November 1984 Report No. STAN-CS-84-1025

Fast Scheduling Algorithms on Parallel Computers

by

David Helmbold and Ernst Mayr

Department of Computer Science

Stanford University
Stanford, CA 94305

Stanford University
Department of Computer Science

Fast Scheduling Algorithms on Parallel Computers

by

David Helmbold and Ernst Mayr

Abstract: With the introduction of parallel processing, scheduling problems have generated great interest. Although there are good sequential algorithms for many scheduling problems, there are few fast parallel scheduling algorithms. In this paper we present several good scheduling algorithms that run on EREW PRAMS. For the unit time execution case, we have algorithms that will schedule n jobs with intree or outtree precedence constraints in $O(\log n)$ time. The intree algorithm requires n^3 processors, and the outtree algorithm requires n^4 processors.

Another type of scheduling problem is list scheduling, where a list of n jobs with integer execution times is to be scheduled in list order. We show that the general list scheduling problem on two identical processors is polynomial-time complete, and therefore is not likely to have a fast parallel algorithm. However, when the length of the (binary representation of the) execution times is bounded by $O(\log^c n)$ there is an $\mathcal{NC}$ algorithm using n^4 processors.

This research was supported in part, by an IBM Faculty Development Award and NSF grant DCR--835 1757.

1. Introduction

It is now feasible to build massively parallel multiprocessor systems. One way to exploit the parallelism in these supercomputers is to partition problems into a number of small subtasks and then schedule the subtasks on the various processors. It is conceivable that, with an inefficient scheduling, all of the supercomputer's parallelism may be lost. It is desirable that these tasks be scheduled optimally, to complete the problem in the shortest possible time. Since some subtasks may have to wait for the results of other subtasks, full schedules are not, always possible. This 'must-wait-for' relationship between tasks is often represented by a directed graph, commonly called the dependency *graph* or *precedence constraint graph*. For many classes of dependency graphs, finding just the length of an optimal schedule is known to be an NP-complete problem [Ull75, May81]. Although good sequential algorithms exist for finding optimal or almost optimal schedules in other cases, few parallel scheduling algorithms are known.

In this paper we first, consider scheduling problems given by: a number of identical processors, a set of n unit execution time jobs, and a dependency graph on the jobs. Our results are algorithms which find greedy optimal schedules for subclasses of this problem in polylog time running on a PRAM with a polynomial number of processors. Our first algorithms are applicable when the dependency graph can be broken into many separate components. We then present an algorithm for scheduling outtree precedence graphs. Our main result is an algorithm which schedules intree precedence graphs. Intree precedence graphs are an important special case because they result from many natural problems, such as expression evaluation and production assembly.

List scheduling is another type of scheduling problem, consisting of a list of jobs with integer execution times. A list scheduling instance does not contain precedence constraints, but jobs must be started in list order. We prove that this problem with arbitrary integer execution times is $\mathcal{P}$-complete under log-space reduction. We also exhibit an $\mathcal{NC}$ algorithm for the list scheduling problem when the execution times are at most $O(\log^c n)$ bits long.

We use the Exclusive-Read, Exclusive- Write (EREW) PRAM model [FWy78] for our algorithms. This model consists of numbered, autonomous processors sharing a common memory and clock. Each processor is capable of the normal arithmetic operations (plus, minus, times and divide by 2), as well as indirect addressing. We assume that the processors are able to compute the memory location of an array cell in unit time (i.e., we use the unit cost model). Although the processors share a common memory, each individual memory cell can only be accessed by a single processor at any one step.

We will often need to fan out many copies of intermediate results between stages of our algorithms. Using n processors we can create n copies of a value in $\log n$ time. This additive factor of $\log n$ does not affect the asymptotic running time of our algorithms.

2. Basic Algorithms and Definitions

This section defines the scheduling problem and important concepts used throughout the paper, as well as presenting fundamental parallel algorithms.

2.1 Basic Algorithms

Since all our algorithms have steps requiring at least n^2 processors, we chose to represent our graph inputs in adjacency matrix form. This form is easy to manipulate in parallel with large numbers of processors. On the other hand, the processor requirements of some steps can be reduced to O(n) by choosing a more appropriate input representation. We have attempted to note which steps can be implemented more efficiently.

One well known parallel algorithm is the prefix sum algorithm. This algorithm takes the array VALUE$(0..n-1)$ of numbers and computes for each $i < n$, $\sum_{0 \leq k \leq i}$ VALUE(k). We assume that n is a power of two, if not the array can be padded with zeros. This algorithm is stated recursively, however it can be implemented iteratively using O(n) space.

```
algorithm PREFIX-SUM (n, VALUE);
Input: The number of inputs, $n$, and VALUE$(0..n-1)$ containing their values.
Output: The array VALUE$(0..n-1)$ containing the prefix sums.
begin
    if $n > 1$ then
        for each j from 0 to $n/2 - 1$ do in parallel
            TEMP(J) := VALUE(2 . $j$)+VALUE(2 . . + 1)
        od;
        PREFIX-SUM ($n$/2,TEMP);
        for each j from 0 to n/2 − 1 do in parallel
            VALUE(2 . . := TEMP( j) - VALUE(2 . . + 1);
            VALUE(2 · j + 1) := TEMP($j$)
        od
    fi
end .
```

Our scheduling algorithms make frequent use of the *path doubling algorithm*. This algorithm essentially computes the transitive closure of a directed acyclic graph with outdegree one, i.e. of an inforest. The algorithm starts with each vertex finding its successor's successor. **This** gives us all paths of length two. Then, using these paths as edges we again find each vertex's successor's successor and so on. It is not hard to find a concurrent read path doubling algorithm. The [DUW84] paper contains an exclusive read path doubling algorithm for linked lists similar to the following.

Our path doubling algorithm runs on a graph of in- and out-degree at most one (i.e. a group of disjoint linked lists). One of the input vertices is *marked.* The algorithm computes the sink reachable from each vertex, the distance to that sink, and marks all vertices reachable from the marked vertex.

```
algorithm PATH DOUBLING;
Input: A set of $n$ vertices, one of which (the start vertex) is marked.
    The array $E(v)$ is initialized to $v$'s successor, for all vertices v; if v is a sink, then
    $E(v)$ is initialized to NIL.
    The array elements of $DIST(v)$ are set to 1 if $E(v) \neq$ NIL and 0 otherwise.
Output: The array $E(v)$ will contain the sink reached from each vertex v.
    The array DIST(v) will contain the distance from each node v to a sink.
    Those vertices on the path from the start vertex to a sink will be marked.
begin
    LENGTH := 1;
    while LENGTH < $n$ do
        for each v do in parallel
            if v is marked then mark $E(v)$ fi;
            if E(v) $\neq$ NIL then
                $DIST(v) := DIST(v)+DIST(E(v))$;
                $E(v) := E(E(v))$;
            fi
        od;
        LENGTH := 2·LENGTH
    od
end PATH DOUBLING.
```

Synchronization of at most two processors reading the same array element is not a problem since the processors run in lockstep. This algorithm takes $O(\log n)$ time on O(n) processors to **compute** the marking, DIST and E arrays.

When the graph is an intrec, the path doubling algorithm results in read conflicts. Our *Path Finding Algorithm* is applicable in this case. The Path Finding Algorithm takes an inforest with one marked node (m) as input and and finds all nodes on the path from *m* to its root. IF the in Forest is initially given in pointer form, WC can convert it to adjacency matrix form using n^2 processors in constant time.

The first step in the algorithm is to expand the tree by replacing each node as in Figure 1. We call the resulting directed graph the *expanded tree.* Each vertex in the expanded tree contains information indicating its node in the tree and whether it is an S- or F–vertex.

WC can now compute the path through the expanded tree. The path enters each expanded node through the S (start) vertex, visits the node's dcsccnrlcnts, and leaves through the F (final) vertex. Using the path doubling algorithm on the expanded tree WC can find each

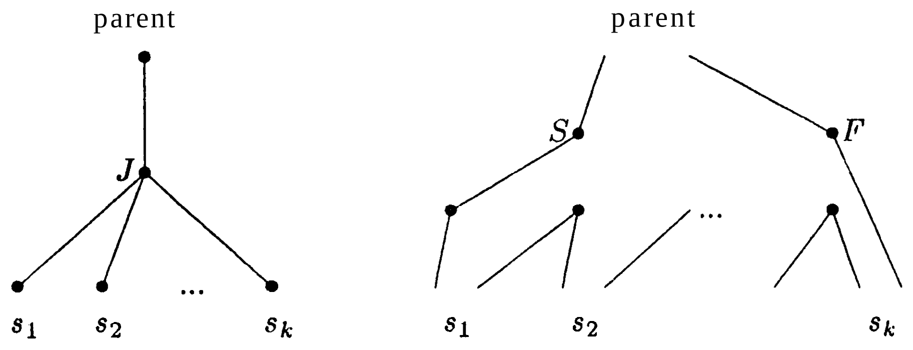


Figure 1: Node expansion to compute the tree traversal.

vertex's position in the path. This allows us to create an array containing the vertices in the order they appear in the path.

Each S-vertex in the path is paired with the F-vortex for the same node. Every node in the tree whose S-vertex appears before *m*'s F-vertex and whose F-vertex appears after m's F-vertex is an ancestor of m. Therefore, if we examine that portion of the path following *m*'s F-vertex, all of the unpaired F-vertices will belong to ancestors of m.

When the path leaves an F–vertex it moves up a level in the tree, when it enters an S–vertex it moves down a level. Thus, we assign the value 1 to all F-vertices, −1 to all S-vertices and 0 to all other vertices. Whenever the partial sum from m to some vertex *k* reaches a new maximum, we are leaving a level for the first time (since reaching node m). Therefore, *k*'s node must be an ancestor of node m. Furthermore, if *k* is an F–vertex for on of *m*'s ancestors, then the partial sum from m's F-vertex to *k* is a new maximum.

Now we can find all of m's ancestors. We first compute all partial sums from *m*'s F-vertex. Then we find the left to right maximum to the left of each position in the array. (This is done similarly to path doubling, except we keep track of the maximum instead of the distance.) Those locations where the maximum increases must contain an F- vertex from one of *m*'s ancestors.

The most expensive step of the Path Finding Algorithm is computing the expanded tree. If the original tree has n nodes, all other steps take O(log n) time on n processors. Computing the (pointer representation of the) expanded tree requires n^2 processors and O(log n) time. Thus the total requirements for the Path Finding Algorithm are n^2 processors and $O(\log n)$ time.

2.2 Definitions

An instance of the scheduling problem is:

a number of identical processors, m;

a set of n *jobs*, $\{J_1, J_2, \ldots . J_n\}$, each taking a single timestep to execute;

a partial order, $\prec$, on the set of jobs.

A solution to the scheduling problem is a schedule, S, which maps jobs to (integer) timesteps. The schedule must be legal, i.e. no more than m jobs are mapped to any one timestep and if $J \prec J'$ then $S(J) < S(J')$. Without loss of generality we assume $\min_i\{S(J_i)\} = 1$ and $\{t : t = S(J_i), i \in \{1, \ldots . n\}\}$ is some interval, $\{1, \ldots . L(S)\}$.

A job, J, *is available* at timestep t if all its *predecessors,* i.e. jobs J' such that $J' \prec J$, are mapped to timesteps less than t. A schedule, S can have several properties. The length of the schedule, L(S), is the number of timesteps which jobs are mapped to. If S maps $k < m$ jobs to timeslot t $(1 \leq t \leq L(S))$, then S has $m - k$ *empty slots* at timestep t. A schedule, S is *full* if it has no empty slots at timesteps $< L(S)$; S is *completely full* if it has no empty slots. It is *greedy* if there does not exist an empty slot at some timestep t and a job, J such that $S(J) > t$ and J is not a successor (directly or indirectly) of a job scheduled at timestep t. A schedule, S, is *optimal* if there is no legal schedule, S', with $L(S') < L(S)$. Every full schedule is optimal, but of course not every optimal schedule is full.

We use the partial order on the jobs to define other useful quantities. The *full precedence graph* is the directed acyclic graph with nodes representing the jobs and an edge going from J to J' iff $J \prec J'$. The sources are those jobs with no incoming edges. The *sinks* are those jobs with no outgoing edges. The *reduced precedence graph* or simply the *precedence graph* is the subset of the full precedence graph obtained by removing transitive edges. The precedence graph is an *inforest* if every node has at most one outgoing edge. The precedence graph is an *outforest* if every node has at most one incoming edge. An inforest or outforest can be trivially made into a tree by the addition of a dummy root job, tying the roots of the forest together. Although we use the term *tree* throughout this paper, the results apply to forests as well.

Definition EPT(J):

The earliest possible timestep a job J can be scheduled, EPT(J), is 1+ the length of a longest path in the precedence graph ending at J.

If we take a precedence graph and schedule all available jobs at each timestep (using up to n processors), then J will be scheduled at timestep EPT(J). Intuitively, EPT levels push jobs up as close to the sources as possible.

Definition d(P):

The *depth* of a precedence graph *P*, denoted *d(P)*, is the maximum EPT value over the jobs.

Definition $\mathbf{LPT}(J)$:

$\mathrm{LPT}(J)$, is *d(P)* --the length of a longest path from J to a sink.

If we have an unlimited number of processors available and we desire an optimal schedule, then each job must be mapped to a timestep at or before the job's LPT level. Intuitively, the LPT levels are obtained by pushing the nodes of P down (towards the sinks) as far as possible.

The i'th LPT level is $\{J : LPT(J) = i\}$. Each LPT level, i, partitions the jobs into three groups, those in the level, those above the level (with $\mathrm{LPT} < i$) and those below the level (with $\mathrm{LPT} > i$). EPT levels are defined similarly. We use N(i) to denote the number of jobs on LPT level i.

When *P* is an outtree, the values $d(P)$, $\mathrm{EPT}(J)$, and $\mathrm{LPT}(J)$ can be found by path doubling. First, if a tree node has *k* sons then we expand the node into a $k + 2$ node structure as shown in Figure 1. We then use the basic path doubling algorithm to find the path starting at node S of the root's structure, entering all of the nodes, and ending at node F of the root's structure. Everytime we hit an S node in this path we go down a level, and every time we hit an F node we go up a level. Therefore, we count S nodes as ones, F nodes as minus ones, and other nodes as zeros. Job J's EPT level is the prefix sum from the start of the path until just after J's S node. Job J's LPT level will be *d(P)* minus the maximum value of the prefix sums starting with J's S node and ending at J's F node. By listing the jobs in the order the path reaches their S (resp. F) nodes, we obtain a preorder (resp. postorder) traversal of the tree. A similar algorithm finds the EPT and LPT values for intrees. All these computations can easily be done in $O(\log n)$ time using n^2 processors.

The EPT (and *d(P)*) values can be computed more efficiently using a different input representation. When the tree is given by pointers to the parent and right brother for each node, the above transformation requires only O(n) processors with no significant loss of time. In [DUW84] there is an O(n) processor, $O(\log n)$ time algorithm for computing the LPT values as well.

Definition Highest Level First:

A schedule S is *highest level first* if it is greedy and there do not exist two jobs J_1, J_2 such that $\mathrm{LPT}(J_1) < LPT(J_2)$, $S(J_2) < S(J_1)$ and J_1 is available at timestep $S(J_2)$.

A highest level first schedule prefers jobs with the lowest LPT values. It is known that highest level first algorithms yield optimal schedules for both intree precedence graphs and outtree precedence graphs [Hu61, Bru81].

Definition $R(P)$:

The *reversal* of a precedence graph P, written $R(P)$, is the precedence graph containing:
(1) the same set of jobs as P;
(2) the edge (J, J') if and only if the edge (J', J) is in P.

In other words, the reversal of a precedence graph is created by reversing all of the edges in the graph. It is easy to see that $R(R(P)) = P$ and the reversal of an intree is an outtree. Any (optimal) schedule for an outtree can be reversed to form an (optimal) schedule for the associated intree (and vice versa), however, the resulting schedule will in general not be greedy.

Definition SIM:

The *Schedulability Interval Matrix,* SIM, is the $d(P) \times d(P)$ array of sets where $\mathrm{SIM}(i, j)$ is the set of jobs J having $\mathrm{EPT}(J) = i$ and $\mathrm{LPT}(J) = j$.

Later it will be important to order the jobs in each SIM cell. We compute a preorder traversal for the tree and order the jobs in each cell according to their positions in the traversal.

Lemma 1:

If $i > j$ then no jobs are in $\mathrm{SIM}(i, j)$.

Proof: Assume to the contrary that $i > j$ and some job J is in $\mathrm{SIM}(i, j)$. Then, $\mathrm{LPT}(J) < \mathrm{EPT}(J)$ or the latest possible time a job can be scheduled is less than the earliest possible time it can be scheduled; contradiction. ∎

Definition SIM **Path:**

A SIM path from cell (i, j) to cell (i', j') $(i' \leq i, j' \geq j)$ is a list of $i - i' + j' - j + 1$ SIM cells such that:
(1) cell (i, j) is the first cell on the list;
(2) cell (i', j') is the last cell on the list;
(3) if cell (i_1, j_1) immediately preceeds cell (i_2, j_2) on the list then it must be that either $(i_2, j_2) = (i_1 - 1, j_1)$ or $(i_2, j_2) = (i_1, j_1 + 1)$.

Thus a SIM path is a rectilinear path running from lower-left to upper-right. The number of jobs in a path is the sum of the number of jobs in each of the path's SIM cells.

Definition Directly **Above:**

Let p and p' be equal length SIM paths, then p is *directly above* p' if the lists differ by only one cell, with p having cell (i, j) and p' containing cell $(i+1, j+1)$.

Definition Above:

Path p_1 is *above* path p_k if there is a sequence of paths, $p_1, p_2, \ldots, p_k$, such that p_i is directly above $p_{i+1}, i = 1, \ldots, k-1$.

Definition **Reverse Running Average:**

We say that the *reverse running average* from some level (or column or row) i up or back to level (or column or row) j, is al *least (at most) c* if for all k $(j \leq k \leq i)$, the averages:

$$\frac{\sum_{l=k}^{i} \text{number of jobs on level } l}{i-k+1}$$

are at least (at mos t) c.

We can find the smallest integer greater than the reverse running average in $O(\log n)$ time on n processors by the following algorithm:

(1) Sum up the number of jobs on each row.
(2) Use the prefix sum algorithm to compute the number of jobs in the rows from i up to *each* $k \geq j$.
(3) For each of these k, find the smallest integer c_k such that:

$$c_k(i-k+1) \geq \sum_{l-k}^{i} \text{number of jobs on row } l.$$

(4) Compute the maximum of these c_k's.

Step (3) can be done in $O(\log n)$ time using only addition by computing the products $2^r(i-k+1), r = 1, \ldots \lfloor \log n \rfloor$. A similar algorithm computes the largest integer less than or equal to the running average.

3. Precedence Graphs With Many Components

Although the general scheduling problem is very hard [May8 1], there are some natural special cases which are easy. Two of these special cases arise when the precedence graph, P can be partitioned into several components, $C_1, C_2, \ldots, C_f$, with **no** edges joining nodes of different componenbs. We use n_i to denote the number of jobs in component C_i.

In this section we use the terms *schedule* and *scheduling* not only for the precedence graph P, but also for subgraphs of P.

Special case 1: The components are all small, i.e.

$$\max_i(n_i) \leq \left\lfloor \frac{n}{m} \right\rfloor.$$

Let $r = n - \lfloor \frac{n}{m} \rfloor \cdot m$.

The algorithm proceeds in three phases. The first phase linearizes the precedence constraints, creating a single list containing every job in P. This list is then partitioned into m *bins* in the second phase. In the third phase, the jobs within each bin are assigned timeslots in a manner consistent with the precedence constraints. These partial schedules can be merged to form a schedule for P.

Algorithm 1:

1. In the first pass, we linearly order the jobs in each C_i in any way which does not violate the precedence constraints. We record the position of each job in its list. These sublists for each C_i are concatenated to form L, a list containing every job in P.
2. Using prefix sums, we divide L between the m bins. We give each of the first r bins $\lfloor \frac{n}{m} \rfloor + 1$ contiguous jobs from L, and each other bin $\lfloor \frac{n}{m} \rfloor$ contiguous jobs from L. Let L_i be the sublist of jobs given to bin i.
3. We examine each L_i in parallel. If it contains a proper prefix of the jobs in some C_k, then that prefix is scheduled before anything else in L_i (using just the one processor). If L_i contains a proper suffix of the jobs in some C_k then we schedule those jobs after everything else in the bin. If L_i completely contains the jobs in C_k then those jobs in C_k can be scheduled whenever convenient.

Now we have m single processor schedules which can be merged to form an m processor schedule for P.

Theorem 1:

Algorithm 1 finds a legal, full schedule for P.

Proof: It is obvious that the resulting schedule is full, what we must show is that it does not violate the precedence constraints. Since there are no constraints between components, all we have to show is that the precedence constraints within components are met. Let L_i be the list for an arbitrary component (from phase 1). We show that the list is scheduled in order. If the entire component is placed in a single bin, then its jobs are scheduled according to L_i. The component can be split between at most two bins (no component has more jobs than a bin). Because L_i contains no more jobs than any bin and the first part of L_i is scheduled at the earliest, and the second part at the latest possible timesteps, the list will be scheduled in order. ∎

Special case 2: The precedence graph P has k components with at least d(P) jobs each and at least $(m - k) \cdot d(P)$ jobs in components with fewer than $d(P)$ jobs.

When this condition holds, P can be divided into m independent groups, each with at least $d(P)$ jobs. These groups have the property that their schedules can easily be combined to form an optimal schedule for P. The algorithm starts by creating the groups. By examining each group independently we create a temporary schedule, S_1, for P. The schedule S_1 satisfies the precedence constraints but may use far more than m processors. Using S_1, the algorithm sorts the jobs of P into a list. The final schedule, which is full and legal, is easily derived from this list.

Algorithm 2:

1. Divide P into m groups, each containing at least $d(P)$ jobs. Create temporary schedules for each of these groups.
   - (a) The k components of P having at least $d(P)$ jobs are each placed in their own group. Schedule each of these group such that the precede-nce constraints are met; each timestep from 1 to $d(P)$ has at least one job and no timesteps after $d(P)$ are used (any number of processors can be used with any number of empty slots; of course, at least one task must be scheduled at every timestep). For each of the above k components, this can be done as follows: Let C be such a component. First calculate the EPT levels of the jobs in C, and $d(C)$. Then find the highest numbered EPT level l in C with more than $d(P) - l + 1$ jobs at or after it. Spread these jobs so they cover the last $d(P) - l + 1$ timesteps, and schedule the jobs J with $\mathrm{EPT}(J) < l$ before them, level by level.
   - (b) The jobs in the remaining components are first linearized as in Algorithm 1. From the beginning of that list, $m - k$ sublists each of length $d(P)$ are taken away and rearranged as in step 3 of Algorithm 1. They form another $m - k$ "columns" of the temporary schedule S_1. Finally, the remaining jobs are put into one more "column", according to their LPT level (in P).
2. Now we have at least m temporary schedules of length $\leq d(P)$, and at least the first m of them have length exactly $d(P)$. We use these schedules to create an ordered list, L, of the jobs in P. The first jobs on L are those scheduled at timestep one in the temporary schedule, then those at timestep two, ... and those at timestep $d(P)$ are placed at the end of L. Within each timestep the jobs are ordered by groups.
3. We assign to timestep t the $(t-1)\cdot m+1$'st job through the $t \cdot m$'th job of L. Of course, the last timeslot may be only partially filled if the list does not contain a multiple of m jobs.

Theorem 2:

Algorithm 2 finds a full schedule for P.

Proof: The resulting schedule is obviously full. It only remains to check that the precedence constraints are satisfied. Jobs of the same component but in different groups are separated by a full timestep by S_1, and thus at least m jobs in L. This means that they cannot possibly be mapped to the same timestep by S. Since the temporary schedule satisfies the precedence constraints of P, the only other way the precedence constraints could be violated is if two jobs in the same group are mapped to different timesteps by S_1, but the same timestep by S. This can't happen because in L there are at least $m - 1$ jobs (one from each other group) between any two jobs of the same group. ∎

Resource usage: Finding the connected components takes $O(\log n)$ time using no more than $O(n^3)$ processors [SVi82]. The EPT levels can be found in $O(\log^2 n)$ time using $O(n^3)$ processors by the following procedure. First, we find the sources of the graph. Then using max-plus transitive closure we find the maximum distances in the precedence graph from each job to the sources. (After each successive "squaring" of the matrix, we create n new copies of the matrix using $O(\log n)$ time.) This gives us the length of the longest path from each source to each job. Taking the maximum over these lengths gives us the EPT levels of the jobs. The depth of the graph $(d(P))$ is the maximum EPT value.

We label each connected component by its least-numbered job. Every job is given a pointer to its, component number. Now, we can employ standard summing techniques to determine the number of jobs in each component.

4. An Outtree Scheduling Algorithm

This chapter describes our outtree scheduling algorithm. Section 4.1 presents several preliminary results for SIN's with outtrees. Section 4.2 describes a sequential algorithm for optimally scheduling outtrees. Section 4.3 shows how to compute this schedule quickly in parallel.

4.1 SIM for Outtrees

Let P be the outtree we are attempting to schedule. Let d be the depth of P.

Lemma 2:

There is only one job in $\bigcup_j SIM(1, j)$; **i.e.** the top row contains only the root of P. Those jobs in the last (d^{th}) column are the leaves of P.

Proof: Only the root has EPT=1. The leaves, and **only** the leaves, have LPT $= d$. ∎

Theorem 3:

If J is in $\mathrm{SIM}(i,j)$ then the parent of J is in $\bigcup_{l<j}\mathrm{SIM}(i-1,l)$; the children of J are in $\bigcup_{l>j}\mathrm{SIM}(i+1,l)$; and at least one child is in $\mathrm{SIM}(i+1,j+1)$.

Proof: The EPT of J's parent is exactly 1 less than J's EPT. The LTT of J's parent is at least 1 less than J's LPT. The EPTs of J's children arc one greater than J's EPT. The

LPTs of J's children are at least 1 greater than J's LPT. At least one of J's children is on the longest path from J to a leaf and therefore has EPT and LPT one greater than J. ∎

Corollary :

If SIM(i, j) $(i, j < d)$ contains r jobs, then SIM$(i + 1, j + 1)$ contains at least r jobs.

Proof: Every job in SIM(i, j) has a child in SIM$(i + 1, j + 1)$. ∎

Corollary :

If p and p' are paths where p is above p' then p' contains at least as many jobs as p.

4.2 A Sequential **Outtree** Algorithm

Definition SIM Schedule:

Given a SIM matrix we can compute the *SIM* schedule iteratively as follows: At the first timestep we schedule the job from the top row. The last cell of the first row is the corner for the first timestep. On the t^{th} timestep we attempt to schedule m jobs; we first schedule those jobs on the t^{th} row up to (and including) the previous timestep's corner. Then we schedule jobs from the next column to the right, working from the top to the bottom. If necessary, we use the left to right, ordering to break ties. If, in this column, we reach the t^{th} row before taking m jobs, then we start on the next column (again working from the top down), and so on. Within each SIM cell, we take jobs according to their preorder position. The corner for timestep t (denoted $CORNER(t)$) is the rightmost SIM cell on the t^{th} row whose jobs have all been scheduled by the t^{th} timestep. We will show below that this algorithm is optimal and does not violate the precedence constraints.

For convenience we define $CORNER(0)$ to be d. Since the algorithm finishes a column (down to the current timestep) before moving on to the next one, no jobs more than one column to the right of $CORNER(t)$ will be scheduled at timestep t. The example below shows how the SIM algorithm works.

Theorem 4:

The SIM schedule is highest level first and therefore optimal.

Proof: The SIM schedule is greedy. Clearly the first timestep schedules all available jobs (the root). Assume that until timestep t the schedule has been greedy and legal.

Case 1: The $CORNER(t - 1)$ is less than $d - 1$. This means that m jobs were scheduled in the previous timestep, none of which was in the last column. Each of these jobs has a son either on the t^{th} row, or in the $CORNER(t - 1) + 1^{st}$ or $CORNER(t - 1) + 2^{nd}$ column above the t^{th} row. The algorithm stops scheduling jobs only after these cells have been exhausted or m jobs have been taken. Since there are at least m jobs in these cells,

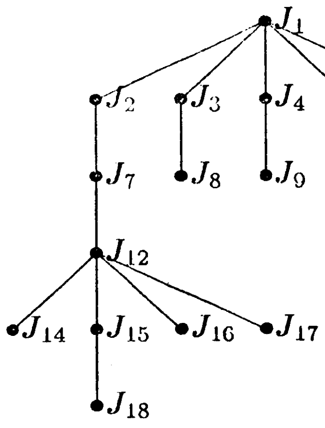
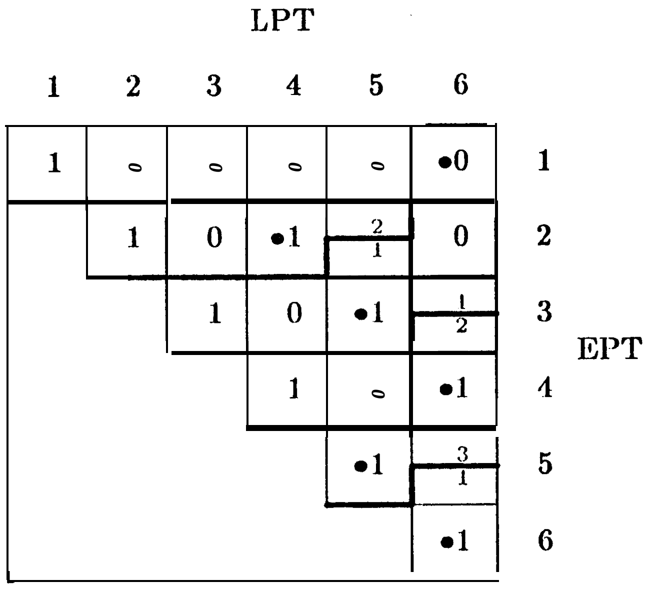


Figure 2a: The tree to be scheduled

Job#	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
EPT	1	2	2	2	2	2	3	3	3	3	3	4	4	5	5	5	5	6
LPT	1	2	5	5	5	4	3	6	6	6	5	4	6	6	5	6	6	6

Figure 2b: The jobs' EPT and LPT values.



$$\begin{array}{c|cccccc|c}
 & \multicolumn{6}{c|}{\text{LPT}} & \\
 & 1 & 2 & 3 & 4 & 5 & 6 & \text{EPT} \\
\hline
 & 1 & \circ & \circ & \circ & \circ & \bullet 0 & 1 \\
 & & 1 & 0 & \bullet 1 & \frac{2}{1} & 0 & 2 \\
 & & & 1 & 0 & \bullet 1 & \frac{1}{2} & 3 \\
 & & & & 1 & \circ & \bullet 1 & 4 \\
 & & & & & \bullet 1 & \frac{3}{1} & 5 \\
 & & & & & & \bullet 1 & 6
\end{array}$$

Figure 2c: The SIM matrix and scheduling for $m = 4$.

A • indicates the corner of that row. Heavy lines divide the sections of SIM scheduled at different timesteps.

the algorithm will schedule m jobs at timestep t.

The ordering within SIM cells ensures that all m sons of jobs scheduled at timestep $t-1$ will be scheduled before any sons of jobs scheduled at timestep t. Therefore the precedence constraints will not he violated.

Case 2: The corner of the previous timestep is d or cl -- 1. If the algorithm schedules less than m jobs, then all unscheduled jobs at or above the t^{th} row will ho taken. Since no jobs below the t^{th} row can possibly be available at the t^{th} timestep, all available jobs will be scheduled.

In this case, only jobs on the t^{th} row or in the last column are scheduled. Since no jobs below the t^{th} row arc schcdulcd, no dependents of jobs on the t^{th} row are schcdulcd. Since the jobs in the last column have no descendants, none of their descendants are scheduled. Therefore the SIM schedule obeys the precedence constraints.

Because the SIM schedule is greedy and it takes available jobs in the lowest LPT columns, it is highest level first. ▮

4.3 Parallel Implementation of' the SIM Algorithm

Once we know where the corners are, it is easy to find the SIM schedule in parallel.

Theorem 5:

The corner of a times tcp (or EPT row) t is the largest column num ber c ($c \leq d$) such that the reverse running average, counting only jobs in or before column c, from row t up to the first row is at most m.

Proof: The corner is no greater than the c computed above. If CORNER(t) were greater, then there is a row $t' \leq t$ such that there are *more* than $m \cdot (t - t' + 1)$ jobs in the rectangle bounded by column CORNER(t) and rows t and t' inclusive. All of these jobs must have been scheduled between timesteps t' and t, however there arc not enough slots for all of the jobs. Contradiction.

We now prove that the corner (when less than d) can not be smaller than the c computed above. Assume, to the contrary, that timestep t is the first timestep where the corner is less than the c computed above. Let t' be the last timestep before timestep t where the corner of t' is at least c. There arc at most $m \cdot (t - t')$ jobs on or to the left of column c and between rows t and $t' + 1$, inclusive. Since the corners of' all timesteps between t and 1' arc less than c, no jobs to the right of column c will be taken in those timesteps. Therefore, all of the $\leq$ m . . --- t') jobs to the left of column c will be taken. Thus the corner of timestep t is at least the c value computed above. ▮

The *extended* SIM array is an $n \times d$ array. The first d rows are identical to the rows of the SIM array. The other rows contain all zeros.

Algorithm: Our algorithm computes the SIM schedule for the outtrce in four phases. The first phase computes the extended SIM array. The second phase computes the corners for each timestep. The third phase computes the number of jobs scheduled from each column before each timestep. The final phase, using information from the first, phases, computes the specific jobs scheduled during each timestep.

The second phase computes the corners for each timestep. This phase consists of three steps:

a. Using prefix sums, find the matrix $ROWSUM(i, j)$, which contains the number of jobs on row j at or to the left of column i in SIM.
b. Create n copies of ROWSUM, one for each possible timestep.
c. For each timestep, t:
   1. For each column i of ROWSUM, compute the least integer greater than reverse running average of column i from t to the top.
   2. Find the rightmost column such that the reverse running average for that column is at most m. This column is the corner for timestep t.

The first step requires just n^2 processors and $O(\log n)$ time. Step b. can be done using n^3 processors and $O(\log n)$ time using binary trees to propagate the values. Step c. is done for n timesteps requiring $O(\log n)$ time and a total of n^3 processors. The entire phase takes $O(\log n)$ time on n^3 processors.

The third phase computes $TAKEN(t, c)$, the number of jobs scheduled from column c before timestep t. For each t we do the following steps:

a. For each c, compute $LAST(t, c)$, the last timestep before t when the corner was at or to the right of c.
h. Set $TAKEN(t, c)$ to the number of jobs in column c at or above row $LAST(t, c)$.
c. Additional jobs may be scheduled from column c when the corner is $c - 1$. Therefore, if $LAST(t, c-1) > LAST(t, c)$ then we add to $TAKEN(t, c)$ the amount by which $m \cdot (LAST(t, c-1) - LAST(t, c))$ exceeds the number of jobs in the rectangle bounded by column $c - 1$ and rows $LAST(t, c) + 1$ and $LAST(t, c-1)$ inclusive.

Claim: The value of $TAKEN(t, c)$ is the number of jobs scheduled from column c before timestep t.

Proof: Jobs can be schcdaled from column c only when the corner is at or to the right of column $c - 1$. Tf at time $t' < t$, the corner is to the right of column $c - 1$ then we know that all jobs in the column c at or before timestep t' have been scheduled. By finding the largest such t', we account for all jobs scheduled from column c during timesteps up to t'. Between t' and t the corner stays strictly to the left of c. By each timestep $t'' > t'$ when the corner is in column $c - 1 < d$, an additional $m \cdot (t'' - t')$ jobs have been scheduled. Exactly the number of jobs in the rectangle bounded by column $c - 1$ and rows $t' + 1$ and t'' (inclusive) do not come from column c, so $m \cdot (t'' - t')$ -- this number have come from c. By taking the latest approprintc t'', we ensure that all jobs taken form column c are counted. ∎

The third phase must be done for n values of t. For each of these values, the first step requires n processors and $O(\log n)$ time; step b, . . . , can be done in $O(\log n)$ time with n^2 processors; and step c takes n^2 processors only a constant amount of time. Thus the entire third phase takes n^3 processors $O(\log n)$ time.

Our outtree algorithm can be adapted for release-deadline scheduling problems. When EPT is replaced by release time and LPT by deadline, the intree algorithm computes a schedule minimizing the maximum tardiness.

5. An Intree Parallel Scheduling Algorithm

This chapter describes our intree algorithm. The first step in the algorithm is to divide the intree into two segments. Section 5.1 describes this division. The first segment can be easily scheduled using our outtree algorithm, this is also presented in section 5.1. Section 5.2 contains the basic theorems and definitions needed to schedule the other segment. Section 5.3 presents an optimal sequential algorithm for scheduling the second segment. Section 5.4 gives the parallel implementation of the algorithm.

5.1 Division of the Intree

This section shows how we divide the intree and schedule one of the segments. The division must be carefully chosen so that the schedules for the two segments can be merged into a greedy optimal schedule for the whole intree. This section closes with an obvious scheduling of the first segment.

Definition Cut:

A *cut* of a precedence graph, P, is a partition of P into two parts, P_1 and P_2 such that some optimal schedule maps all jobs of P_1 to timesteps at or before some timestep t, and maps all jobs of P_2 to timesteps after t.

Theorem 6 (Optimality Theorem):

If P is cut into P_1 and P_2, then any optimal schedule for P_1 can be concatenated with any optimal schedule for P_2 to create an optimal schedule for P.

Proof: If the concatenated schedule is not optimal then one of the two pieces must take more timesteps than the corresponding part of the whole schedule, and is therefore not optimal; contradiction. ∎

The algorithm works by partitioning the intree, T, into two segments, A, and B. The division is chosen so that both segments are easy to schedule independently, and once found, the two partial schedules can be merged to form an optimal schedule for T.

To End the A segment, we divide the tree at an LPT level, t. We pick t to be the largest LPT level such that the reverse running average from t up to the first LPT level is at least m. Those jobs at or above LPT level t form the A^+ segment. Let $\{J_1, J_2, \ldots, J_{N(t)}\}$ be the $N(t)$ jobs whose LPT is t. Reorder these jobs by EPT value so that J_1 has the least EPT value, J_2 has the second least, etc. The number of jobs in the A segment will be the largest multiple of m which is not greater than $\sum_{k<t}$ N(k). The jobs in the A segment are all those with LPT $< t$ and enough low-numbered jobs with LPT $= t$ so that A has the right cardinality. Let T_A be the precedence subgraph restricted to the jobs in the A segment. The B segment consists of all jobs not in the A segment; let T_B be the precedence subgraph containing the jobs of the B segment. An example is given in Figure 3.

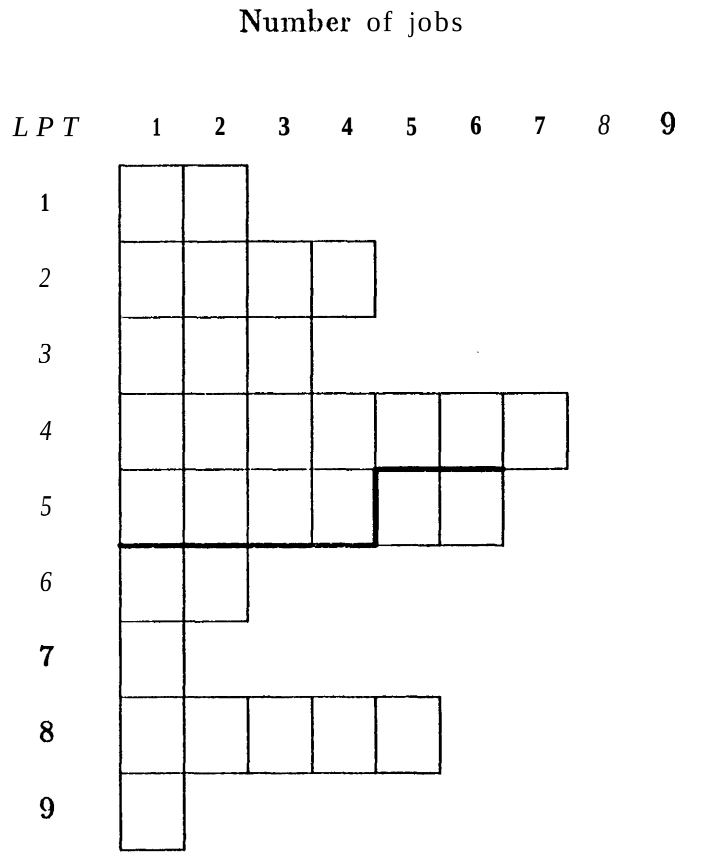


Figure 3: Division into A and B segments, m = 4

Assuming jobs on the same LPT level are sorted by EPT values, the jobs in this figure above the line are in the A segment and those below are in the B segment. The A^+ segment consists of all jobs at or above level 5.

We will show below that this partition forms a cut, and that there is a completely full highest level first schedule for the A segment. Using our outtree algorithm we can End an optimal schedule for It(A). This schedule can then be reversed to yield an optimal schedule for the A segment.

Lemma 3:

If the reverse running average from level d up to level l is at least k, then if some job is moved down from level i to level j $(l \leq i < j \leq d)$ the reverse running average from level d up to level l remains at least k.

Proof: The reverse running average is just a series of sums. When a job is moved down, it continues to be counted in all the sums where it originally appeared. ∎

Lemma 4:

If LPT level l of an intree has N(1) unscheduled jobs and level $l+1$ has $N(l+1) \geq$ N(Z) unscheduled jobs then there are at least $N(l+1) - N(l)$ available jobs on level $l+1$.

Proof: Since we have an intree, each job has at most one successor. Any job without an unscheduled predecessor on the previous LPT level is available. At most N(1) jobs on level $l+1$ have unscheduled predecessors. Therefore, at least the remaining $N(l+1) - N(l)$ jobs on level $l+1$ are available. ∎

Corollary :

If there are k unscheduled jobs on LPT level l, and $k' \geq k$ unscheduled jobs on a lower level, l', then there are at least $k' - k$ available jobs below level l and at or above level l'.

Lemma 5:

At any step (except possibly the last) in an HLF schedule for the A^+ segment, there are at least m leaves. Furthermore, if *1* nonempty LPT levels remain in the A^+ segment then the reverse running average from level $d(A^+)$ up to (and including) $d(A^+) - l + 2$ is at least m *jobs.*

Proof: : This obviously holds before the first timestep. Assume it holds through timestep $t-1$ when the $k-1$st (but not the kth) level has been completed. By the induction hypothesis, the reverse running average from level $d(A^+)$ to $k+1$ is at least m.

Case 1: If the kth level has more than m jobs, then no jobs are scheduled from below the kth level so the running average is still at least m.

Case 2: If the kth **level has less than m jobs,** and the $k+1$st **level** has m or more jobs then the $k+1$st level becomes the new top level. No jobs below this level are scheduled at timestep t. Therefore the running average from level d up to the $k+2$nd level is still at least m.

Case 3: If the kth level and the $k+1$st level both have less than m jobs then there must be some level with more than m jobs. Let level i be the highest level with more than m jobs.

Since, in step t, only tasks from levels between k and i are taken, the reverse running average is unchanged after step t for all levels lower than i. If it would drop below m for some level j, $k+2 \leq j \leq i$, this would imply that before step t the reverse running average for level $k+1$ was below m since all levels above i contain fewer than m jobs and m jobs are scheduled in timestep t. ∎

Given an inforest, P, examine the **EPT values of** the roots. **If** those jobs scheduled in the first timestep of any HLF schedule are removed, then the new EPT values will not increase. Furthermore, if the EPT value of a root decreases then the values for all roots with larger EPT values will also decrease.

Lemma 6:

Let P be an inforest with roots $r_1, r_2, \ldots . r_r$ such that $\text{EPT}(r_1) \leq EPT(r_2) \leq \ldots \leq EPT(r_r)$. There is a highest level first schedule for P which maintains these inequalities after every timestep.

Proof: By contradiction. Whenever we have a choice in the highest level first algorithm we schedule those jobs in the lowest numbered root's subtree first. Assume at some timestep the EPT value of root r_j becomes less than $\text{EPT}(r_{j-1})$. Since EPT values can decrease by at most one (and never increase) each timestep, the EPT values of r_j and r_{j-1} must have been equal at the previous timestep. This implies that all the ancestors of r_j at the highest level were scheduled, but at least one ancestor of r_{j-1} at that level was not. But that contradicts the algorithm's tie breaking method. ∎

Hence, the k leftover jobs in the last timestep must be the k highest numbered roots of the A^+ segment (therefore they can easily be found).

Corollary :

There is a completely full highest level first schedule of the reduced A segment.

5.2 Preliminary **Lemmas and Definitions**

The B segment algorithm uses the SIM matrix. Since the B segment is an intree, we prove several facts about SIM's for intrees.

Lemma 7:

Those jobs in $\bigcup_j \text{SIM}(1,j)$, i.e. the top row, are the leaves of T.

Proof: Every leaf, and only the leaves, have EPT = **1.** ∎

Lemma 8:

If J is in $\text{SIM}(i,j)$ then all of the immediate predecessors of J are in $\bigcup_{k<i}\text{SIM}(k,j-1)$.

Proof: Let J' be an immediate predecessor of job J. The path from J' to the root is one longer than the path from J to the root (it includes the arc (J', J)). The longest path from J' to a leaf is at most one less than the longest path from J to a leaf. Thus, $\mathrm{LPT}(J') = \mathrm{LPT}(J) - 1$ and $\mathrm{EPT}(J') < \mathrm{EPT}(J)$. ∎

Theorem 7:

If J is in $\mathrm{SIM}(i, j)$, with $i, j > 1$, then J has an immediate predecessor, J', in $\mathrm{SIM}(i - 1, j - 1)$ (the cell diagonally up and to the left).

Proof: By Lemma 7 we know that J is not a leaf. Examine any longest path from J to a leaf. Let J' be the job following J on the path. Then $\mathrm{EPT}(J') = \mathrm{EPT}(J) - 1 = j - 1$ and since J' is a direct predecessor of J, $\mathrm{LPT}(J') = \mathrm{LPT}(J) - 1 = i - 1$ (by Lemma 8). Therefore J' is in $\mathrm{SIM}(i - 1, j - 1)$. ∎

Corollary :

If cell (i, j), $i, j > 1$, contains k jobs then the cell $(i - 1, j - 1)$ contains at least k jobs.

Proof: Every job in cell (i, j) has a predecessor in cell $(i - 1, j - 1)$. Since we are dealing with an intree, these are all distinct. ∎

Corollary :

If path p is directly above path p' then p contains at least, as many jobs as p'.

Proof: The paths p and p' differ by only one SIM cell, p contains some cell (i, j) while p' contains cell $(i + 1, j + 1)$. From the above, cell (i, j) contains at least as many jobs as cell $(i + 1, j + 1)$. ∎

Corollary :

If path p is above path p' then p contains at least as many jobs as p'.

The most difficult portions of the B segment to schedule are the **LPT** levels containing more than m jobs. Since an optimal schedule progresses one LPT level each timestep (Lemma 15 below), those jobs in excess of m must be scheduled before the level is reached. The following algorithm meets this condition by special treatment of the columns in SIM with more than m jobs. First we give some definitions.

Definition Bad Column:

If an LPT column of the SIM matrix contains more than m jobs, then that column is a *bad column*.

Definition Bad Cells:

Cell (i, j) is a **bad** cell if there are at least m jobs in column j below row i. All other cells are *good* cells.

If a column has k bad cells, they will be the k topmost, cells of the column.

Theorem 8:

If cell (i, j), $i, j > 1$, is a bad cell, then so is cell $(i-1, j-1)$

Proof: Since cell (i, j) is a bad cell, the path from cell (j, j) to $(i+1, j)$ contains at least m jobs. The path from cell $(j-1, j-1)$ to $(i, j-1)$ is above this path, therefore it also contains at least m jobs. Thus there are at least m jobs in column $j-1$ below row $i-1$ so cell $(i-1, j-1)$ is a bad cell. ▮

Definition Barrier Cell:

If cell (i, j) is a bad cell and cell $(i, j+1)$ is not, then cell (i, j) is a *barrier cell.*

Definition Barrier Diagonal:

If an (upper left to lower right) diagonal in the SIM array contains a barrier cell then that diagonal is a *barrier diagonal.*

Lemma 9:

If cell $(1, j)$ is on a barrier diagonal then cell $(1, j)$ is a bad cell.

Proof: There is some bad column $c = j + k$ with at least $k+1$ bad cells if the diagonal is a barrier diagonal. Therefore, by Theorem 8, column c has at least 1 bad cell, so cell $(1, c)$ is a bad cell. ▮

Definition Flow:

. The *flow* is an ordering of the upper triangular SIM cells derived from the barrier positions. We define it inductively. The first layer of the flow starts in the first row of the first column. Let the last cell so far be cell (i, j). If (i, j) is on a barrier diagonal then the next cell is $(i-1, j)$, otherwise the next cell is $(i, j+1)$. When the next cell would be outside of the SIM array the layer ends. The i^{th} layer starts with cell (i, i), and proceeds as above.

Note that the flow order can easily be computed by the following algorithm: First map the SIM cells into a d by d array, sending cell (i, j) to cell $(i - B(i, j), j + B(i, j))$ (where $B(i, j)$ is the number of barrier diagonals below cell (i, j)). Each row of this new matrix is a flow layer. The flow order is found by concatenating the rows of the new matrix. From this mapping it is clear that the flow layers partition the cells of SIM.

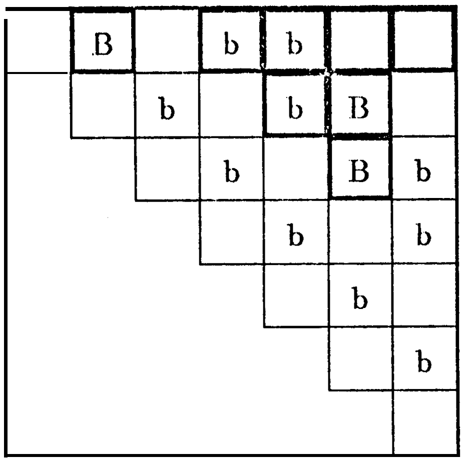


Figure 4a: Barrier cells and diagonals.

Cells with a heavy outline are bad cells. Those containing a "B" arc barrier cells. Cells containing a "B" or "b" are on a barrier diagonal.

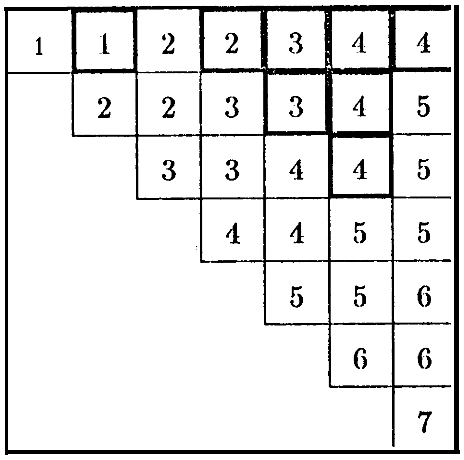


Figure 4b: Flow layers.

Each cell is labeled with its flow layer.

Lemma 10:

Once a flow layer enters a bad cell, it ends before entering another good cell.

Proof: Assume the flow layer is currently on a bad cell. If the flow layer goes up, it enters another bad cell (or leaves the array). If the flow goes to the right then wc arc not on a barrier diagonal, thus not on a barrier cell, and therefore not moving to a good cell. ∎

Lemma 11:

The last cell in each flow layer is either a bad cell or in the last column.

Proof: Flow layers end by leaving STM. They can leave by being in the top row at a barrier diagonal or by being in the last column at a non-barrier diagonal. By Lemma 9, if a cell on the top row is also on a barrier diagonal, then that cell is a bad cell. ∎

Lemma 12:

When flow layer f first enters a bad cell, then it enters from the left, not from the bottom.

Proof: Assume, by contradiction, that flow layer f is on a good cell, (i, j) and moves up onto the bad cell $(i-1, j)$. For this to happen there must be a barrier cell on (i, j)'s diagonal. Because of Theorem 8, there are no bad cells, and thus no barrier cells on (i, j)'s diagonal below (i, j). Also there are no good cells on $(i-1, j)$'s diagonal above (i, j), and thus no barrier cells on (i, j)'s diagonal above (i, j). Therefore (i, j) is not on a barrier diagonal and the flow moves right, not up. Contradiction. ∎

The flow order combined with the ordering within each cell is a complete ordering of the jobs in SIM. This ordering is consistent with the partial ordering given by the precedence constraints among the jobs.

Lemma 13:

For every (integer) value c, the first c LPT levels of the B segment contain less than $c \cdot m$ jobs.

Proof: Assume by contradiction that c is the first LPT level in the B segment where there are at least $m \cdot c$ jobs at or above c. Then for every level j, $1 \leq j \leq c$, of the B segment, there are less than $m \cdot (j-1)$ jobs above level j. This means that there are at least $m \cdot (c - j + 1)$ jobs between levels j and c inclusive, for every j. Therefore c would be in the A segment not the B segment. ∎

5.3 A Sequential B Segment Algorithm

Now we state a sequential algorithm for scheduling the B segment. Just before timestep t the $t-1$st column will have been completed and there will be a first unscheduled job from the flow order. At timestep t we schedule the remaining jobs from column t and as many additional jobs in flow order as possible (until on is reached or precedence would be violated). We prove below that this algorithm yields an optimal schedule.

We say the *flow has entered the rightmost column* when all jobs up to and including the first job with LPT $= d$ (in the flow ordering) have been scheduled.

Lemma 14:

The schedule is full until the flow enters the rightmost column.

Proof: Assume we are working on LPT column c and flow layer f. Let cell (i, j) be the first cell in f with unscheduled jobs. Let cell (i', j') be the first bad cell entered by f. Let path p go from cell (j', j') to cell $(i' + 1, j')$. Since (i', j') is a bad cell, there are at least m jobs in path p. Let path p' go from cell (c, c) to the first cell in flow level $f + 1$ and then along flow level $f + 1$ until it hits cell $(i' + 1, j')$. Since path p' is above p there are at least m jobs in path p', so there are at least m jobs that can be scheduled. If cell (i, j) precceds cell (i', j') in f, then for every job in p' which is not available, there is at least one job on flow level f available, so there are at least m total jobs for the algorithm to schedule. If cell (i, j) does not preceed cell (i', j') in f, then all of the jobs in path p' can be scheduled by the algorithm without violating precedence constraints. ∎

Lemma 15:

The algorithm completes an LPT column every timestep.

Proof: by contradiction. Assume that at timestep t the algorithm fails to complete LPT column t. Let f be the current flow layer. No flow layer before f contains a good cell from column t, because it would have been scheduled leaving less than m jobs in the column. By the same reasoning, if f contains a good cell from t, then that cell has not yet been completed. Therefore no good jobs to the right of column t have been scheduled. Since the flow has not yet reached the right edge of SIM (all layers before f end in bad cells), m jobs have been scheduled every timestep. Let c be the leftmost column from which jobs have been scheduled. Since no good jobs to the right of t have been scheduled, this column c and all columns between t and c contain bad cells. Therefore they all contain m good jobs. Thus there are $m \cdot (c - t + 1)$ unscheduled jobs at or before column c. In addition, there is at least one unscheduled bad job in column t. Adding these up we find that there are more than mc jobs at or to the left of column c, so c should be in the A segment rather than the B segment. Contradiction. ∎

Lemma 16:

The B segment algorithm is greedy.

Proof: Until the flow enters the rightmost column, the schedule is full. After the rightmost column has been entered, the algorithm takes either m or as many jobs in flow order as possible. Any job not scheduled is the descendant of some job scheduled, therefore it is greedy. ∎

Theorem 9:

The B segment algorithm yields a greedy optimal schedule for the B segment, Proof: The schedule is greedy and it progresses one LPT level each timestep. ∎

5.4 The Parallel B Segment Algorithm

Our B segment algorithm can be parallelized. The parallel version is stated below:

Parallel Algorithm for B segment:

```
1 Create the SIM matrix for $T_B$.
2 Find the bad cells of SIM.
3 Find the barrier cells of SIM.
4 For each job, $J$, in SIM compute $F(J)$ = the position of $J$ in the complete flow
  ordering.
5 Create the graph of pairs $(l, f)$ where $l$ is the last LPT level completely scheduled,
  and $f$ is the position in the flow last scheduled.
6 For each node (pair) in the graph above add an edge to the successor pair obtained
  by:
    a first scheduling the jobs on the next LPT level of SIM,
    b and taking the next jobs (until $m$ reached or precedence trouble encountered)
      from the flow ordering.
7 Use path finding to find the path through the graph starting from node (0,0).
8 From this path it is easy to compute the jobs scheduled at each timestep.
```

Only step 7 in the above algorithm takes more than n^3 processors to run in $O(\log n)$ time. Step 7 requires $(nd)^2 \leq n^4$ processors to complete in $O(\log n)$ time. Therefore the entire algorithm runs in $O(\log n)$ time on n^4 processors.

6. List Scheduling

Section 6.1 states the list scheduling problem and defines the *offset*. Section 6.2 exhibits a proof that the list scheduling problem is $\mathcal{P}$-complete under log-space reduction. This proof, like the max-flow proof uses exponentially large numbers. Section 6.3 contains an $\mathcal{NC}$ algorithm for the list scheduling problem, even when the execution times are rather large (i.e. their maximum number of bits is bounded by $O(\log^c n)$).

6.1 Problem Statement

A list scheduling problem instance consists of:
an ordered list of n jobs, $\{J_1, J_2, \ldots . J_n\}$;
and a positive integer execution time for each job, $T(J_i)$.
The jobs are to be executed on two (identical) processors.

A solution to the list scheduling problem is a (nonpreemptive) schedule mapping each job to a processor and start time. Any such mapping must satisfy the following properties:

1. no two jobs run on the same processor at the same time;

2. no job starts before any jobs preceeding it on the list;
3. both processors are in use until all jobs have been started.

There is a trivial sequential algorithm for this problem: simply start at the front of the list and deal the jobs out one by one to the processor whose jobs have the least total execution time. At each stage in this algorithm there is an *offset*, by which the execution times of one processor's jobs exceeds the others. Clearly, jobs J_i and J_{i+1} are mapped to the same processor only if the offset before J_i is at least $T(J_i)$. The *final offset* of a list scheduling problem is the difference in total execution times of the two processors.

Although we state the list scheduling problem for two target processors, our results generalize to any constant number of target processors (with some increase in the processor requirements of our $\mathcal{N}\mathcal{C}$ algorithms).

6.2 P-Completeness Result

Here we prove that computing the value of a circuit containing only NOR gates reduces to computing the final offset of a list scheduling problem. Technically, the circuit value problem is one of recognizing the set of all inputs which encode a circuit whose output is true. We can redefine the list scheduling problem as one of recognizing all properly encoded list scheduling instances where the final offset is non-zero. It is not hard to see that the reduction below also works for the set recognition problems.

Theorem 10:

The general list scheduling problem is polynomial-time complete.

Proof: By reduction from the circuit value problem [Gol77] for boolean circuits containing only NOR gates. The basic idea of this reduction is to encode the value of wires in the offset. We start with an offset encoding the true inputs. Each gate is represented by a chunk of jobs. Scheduling the chunk for a given gate modifies the offset so the inputs of the gate are no longer represented, but the outputs are.

We use the term wire in this construction to mean the connection between gates. Therefore, although each gate has only a single output, it may have several output wires.

The construction starts by topologically numbering the gates, with the gate generating the output getting 1. The output wire is labeled with the value 4. The input wires of gate i are labeled 4^{2i} and 4^{2i+1}. We define V_i to be the sum of the labels on all output wires of gate i. For each gate we create the following gadget:

A chunk of 17 jobs with times: one at $2 \cdot 4^{2i+1}$, fourteen at $\frac{4^{2i}}{2}$, and two at $\frac{4^{2i}+V_i}{2}$.

The list of jobs starts with one whose execution time equals the sum of all true circuit input wire labels. Then the gadgets for each gate appear (in descending gate order). We shall see that, after each chunk, the offset equals the sum of all dangling true wires. The

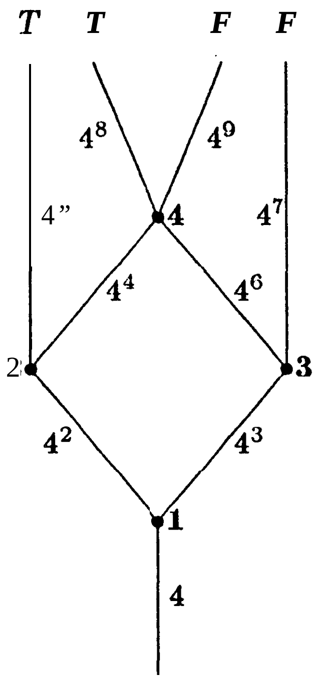


Figure 5: Labeled circuit, nodes represent NOR gates.

final offset will be four (zero) iff the output of the boolean circuit is true (false).

The idea behind the chunks is that if the offset before the chunk is small enough (both inputs arc false) then the offset after the chunk's firs6 job will be large. If one or both inputs are true, then the offset after the first job is small. The next fourteen jobs reduce small offsets 60 the low order bits, so the last two jobs' effects on the offset cancel. When the offset is large enough, the fourteen jobs do not entirely reduce it, so the $V_i/2$ terms in the last two jobs combine to affect the offset.
The above table shows the effects of a chunk on the offset for all possible input combinations. The low order bits (LOB) are modified only when both inputs arc false, and those bits cncoding the gate's inputs are always consumed. Since chunks are placed on the job list in reverse gate order, the most significant bits of the offset before a chunk will always encode the inputs 60 that chunk's gate. Therefore the construction will faithfully mimic the evaluation of the gates so that the final offset will be 4 if and only if the output of the circuit is true.

Now we must show that the reduction can be done in $O(\log n)$ space. Since wc assume the gates are given in topological order, the topological numbering can be done in logarithmic space, as cm determining if two gates are joined by a wire. These procedures are repeated each time one of their values is needed.

When wc are to output a bit of the first job (initial offset), we simply look to see if the

TT		TF		FT		FF		next job	
01010	LOB	01000	LOB	00010	LOB	00000	LOB	**10000**	**0**
00110	−LOB	01000	-LOB	01110	-LOB	10000	−LOB	00001	0
00101	-LOB	00111	−LOB	01101.	-LOB	01111	−LOB	00001	0
00100	−LOB	00110	-LOB	01100	−LOB	01.110	-LOB	00001	0
00011	-LOB	00101	-LOB	01011	--LOB	01101	-LOB	00001	0
00010	-LOB	00100	-LOB	01010	-LOB	01100	-LOB	00001	0
00001	-LOB	00011	−LOB	01001	-LOB	01011	-LOB	00001	0
00000	+LOB	00010	-LOB	01000	-LOB	01010	-LOB	00001	0
00001	-LOB	00001	−LOB	00111	-LOB	01001	-LOB	00001	0
00001	-LOB	00001	-LOB	00001	-LOB	00011	-LOB	00001	0
00000	+LOB	00000	+LOB	00000	+LOB	00010	-LOB	00001+$V_i/2$	
00001+$V_i/2$	−LOB	00001+$V_i/2$	−LOB	00001+$V_i/2$	−LOB	00001	−LOB−$V_i/2$	00001+$V_i/2$	
00000	+LOB	00000	+LOB	00000	+LOB	00000	+LOB+V_i		

Figure 6: The effects of a chunk.

appropriate wire contains a true input. Thus we need only keep track of the gate number we are working on. While we are generating the chunks, we need to keep the current gate number. The value $V_i/2$ can be computed each time it must be written by the same procedure which computes the initial offset. Since gate numbers can be stored in binary, the entire reduction can be done in $O(\log n)$ space. ∎

6.3 An NC Algorithm

In this section we give a fast parallel algorithm which deterministically solves the list scheduling problem. The algorithm works by computing an estimate for the offset after each job, and then iteratively improving the estimate. If the largest $T(J)$ is k bits long, then we represent all offsets and job times as k bit numbers. The precision of an offset estimate is the number of leading bits, from the k-bit job times, used to compute the estimate.

Theorem **11:**

When the job times are bounded by $2^{L(n)}$ for some function L, then the list scheduling problem can be solved in $O(L(n) \cdot \log n)$ time using n^4 processors.

Proof: We can estimate the first log n bits of the offset by path finding. This is done by examining each of the n possible offset estimates before each job and computing the next offset estimate. This uses n^4 processors and $O(\log n)$ time to give us log n bits of precision.

Increasing the precision by one bit changes any offset estimate by at most $\pm n$. For each position and $2n - 1$ possible changes, we can compute the change in the next position. Using path finding we can determine the effect of the additional bit of precision on each offset. Adding one bit of precision thus takes $O(\log n)$ time and n^4 processors.

We can iterate the above process $O(L(n))$ times to recover the entire offset before each job. Using these offsets it is trivial to compute the start time and processor for each job. Therefore the entire process takes $O(L(n) \cdot \log n)$ time on n^4 processors. ∎

Corollary:

If $L(n) = O(\log^c n)$ for some constant c, the list scheduling problem can be solved in $O(\log^{c+1} n)$ time on n^4 processors, and is thus in NC.

When concurrent reading is allowed, path doubling can be used instead of path finding. In this case the algorithm requires only n^2 processors.

7. Conclusions

The list scheduling result provides a very simple problem which is P-complete under logspace reductions. It uses large numbers as does the reduction for the max flow problem [GSS82]. In addition, the parallel time complexity of the list schcdnling problem is, in one direction, closely tied to the size of the numbers in a problem instance since the analysis of our algorithm shows that this size (measured in maximum number of bits) basically yields the parallel running time. It should be interesting to study whether, in the other direction, lower bounds greater than log n can be obtained. This problem, however, seems to be very hard in general, and no progress has been made so far.

Our intree and outtree algorithms are based on a common data structure, the schedulability interval matrix. While, in sequential computation, it is straightforward to derive a greedy optimal schedule for an inforcst given an optimal algorithm for outforests, this need not be true for fast parallel algorithms. In fact, for slight variations of these problems, such as outtrees on a varying number of processors, greedy algorithms become non-parallelizable [DUW84]. There are many other examples of problems with good sequential greedy algorithms which can not be parallelized [AMa84].

It is now known that scheduling problems with either intree or outtree precedence constraints are in $\mathcal{NC}$. However, there are still several classes of scheduling problems, such as chordal graph precedence constraints or other kinds of scheduling problems with empty precedence constraints, which are not known to be either $\mathcal{P}$-complete or in NC.

References

[AMa84] Anderson, R..; Mayr, E.: Parallelism and Greedy Algorithms
TR STAN-CS-84-1003, CS Dept. Stanford University (1984).

[Bru81] Bruno, J.; Deterministic and stochastic scheduling problems with treelike precedence constraints. NATO Conference, Durham England, (198 1)

[DSa81] Dekel, E.; Sahni, S.: Parallel scheduling algorithms.
TR81-1, Dcpt of Computer Science, U. of Minnesota, Minneapolis, Minn. (1981)

[DUW84] Dolev, D.; Upfal, E.; Warmuth, M.: Scheduling trees in parallel.
Proc. Internation Workshop on Parallel Computing and VLSI,
Amalfi, (1984), pp. l-30.

[FWy78] For tune, S.; Wyllic, J.: Parallelism in random access machines.
Proc. 10th ACM STOC (1978), pp. 114-118

[Gol77] Goldschlager, L.M.: The monotone and planar circuit value problems are LOG SPACE complete for I? SIGACT News 9,2 (1977), pp. 25-29

[GSS82] Goldschlager, L.M.; Shaw, R.A.; Staples, J.: The maximum flow problem is log space complete for P. Thcorctical Computer Science, 21 (1982), pp. 105-111

[Hu61] Hu, T.C.: Parallel sequencing and assembly line problems.
Operations Research 9 (1961), pp. 841-848

[May8 1] Mayr, E.: Well structured programs are not easier to schedule.
TR STAN-CS-81-880, Computer Scicncc Dept. Stanford University (1981).

[Pip75] Pippenger N.: The complexi ty theory of switching notworks.
Technical Report, Research Lab of Electronics, MIT (1975)

[SVi82] Shiloach, Y.; Vishkin, U.: An O(log n) parallel connectivity algorithm.
J. of Algorithms 3,1 (1982), pp. 57-67

[Ul175] Ullman, J.D.: NP-complete schctluling problems.
J. Compu t. System Sci. 10 (1975), pp. 384-393