1,\ldots,T+k-1\}$, 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 $\{1,\ldots,m\}$.

5: Construct a schedule by processing layers $L''_1, L''_2, \ldots$ 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 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.

$$H(\mu,p) = \begin{cases} C \cdot \frac{\ln(p^{-1})}{\ln(\ln(p^{-1})/\mu)} & \text{if } \mu \le \ln(p^{-1})/e; \\ Ce\mu & \text{otherwise.} \end{cases} \tag{4}$$

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

Corollary 3 *(a) If we fix p, then $H(\mu,p)$ is a concave function of μ. (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 $H(t/m, 1/m^2) + t/m$.*

PROOF. (a). Fix p. The second derivative of $H(\mu,p)$ w.r.t. μ, can be seen to be non-positive when $\mu \le \ln(p^{-1})/e$; thus, H is concave in this region. Since H is linear for larger μ, it is trivially concave in this region also. We see that H is concave in general, by noting as above that H is continuous and differentiable at $\mu = \ln(p^{-1})/e$. (b) Consider any machine i; the load X_i on it is a sum of i.i.d. indicator random variables, and $\mathbf{E}[X_i] = t/m$. Now, it is easy to verify that if C is large enough, then $F(\mu,p) \le H(\mu,p)$. Thus, letting $\mathcal{E}_i$ be the event "$X_i \geq H(t/m, 1/m^2)$", Lemma 1(b) shows that $\Pr[\mathcal{E}_i] \leq 1/m^2$; so, $\Pr[\exists i : \mathcal{E}_i] \leq 1/m$. If the event "$\exists i : \mathcal{E}_i$" does not hold, then the maximum load on any machine is at most $H(t/m, 1/m^2)$ with probability 1; else if "$\exists i : \mathcal{E}_i$" is true, then the maximum load on any machine is at most t with probability 1. Therefore, the expected maximum load is at most $H(t/m, 1/m^2) + (1/m) \cdot t$.

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. It can be verified that the second derivative of ϕ_a satisfies $\phi_a''(x) = x^{a-2}e^{-x}((a-x)^2 - a)$, which in turn is at least $x^{a-2}e^{-x}((a-1)^2 - a)$, for the given range of x and a. Since $(a-1)^2 - a \geq 0$ for $a \geq 3$, the lemma follows.

4.3.0.3 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 μ_t.

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

PROOF. Part (a) follows by the definitions of Z_j and μ_t, and from the linearity of expectation. For part (b), note first that a node $(v,i) \in L_{ij}'$ can get assigned to layer L_t'' only if $t - k + 1 \leq j \leq t$. By the preprocessing step, $|\cup_i L_{ij}'| \leq m$, for each j, and therefore, the number of such nodes (v,i) assigned to L_t'' is at most mk. For each such node (v,i), the probability of getting assigned to layer L_t'' is $1/k$, since X_i is chosen uniformly random in the range $0, \ldots, k-1$. Thus, $\mu_t \leq m$.

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

PROOF. Fix $j \geq 2$, and let $a = 2^j$. Let N_v be the random variable denoting the number of copies of job $v \in V$ that get assigned to layer L_t''; letting $b_v = \mathbf{E}[N_v]$, we also have $b_v \leq 1$. Furthermore, $\mu_t = \sum_v b_v$ and $\mathbf{E}[Z_j] \leq \sum_v \Pr[N_v \geq a]$.

Now, Lemma 1(a) yields $\Pr[N_v \geq 2^j] \leq (e/a)^a \cdot b_v^a e^{-b_v}$, and so

$$\mathbf{E}[Z_j] \leq \sum_v (e/a)^a \cdot b_v^a e^{-b_v}. \tag{5}$$

Now, $(e/a)^a \cdot b_v^a e^{-b_v}$ is a convex function function of b_v (for fixed j), by Lemma 5. Thus we get, for any fixed value of μ_t:

- if $\mu_t < 1$, then the r.h.s. of (5) is maximized when $b_v = \mu_t$ for some v, and $b_w = 0$ for all other w; so, in this case, $\mathbf{E}[Z_j] \le (e/a)^a \cdot {\mu_t}^a e^{-\mu_t}$.
- if $\mu_t \ge 1$, then the r.h.s. of (5) is at most what it would be, if we had $\lceil \mu_t \rceil$ indices v with $b_v = 1$, with $b_w = 0$ for all other w; so, in this case, $\mathbf{E}[Z_j] \le (2\mu_t) \cdot (e/a)^a \cdot e^{-1}$.

This yields the lemma.

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_1 to completion, then run all nodes in Z_2 to completion, and so on. 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 3, 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:

$$\begin{aligned}
\mathbf{E}[Y_t \mid (Z_0, Z_1, \ldots)] &\le [\sum_{j=0}^{\ln\ln m} 2^{j+1} \cdot (H(|Z_j|/m, 1/m^2) \\
&\quad + |Z_j|/m)] + [\sum_{j > \ln\ln m} 2^{j+1} \cdot |Z_j|] \\
&\le [\sum_{j=0}^{\ln\ln m} 2^{j+1} \cdot (\mathbf{E}[H(|Z_j|/m, 1/m^2)] + \mathbf{E}[|Z_j|]/m)] \\
&\quad + [\sum_{j > \ln\ln m} 2^{j+1} \cdot \mathbf{E}[|Z_j|]] \\
&\le [\sum_{j=0}^{\ln\ln m} 2^{j+1} \cdot (H(\mathbf{E}[|Z_j|]/m, 1/m^2) + \mathbf{E}[|Z_j|]/m)] \\
&\quad + [\sum_{j > \ln\ln m} 2^{j+1} \cdot \mathbf{E}[|Z_j|]]
\end{aligned}$$

This follows since H is concave by Corollary 3(a). (We are using Jensen's inequality: for any concave function f of a random variable T, $\mathbf{E}[f(T)] \le f(\mathbf{E}[T])$.) 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 (4) that if p is fixed, then $H(\mu, p)$ is a non-decreasing function of μ. So, Lemmas 6 and 7 show that there is a value $\alpha \le O((\ln m)/\ln\ln m)$ such that $H(\mathbf{E}[|Z_j|]/m, 1/m^2) \le \alpha$ for $j = 0, 1$.

Hence,

$$
\begin{aligned}
&\sum_{j=0}^{\ln\ln m} 2^{j+1} \cdot H(\mathbf{E}[|Z_j|]/m, 1/m^2) \\
&\leq \; O(\alpha) + \sum_{j=2}^{\ln\ln m} 2^{j+1} \cdot H(\mathbf{E}[|Z_j|]/m, 1/m^2) \\
&\leq \; O(\alpha) + \sum_{j=2}^{\ln\ln m} 2^{j+1} \cdot H((e/2^j)^{2^j}, 1/m^2) \\
&\quad \text{(by Lemmas 6(b) and 7)} \\
&\leq \; O(\alpha) + O\left(\sum_{j=2}^{\ln\ln m} 2^{j+1} \cdot \frac{\ln m}{\ln\ln m + j2^j}\right). \qquad (6)
\end{aligned}
$$

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

$$
\sum_{j > \ln\ln m} 2^{j+1} \cdot \mathbf{E}[|Z_j|] \leq \mu_t \cdot \sum_{j > \ln\ln m} 2^{j+1} \cdot (e/2^j)^{2^j} = O(\mu_t/m), \qquad (7)
$$

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.

4.4 Communication cost

We briefly discuss the problem of bounding the makespan for the GSS problem subject to inter-processor communication latencies. We call this problem GSS-ICD - the GSS problem with inter-processor communication delays. As mentioned earlier, we use an extension of the model proposed by Hwang et al. (11). Their model is an extension of the well known model of Rayward-Smith (26). In the extended model, we are given an instance of GSS as before. The jobs are required to be processed on a parallel machine with m processors as before. The crucial difference now is that we now have two additional costs: each edge (v, w) of the DAG G has an associated weight $\eta(v, w)$ denoting the size of message sent by job v to w upon completion of V_i. Secondly, the m processors are joined together in form of a network and there is parameter

$\tau(p_i, p_j)$ denoting the delay in sending a message from p_i to p_j. Thus if v is assigned to processor p_l and w is assigned to processor p_k and w is an immediate successor of v, then the w has to wait an additional $\eta(v, w) \times \tau(p_l, p_k)$ time after v is completed before it can be considered for processing.

We now briefly describe how we can extend the $O(\log m \log\log\log m)$ bound of Section 4.3 to the communication latency model of (26; 21; 11). Let $C_{\max} = \max_{v,w,l,k}\{\eta(v, w) \times \tau(p_l, p_k)\}$ denote the maximum communication latency. Note that the schedule $\mathcal{S}$ computed by Algorithm 3 processes the layers $L''_1, L''_2, \ldots$ sequentially. We dilate the schedule $\mathcal{S}$ by an $O(C_{\max})$ factor in the following manner: for each t, after layer L''_t has been completely processed, we wait for $C_{\max}$ steps for all communication to be completed, and then start layer L''_{t+1}. Denote the modified algorithm as Algorithm 4. By Corollary 2, we get the following result.

Corollary 4 *Let OPT denote the length of the optimal schedule for the GSS-ICD problem. Then Algorithm 4 yields a schedule of expected length $O(C_{\max} \cdot \log m \log\log\log m \cdot OPT)$.*

The communication model crucially assumes no communication contention: a processor can simultaneously communicate with all processors that it needs to in a given round. In the next section, we consider an extension of this model that accounts for certain aspects of contention present in real systems. Nevertheless, despite its simplicity, the model does capture the inter-processor communication delay and has thus been used frequently in the scheduling community, see a recent comprehensive survey of Kwok and Ahmad (15) for additional discussion.

5 Experiments with Random Delay Algorithms

In this section we report on our empirical analysis. We implemented the algorithms described in Section 4.1 and 4.2 and tested them on a number of instances. A number of objective functions were used to evaluate the performance of our algorithms. Below, we describe these components of the empirical analysis in detail

5.0.0.4 Data. We used the following four meshes in our experiments: (i) `tetonly`, with 31481 cells, (ii) `well_logging` with 43012 cells, (iii) `long` with 61737 cells, and (iv) `prismtet`, with 118211 cells. For each of these, we tried a large range of directions (up to 120), blocks sizes (up to 256) and number of processors (up to 500), but we made sure that the number of blocks per processor did not become too small (we chose this lower limit as 5 in our

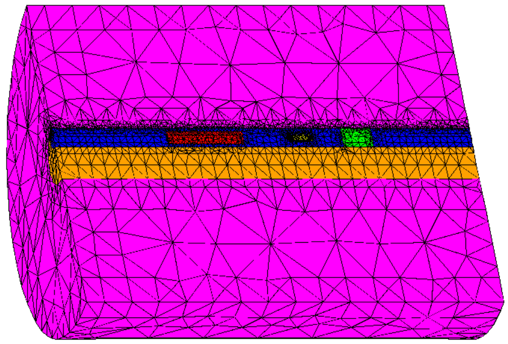


Fig. 3. A picture of the `well_logging` mesh from (27), who in turn take this configuration from (29). This picture is shown here with the permission of Jim Warsa. This mesh has 43012 cells, and has length 140cm and radius 60 cm, and comes out of an oil well drilling application.

experiments). The two meshes, `tetonly` and `prismtet`, are unstructured with similar dimensions. The difference is that the `tetonly` mesh has aspect ratios close to unity while the prismtet has a region of structured prisms, subdivided into tetrahedrons, extruded with increasing logarithmic spacing into one of the regions for a boundary layer resolution. `long` is a structured hexagonal mesh whose vertices are randomly perturbed and subdivided into tetrahedrons. The `well_logging` mesh is a fully unstructured, well-shaped mesh composed of tetrahedrons, with some regions highly-resolved to capture the features of the physical model; Figure 3 shows a picture of it. As mentioned in (27), this mesh is based on the configuration from (29). These meshes were chosen because they had varying number of cells, and varying densities; they were given to us by Jim Warsa, a researcher in the Radiation Transport group (CCS-4) at the Los Alamos National Laboratory and have been used in other papers by Warsa et al. (27; 28). Throughout our experiments, the qualitative results have been very similar across the different meshes. Therefore, for purposes of exposition, we only show plots for a few meshes.

5.0.0.5 Objective functions. As mentioned earlier, we will *simulate* the sweeps, instead of actually running them on a distributed machine, and identify machine independent optimizations. We compute the whole schedule, but do not actually *run* it on a parallel cluster. As mentioned in Theorem 2, the schedule can be computed very fast, in $O((mk + nk) \log nk)$ time, and such a simulation allows us to make many runs. 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 (i) after each step of computation, there is a round of communication, (ii) at any time, a processor can only send or receive a message (i.e., it cannot do both), and, (iii) at a time a processor can only communicate with one other processor. In reality, all of these can be relaxed, but this model serves as a good first abstraction, and it is possible to perform all the communication in time proportional to the maximum number of messages any processor has to send (or, the maximum degree). Thus, for the cost C_2, we sum up the maximum degree of any processor over all the communication rounds. Furthermore, it is not necessary to strictly alternate the computation and communication rounds - it might be possible to have multiple rounds of computation, before performing a communication round.

5.0.0.6 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 5(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 5(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

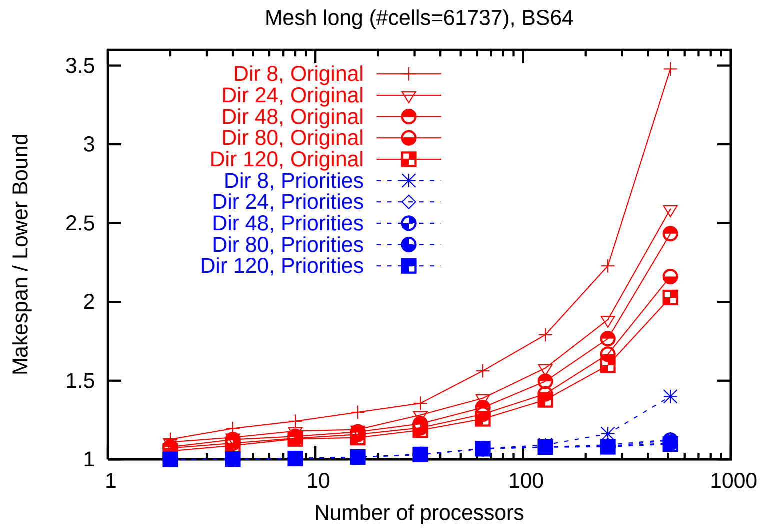


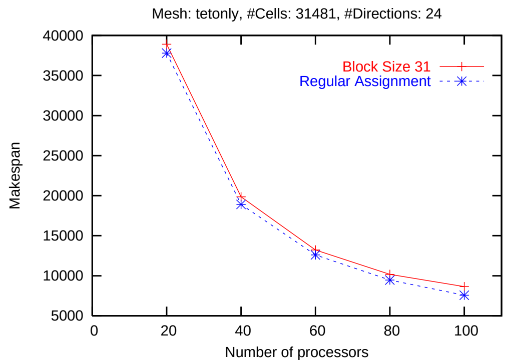
Fig. 4. Original implementation of random delay algorithm versus priority based implementation for different numbers of directions and increasing numbers of processors for mesh `long`.

to a factor of 4 (Figure 5(c)). Also, in all our runs, the total makespan was at most $3nk/m$; we ran the simulations with varying number of processors, while ensuring that the minimum number of blocks per processor was not too small (at least 10, which was chosen arbitrarily).

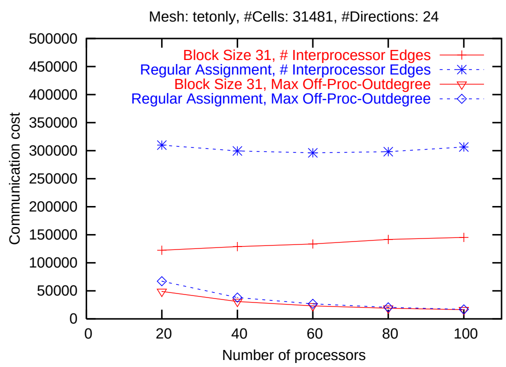
5.1.0.7 Partitioning into Blocks. In an attempt to bring down the number of interprocessor edges (C_1), we partition the cells of the mesh into blocks, using METIS, which is a popular freely available graph partitioning software (20). Instead of choosing a processor for each individual cell (as in Algorithm 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 5(a)), while the number of interprocessor edges decreases significantly (Figure 5(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

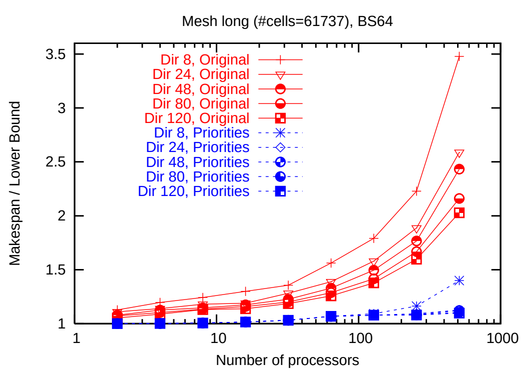
In this section, we compare the empirical performance of the algorithm "Random Delays with Priorities" with several different heuristics. We show that we get even better performance by combining ideas from these different heuris-



(a)

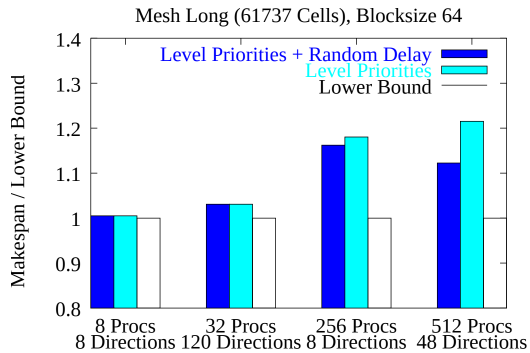


(b)

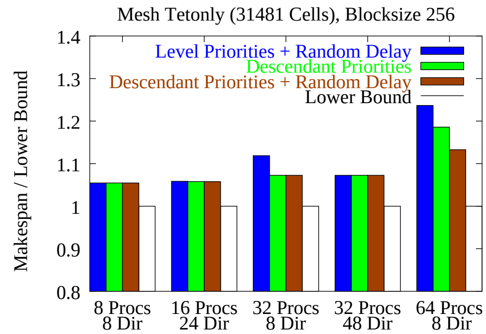


(c)

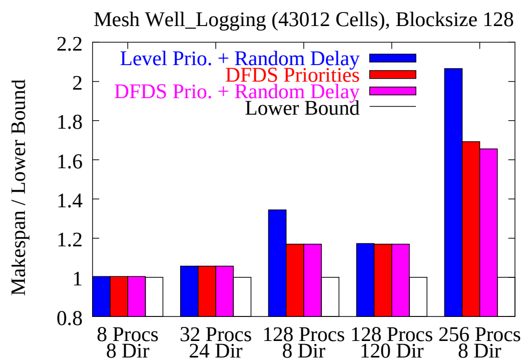
Fig. 5. (a)&(b) Random delay scheduling on the mesh `tetonly` with 24 directions. Plotted is (a) the makespan, and (b) the number of interprocessor edges C_1, and the cost C_2 (labeled as *Max Off-Proc-Outdegree* in the plots), for a regular assignment of cells to processors as well as for a block assignment 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)

Fig. 6. 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.

tics. 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.

5.2.0.8 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 (23). 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 (22). In 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.

5.2.0.9 Performance of Level Priorities. In the bar chart shown in Figure 6(a) we compared our random delays algorithm 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.

5.2.0.10 Performance of Descendant Priorities. In the bar chart shown in Figure 6(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.

5.2.0.11 Performance of DFDS Priorities. In the bar chart shown in Figure 6(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.

6 Concluding Remarks

In this paper, we have described the first algorithms with provable logarithmic performance guarantees, for the generalized sweep scheduling problem. Our algorithm does not require the precedence constraints to be geometric, and therefore are applicable in more general settings that radiation transport. Our empirical analysis suggests that these algorithms compare well with the sweep scheduling algorithms studied in the literature. We conclude by suggesting a number of directions for future research.

(1) Obtaining theoretical bounds for the communication cost along with the makespan remains a challenging problem. It would also be interesting to take machine architecture into account in defining the communication cost.
(2) More research is needed to refine the algorithms discussed here, and possibly combine with other algorithms, to improve the empirical performance. Additionally, performance analysis of these algorithms on real systems remains to be done.
(3) We have not taken into account the cost of accessing data from levels of memories that exist in today's computing systems. It is possible to combine the communication and memory access costs, including features such as caching, in a single model. Another open question relates to real time scheduling; it is indeed possible that the performance of some of our algorithms can be improved by real time scheduling methods.

Acknowledgments. The work was partially supported by Department of Energy under Contract W-7405-ENG-36. 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 (both at Los Alamos) for valuable discussions throughout the course of this research. We also thank Jim Warsa for providing us the meshes, helping us with the code, and allowing us to use the Figure 3 from (27). Finally, we thank the

anonymous referees for very valuable comments that helped us improve the overall presentation.

References

[1] B. Agarwalla, N. Ahmed, D. Hilley and U. Ramachandran. Streamline: A Static Scheduling Heuristic for Streaming Applications on the Grid, To appear in *Proceedings of the Thirteenth Annual Multimedia Computing and Networking (MMCN'06)*, January 2006.

[2] N. Amato and P. An. Task Scheduling and Parallel Mesh-Sweeps in Transport Computations. Technical Report, TR00-009, Department of Computer Science, Texas A&M University, Jan 2000.

[3] C. Barrett, R. Beckman, K. Berkbigler, K. Bisset, B. Bush, K. Campbell, S. Eubank, K. Henson, J. Hurford, D. Kubicek, M. Marathe, P. Romero, J. Smith, L. Smith, P. Speckman, P. Stretz, G. Thayer, E. Eeckhout, and M.D. Williams. TRANSIMS: Transportation Analysis Simulation System. Technical Report LA-UR-00-1725, Los Alamos National Laboratory Unclassified Report, 2001. An earlier version appears as a 7 part technical report series LA-UR-99-1658 and LA-UR-99-2574 to LA-UR-99-2580.

[4] C. Barrett, S. Eubank, V. Anil Kumar, M. Marathe. Understanding Large Scale Social and Infrastructure Networks: A Simulation Based Approach, *SIAM news*, March 2004, Appears as part of Math Awareness Month on The Mathematics of Networks.

[5] C. Barrett, S. Eubank and M. Marathe, Modeling and Simulation of Large Biological, Information and Socio-Technical Systems: An Interaction Based Approach, to appear in *Interactive Computation: The New Paradigm,* D. Goldin, S. Smolka and P. Wegner Eds. Springer Verlag, (2005).

[6] 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.

[7] S. Eubank, H. Guclu, V. S. A. Kumar, M. V. Marathe, Srinivasan A., Z. Toroczkai, and N. Wang. Modeling disease outbreaks in realistic urban social networks. *Nature*, 429(6988):180–184, May 13, 2004.

[8] 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.

[9] W. Hoeffding. Probability Inequalities for Sums of Bounded Random Variables. *American Statistical Association Journal*, 58:13–30, 1963.

[10] 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.

[11] J-J. Hwang, Y-C. Chow, F. Angers and C-Y. Lee. Scheduling precedence graphs in systems with interprocessor communication times, *Siam Journal of Computing*, vol 18 (2), pp. 244-257, 1989.

[12] D. Karger, C. Stein, J. Wein. Scheduling Algorithms. Chapter in the *Handbook of Algorithms*, CRC Press, 1997.

[13] A. Keller, J.L. Hellerstein, J.L. Wolf, K.-L. Wu and V. Krishnan. The CHAMPS System: Change Management with Planning and Scheduling, *IBM Research Report*, RC22882 (W0308-089), 2005.

[14] 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.

[15] Y. Kwok and I. Ahmad. Static Scheduling Algorithms for Allocating Task Graphs to Multiprocessors, *ACM Computing Surveys,* 31(4), pp. 406-471

[16] 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.

[17] T. Leighton, B. Maggs and S. Rao. Packet Routing and Job Shop Scheduling in O(Congestion+Dilation) Steps. *Combinatorica 14(2): 167-186 (1994).*

[18] J.K. Lenstra and A.H.G. Rinnooy Kan. Complexity of scheduling under precedence constraints. *Operations Research*, 26, 22-35, 1978.

[19] M. Mathis, N. Amato and M. Adams. A General Performance Model for Parallel Sweeps on Orthogonal Grids for Particle Transport Calculations. *Proc. ACM Int. Conf. Supercomputing (ICS)*, pp. 255-263, Santa Fe, NM, May 2000.

[20] METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0 (1995). `http://www-users.cs.umn.edu/~karypis/metis/index.html`

[21] 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.

[22] S. Pautz. An Algorithm for Parallel S_n Sweeps on Unstructured Meshes. *Nuclear Science and Engineering*, 140, 111-136, 2002.

[23] S. Plimpton, B. Hendrickson, S. Burns and W. McLendon. Parallel Algorithms for Radiation Transport on Unstructured Grids, *Super Computing*, 2001.

[24] S. Plimpton, B. Hendrickson, S. Burns, W. McLendon III and L. Rauchwerger. Parallel Sn Sweeps on Unstructured Grids: Algorithms for Prioritization, Grid Partitioning, and Cycle Detection, *J. American Nuclear Society*, 150(3), 2005, pp. 267-283.

[25] Transims: Transportation analysis simulation system. `http://transims.tsasa.lanl.gov/`.

[26] V.T. Rayward-Smith. UET scheduling with interprocessor communication delays, *Discrete Applied Mathematics*, v. 18 (1), pp. 55 - 71 , 1987.

[27] J. S. Warsa, M. Benzi, T. A. Wareing and J. E. Morel. Preconditioning a Mixed Discontinuous Finite Element Method for Radiation Diffusion, *Numerical Linear Algebra with Applications*, v. 11, pp. 795-811, 2004.

[28] J. S. Warsa, T. A. Wareing, and J. E. Morel. Krylov Iterative Methods and the Degraded Effectiveness of Diffusion Synthetic Acceleration for Multidimensional SN Calculations in Problems with Material Discontinuities, *Nuclear Science and Engineering*, v. 147, pp. 218-248, 2004.

[29] Wareing TA, McGhee JM, Morel JE. ATTILA: A three dimensional, un-

structured tetrahedral mesh discrete ordinates transport code. *Transactions of the American Nuclear Society* vol. 75, pp. 146-147, 1996.