1988

An Optimal Parallel Algorithm for Preemptive Job Scheduling that Minimizes Maximum Lateness

Susan Rodger

Report Number:
88-798

AN OPTIMAL PARALLEL ALGORITHM FOR PREEMPTIVE JOB SCHEDULING THAT MINIMIZES MAXIMUM LATENESS

Susan Rodger

CSD-TR-798
August 1988

An Optimal Parallel Algorithm for Preemptive Job Scheduling that Minimizes Maximum Lateness

Susan Rodger †

August 8, 1988

Department of Computer Sciences

Purdue University

West Lafayette, IN 47907

ABSTRACT

In this paper we present the Preemptive Minimize Maximum Lateness algorithm, an optimal parallel algorithm that schedules jobs for execution on a single processor machine with preemption. Each job is described by a release time, a deadline and a processing time. A job is considered late if it does not complete by its deadline and its lateness is defined as the difference between its completion time and deadline. Using the CREW PRAM model, our algorithm schedules n jobs for execution and minimizes the maximum lateness of the jobs. Our algorithm runs in $O(\log n)$ time and uses $O(n)$ processors, which is the first optimal processor-time product algorithm for this problem. The technique we use can be applied to other job scheduling problems, improving their results. We give faster parallel algorithms for three additional scheduling problems.

---

† This research was partially supported by the Office of Naval Research on contract N 00014-86-K-0689.

An Optimal Parallel Algorithm for Preemptive Job Scheduling that Minimizes Maximum Lateness

1. Introduction

In this paper we present an optimal parallel algorithm that preemptively schedules jobs for execution on a single processor machine. Each job is described by a release time, a deadline and a processing time. A job is considered late if it does not complete by its deadline and its lateness is defined as the difference between its completion time and deadline. Using the CREW PRAM model, our algorithm schedules n jobs for execution and minimizes the maximum lateness of the jobs. Our algorithm runs in $O(\log n)$ time and uses $O(n)$ processors, which is the first optimal processor-time product algorithm for this problem. The technique we use can be applied to other job scheduling problems, resulting in faster parallel algorithms for these problems.

There are efficient sequential algorithms for many scheduling problems, but few parallel algorithms for these problems. Sequential job scheduling algorithms that minimize the maximum lateness have been studied extensively. If preemption is allowed, scheduling jobs to run on one machine is solvable in polynomial time [Hor74]. If preemption is not allowed, scheduling jobs to run on one machine is NP-hard [BLR77], but polynomial time algorithms exist if all release times are equal [Jac55], all deadlines are equal [Jac55], or all processing times are equal [Sim78] [GJST81]. Recently, parallel algorithms for job scheduling problems are being studied ([DUW86], [DeS83a], [DeS83b], [DeS84], [HeM84], and [Mar88]). In particular, Dekel and Sahni [DeS83a] give parallel algorithms for minimizing the maximum lateness of jobs scheduled to run on m identical machines. The specific problems they examine are characterized by 1) the jobs have unit processing times and integer release times, 2) the scheduling is on $m = 1$ machine and preemptions are allowed, and 3) cases 1 and 2 with precedence constraints. In this paper, we improve on their results for cases 1 and 2.

This paper concentrates on minimizing the maximum lateness for the case when jobs are scheduled to run on one machine and preemption is allowed. For this problem, Dekel and Sahni [DeS83a] give a parallel algorithm that is based on parallel divide-and-conquer.

The bottleneck in their algorithm is that it takes $O(\log n)$ time to combine two subproblems. There are $\log n$ levels of subproblems, so their algorithm takes $O((\log n)^2)$ time. Our algorithm, the Preemptive Minimize Maximum Lateness algorithm, improves the running time by pipelining the subproblems. It consists of two phases. In phase one we apply the pipeline merge technique of Cole [Col86] to the first part of Dekel and Sahni's algorithm. Subproblems on all levels calculate their solutions at the same time, so phase one completes in $O(\log n)$ time. The difficulty in pipelining the subproblems lies in determining preempted jobs at intermediate steps, since only a sample of the final solution set is available. At the intermediate steps, the preempted job is estimated. As the algorithm proceeds, the estimate improves and it converges on the exact job when all the jobs in a subproblem are available. In phase two of our algorithm, we cannot directly apply the pipeline merge technique to the second part of Dekel and Sahni's algorithm. Instead, we design a new algorithm for this phase to which we can then apply Cole's pipeline merge technique. Dekel and Sahni's algorithm runs in $O((\log n)^2)$ time and uses $O(n)$ processors. Our algorithm reduces their running time by $O(\log n)$ while using the same number of processors.

The Preemptive Minimize Maximum Lateness algorithm has an optimal processor-time product. Let the fastest known sequential algorithm for a problem have a running time of $t_s(n)$ and a parallel algorithm for the same problem have a running time of $t_p(n)$ and use $p(n)$ processors. The parallel algorithm is a *fast* parallel algorithm if $O(t_s(n)) = O((t_p(n)) * (p(n)))$. Furthermore, if $t_s(n)$ is the lower bound for the problem, then the parallel algorithm is *optimal*. An adversary argument [Fre88] shows that the Preemptive Minimize Maximum Lateness problem has $\Omega(n \log n)$, so our parallel algorithm is optimal.

The model we use is the shared memory model (SMM) parallel RAM (PRAM). There are n processors (PE's) indexed $PE_1, PE_2 \ldots PE_n$. Each PE knows its index and performs simple, syncronized operations. All PE's access a common memory. In the CREW (concurrent reads, exclusive writes) PRAM model, PE's can simultaneously read from the same memory location, but writes exclude all accesses other than the PE performing the write.

This paper is organized as follows. Section 2 reviews previous work that is relevant to

our algorithm. Section 3 presents the Preemptive Minimize Maximum Lateness algorithm. Section 4 applies our approach to other job scheduling problems, improving their running times. The problems in section 4 are 1) schedule n jobs with integer release times and unit processing times to minimize maximum lateness, 2) schedule n jobs with release times equal to zero and unit-processing times to minimize the sum of weights of tardy jobs, and 3) schedule n jobs with release times equal to zero and unit processing times to minimize the number of tardy jobs.

2. Definitions and Previous Results

In this section we formally define the scheduling problem for minimizing the maximum lateness. Next, we review Dekel and Sahni's [DeS83a] parallel algorithm and then we review Cole's [Col86] sorting algorithm. In section 3, phase one of our algorithm applies Cole's parallel merge technique to the first part of Dekel and Sahni's algorithm and then concentrates on estimating which jobs are preempted.

2.1. Definitions

Allowing preemption, we want to schedule n jobs to be processed on one machine so that we minimize the maximum lateness of the jobs. Each job i has an associated release time r_i, deadline d_i and processing time p_i. A job cannot start earlier than its release time and it should finish by its deadline. By allowing preemption, jobs can run in time slices, with the sum of the time slices equaling the job's processing time. A job is late if it completes after its deadline. Let c_i be the completion time for job i, then the lateness of job i is defined by $c_i - d_i$. We want to schedule the jobs so that we minimize the maximum $c_i - d_i$ over all jobs i.

A sequential algorithm for this problem runs in $O(n \log n)$ time [Hor74]. The algorithm scans over ordered release times, selecting available jobs with minimum deadlines for the schedule. A job j is *available* at r_i if $r_j \leq r_i$ and j has not been selected. The algorithm begins by sorting the jobs in nondecreasing order by release times and numbering them so that $r_1 \leq r_2 \leq \ldots \leq r_n$. Starting with r_1, scan over the ordered release times selecting jobs for the schedule. At each r_i, select available jobs with minimum deadlines until the

sum of the processing times of the selected jobs equals or exceeds $r_{i+1} - r_i$, or there are no more available jobs. If the sum is exceeded, split the last job selected into two new jobs. Its processing time is split so that one of the new jobs is selected, forcing the sum of the processing times of the selected jobs to equal $r_{i+1} - r_i$. The other new job is available for selection at larger release times. At the end of the algorithm, the jobs in the order they were selected form the desired schedule.

2.2. Dekel And Sahni's Parallel Algorithm

Using the EREW (exclusive read, exclusive write) PRAM model, Dekel and Sahni [DeS83a] give a parallel solution to the preemptive minimize maximum lateness problem. In this section, we review Dekel and Sahni's algorithm. First, we give some definitions and then we review the two phases of their algorithm.

Dekel and Sahni's algorithm uses parallel divide-and-conquer, dividing the problem into subproblems, solving the subproblems in parallel, and then combining their results. A complete binary tree is used to illustrate the order the subproblems are combined. The jobs are ordered by nondecreasing release times and divided into groups that have jobs with the same release time. The groups are assigned to the leaves of the binary tree, one group per leaf, in the release time order.

An interval is associated with each node in the tree. This interval is based on release times and it represents available time for scheduling jobs. If V is a leaf node that contains jobs with release time r_i and the leaf node to its right contains jobs with release time r_j, then the interval $[r_i, r_j)$ is associated with V. We can schedule some of the jobs in V to run during this interval such that the total time for processing the scheduled jobs is less than or equal to $r_j - r_i$. The rightmost leaf, which has release time $r_{n'}$, represents the interval $[r_{n'}, \infty)$. If V is an internal node, it is associated with the interval $[r_i, r_j)$ where $[r_i, r_k)$ is the interval associated with V's left child and $[r_k, r_j)$ is the interval associated with V's right child. The intervals are designed so that for any job z assigned to a leaf in V's subtree, $r_i \leq r_z < r_j$.

Phase one begins by assigning one processor per job and in parallel sorting the n jobs by nondecreasing release times. The jobs are then divided into groups by release times

and the groups are assigned to the leaves of the tree. In parallel, the jobs within a group are further sorted by nondecreasing deadlines. Each node V calculates two ordered lists of jobs, $SCHED_V$ and REM_V. $SCHED_V$ is the set of jobs, considering only those jobs in V's subtree, that can be scheduled in the interval associated with V to minimize the maximum lateness. REM_V is the set of remaining jobs from V's subtree that are not in $SCHED_V$. Both $SCHED_V$ and REM_V are in sorted order by deadlines, since their solutions can be calculated quickly in $\log n$ time by merging sets sorted by deadlines. The calculation does not produce the sets in their scheduled order. If V is a leaf node then all its jobs have the same release time r_i and the jobs are sorted by nondecreasing deadlines. The interval associated with V is $[r_i, r_j)$. Let $j_1, j_2, \ldots j_t$ be the jobs in V in sorted order. The jobs in V are split into the two sets $SCHED_V$ and REM_V. The partial sums of the processing times are calculated to determine where this split occurs. Let z be the index such that

$$\sum_{h=1}^{z} p_{j_h} \leq r_j - r_i \quad and \quad \sum_{h=1}^{z+1} p_{j_h} > r_j - r_i$$

The jobs $j_1, j_2, \ldots j_z$ are placed into $SCHED_V$. If the first summation above does not equal $r_j - r_i$ and job j_{z+1} exists, then j_{z+1} is split into two new jobs. Its processing time is split so that one of the new jobs can be added to $SCHED_V$, resulting in the sum of the processing times of jobs in $SCHED_V$ equal to $r_j - r_i$. The other new job and all the remaining jobs not chosen are placed into REM_V.

If V is an internal node, then $SCHED_V$ and REM_V are calculated using V's left child and right child, $left(V)$ and $right(V)$. V is associated with the interval $[r_i, r_j)$, $left(V)$ with the interval $[r_i, r_k)$ and $right(V)$ with the interval $[r_k, r_j)$. Assume $SCHED_{left(V)}$, $REM_{left(V)}$, $SCHED_{right(V)}$, and $REM_{right(V)}$ have already been calculated. Every job in $right(V)$ has release time r_z such that $r_z \geq r_k$, so $REM_{right(V)} \subset REM_V$. Similarly, $SCHED_{left(V)} \subset SCHED_V$. Of the jobs in $SCHED_{right(V)}$ and $REM_{left(V)}$, choose the jobs with minimum deadlines that use at most $r_j - r_k$ total processing time. This is done as follows. Let $W = SCHED_{right(V)} \cup REM_{left(V)}$ where $\cup$ is a merge operation in order of nondecreasing deadlines. Calculate the partial sums of the processing times in W and using these split W into the two sets $SCHED_W$ and REM_W. W is split in the same manner the

leaf nodes were split, so that the sum of the processing times of the jobs in $SCHED_W$ is less than or equal to $r_j - r_k$. This may involve splitting one job into two new jobs as before. REM_W will contain the remaining jobs. Then $SCHED_V = SCHED_{left(V)} \cup SCHED_W$ and $REM_V = REM_W \cup REM_{right(V)}$. $SCHED_V$ will contain the set of jobs that minimize the maximum lateness for V's interval, considering only the jobs in V's subtree. If $ROOT$ is the root of the tree, $SCHED_{ROOT}$ will contain all the jobs in nondecreasing sorted order by deadlines and REM_{ROOT} will be empty. Note that the jobs in $SCHED_{ROOT}$ are not necessarily in scheduled order, since each $SCHED$ and REM set calculates the jobs in the sets, but never the scheduled order of the jobs. At most one job per node is split into two jobs, so there are $O(n)$ jobs at the root.

In phase two, Dekel and Sahni calculate the solution set $SCHED$ at each node so that a solution at a node considers all the jobs, and not just the jobs in its subtree. They start at the root of the tree and proceed downward by levels. Suppose a node V has an updated $SCHED_V$ set, call it NEW_$SCHED_V$. V has a left child $left(V)$ and a right child $right(V)$ and the interval associated with V is $[r_i, r_j)$. To calculate NEW_$SCHED_{left(V)}$, they extract all the jobs from NEW_$SCHED_V$ whose release times are smaller than r_i. These are the jobs that cannot be scheduled between $[r_1, r_i)$ and were not considered for $SCHED_{left(V)}$ in phase one. They merge these jobs with $SCHED_{left(V)}$, and then split the resulting set so that the sum of the processing times of the jobs in the left half of the split is equal to $r_j - r_i$. Like the splits in phase one, this may involve job splitting. The left half of the split forms NEW_$SCHED_{left(V)}$. NEW_$SCHED_{right(V)}$ can be calculated from the set NEW_$SCHED_V$ − NEW_$SCHED_{left(V)}$. The final schedule will appear in the NEW_$SCHED$ sets of the leaves. We give a different algorithm for phase two that does not use extractions. Our algorithm is designed so that we can apply Cole's parallel merge sort to it.

Dekel and Sahni's algorithm proceeds by levels, at each level merging sorted lists, calculating partial sums of processing times, or performing extractions, spending $O(\log n)$ time per level and resulting in an overall time of $O((\log n)^2)$. Our algorithm reduces the total time to $O(\log n)$.

2.3. Cole's Sorting Algorithm

This section reviews Cole's parallel algorithm for sorting n elements in $O(\log n)$ time using $O(n)$ processors. The key point in Cole's algorithm is that two sorted lists are merged together by pipelining the merge of samples of the two lists, starting with samples of one element and then doubling the sample size at each step in the pipeline. Each merge of two sample lists uses information from the previous merge in the pipeline to calculate its merge in constant time. Cole defines the sorting algorithm for a complete binary tree with n a power of 2. In [ACG87] the sorting technique is generalized for an arbitrary binary tree, where there is at most one element per leaf.

We begin with some definitions. Let L and J be sorted arrays of elements. Suppose that f is an element in J and e and g are adjacent elements in L, $e < g$, such that f falls between e and g, i.e. $e \leq f < g$. Then the *rank* of f in L is defined to be the rank (or position) of e in L. L is a *c-cover* of J if for any two adjacent items e and g in L, there are at most c items in J whose rank is e.

Cole's algorithm works in the following way. The n elements are assigned to the leaves of a complete binary tree, one element per leaf. For each node V we want to calculate the sorted list of all the elements in its subtree. Merging together the sorted lists of its children, $left(V)$ and $right(V)$, would take $O(\log \log n)$ time per level [Val75]. Instead, pipeline samples of the lists, starting with a sample of size one, and doubling the size of the sample each time until the whole list is sent. Two sample lists are merged together quickly by using the merged lists of the previous two sample lists in the pipeline as a guide. Instead of processing the data one level at a time, all the levels pipeline merge their sample lists at the same time.

Let $V(s)$ be the current list for node V at stage s in the pipeline, formed by merging together sample lists of V's children. $V(s+1)$ is the next list to calculate at V at stage $s+1$, $V(s-1)$ is the list at V one step earlier at stage $s-1$, and SAMP(L) is a sample of list L. To merge the two lists SAMP($left(V)(s)$) and SAMP($right(V)(s)$) together to form $V(s+1)$, each element in SAMP($left(V)(s)$) calculates its rank in SAMP($right(V)(s)$). Then its rank in $V(s+1)$ is the sum of its rank in SAMP($left(V)(s)$) and its rank in SAMP($right(V)(s)$). Similarly, each element in SAMP($right(V)(s)$) calculates its rank in

SAMP($left(V)(s)$) to determine its rank in $V(s+1)$. Cole shows that for all lists $V(s)$ and $V(s-1)$, $V(s-1)$ is a 3-cover of $V(s)$. Thus, $V(s)$ is a 3-cover of SAMP($left(V)(s-1)$) and SAMP($right(V)(s-1)$), so an element in SAMP($left(V)(s)$) can calculate its rank in SAMP($right(V)(s)$) in $O(1)$ time by using $V(s)$ and SAMP($left(V)(s-1)$).

To use a linear number of processors, the number of elements being merged together at one time step must be $O(n)$ elements. Cole achieves a linear number of processors by defining a sample list as every fourth element of the current list in the pipeline. Thus, when the algorithm starts, every node in the tree will begin constructing its sorted list of elements as soon as one element arrives. Until the list is complete, the node will always send to its parent a sample list of every fourth element of its current list. When a node has its complete sorted list, the node is *external*. In the next three time steps it will send every fourth element, then every second element, and then every element. Cole shows that every three time steps, another level becomes full, so that the root contains the complete sorted list after $3 \log n$ time steps.

3. A New Algorithm for Minimizing the Maximum Lateness

In this section we present the Preemptive Minimize Maximum Lateness algorithm. Phase one of our algorithm applies Cole's pipeline merge technique to phase one of Dekel and Sahni's parallel scheduling algorithm, maintaining partial sums of processing times to estimate which jobs are preempted. Phase two uses the information calculated from phase one to design an algorithm composed of merges to which we can apply Cole's parallel merge technique. The merges in phase two are more complicated than the merges in phase one, as phase two merges do not follow the binary tree structure and some lists may merge with several lists at the same time.

Section 3.1 describes the merge operation, merging two sorted lists while estimating preempted jobs at each intermediate step. Section 3.2 describes phase one of the algorithm and section 3.3 describes phase two.

3.1. The Merge with Partial Sums

In this section we show how to incorporate the partial sums into the merge operation.

The most common operation throughout the algorithm is the merging of two sorted sample lists of jobs. While the merge is the same merge as that in Cole's sorting algorithm, we are in addition calculating estimated partial sums of processing times for each job. The estimated partial sums are used to estimate which jobs are preempted and then split a sorted list at that job. This section concentrates on the estimated partial sums. The next section uses the estimated partial sums to calculate preempted jobs.

We estimate partial sums of processing times for jobs in an ordered list at node V because the sample lists we merge together do not contain all the jobs that will be in the final list at a node until the node becomes external. Let e_j be the estimated partial sum of processing times for job j, where j is in an ordered list. Suppose S is a complete sorted list of jobs at a node, i.e. the node is external, then e_j for $j \in S$ is defined as

$$e_j = \sum_{i=1}^{j} p_i$$

Here e_j is not an estimate, but the exact value. When a list of jobs is complete, we can easily calculate the partial sums of processing times. But in our algorithm, we merge together larger and larger samples of jobs using a pipeline, with the lists doubling in size from one stage to the next, until a list is complete. At intermediate stages, we estimate the partial sums of processing times using the e_j of the jobs in the sample lists.

Let SAMP(X) = $\{x_1, x_2, \ldots x_k\}$ be a sample list of jobs, and SAMP(Y) = $\{y_1, y_2, \ldots y_k\}$, be another sample list of jobs, with both lists sorted by deadlines. We show how to merge the two sorted sample lists into list Z ($Z = SAMP(X) \cup SAMP(Y)$). For each job $y_i \in$ SAMP(Y), calculate its rank in the list SAMP(X). Suppose y_i's rank in SAMP(X) is l, i.e. y_i's deadline falls between the deadlines of x_l and x_{l+1}. Then y_i's position in list Z is $i + l$. Estimate the partial sum by using the estimated partial sums in the SAMP(X) and SAMP(Y) lists, e_{i+l} of z_{i+l} is equal to e_l of x_l plus e_i of y_i. Similarly, each job in SAMP(X) can calculate its rank in SAMP(Y) and thus its position in Z. We show in Lemma 4 that two sample lists of size at most k each can be merged together in $O(1)$ time using $O(k)$ processors.

3.2. Phase One

Phase one of our algorithm calculates the set of jobs that can be scheduled, but does not produce the jobs in their scheduled order. This problem is divided into $\log n$ levels of subproblems. Each subproblem has a set of jobs and a time interval in which to schedule the jobs. For each subproblem, phase one can quickly calculate the set of jobs that can be scheduled to run in its associated interval, because the jobs are in sorted order by deadlines. Solutions are calculated by merging and splitting ordered lists. Furthermore, the algorithm is speeded up by pipelining the jobs through the subproblems, calculating the solutions to all subproblems at the same time. Phase two uses the solutions of the subproblems in phase one to calculate the schedule of the jobs.

Phase one begins by sorting in parallel the n jobs by release times, using Cole's [Col86] parallel merge sort. For those jobs with identical release times, it further sorts these jobs by deadlines. We assign the jobs in their order to the leaves of a binary tree, assigning *one* job per leaf. One processor is assigned to each job and it remains with the job as it is pipelined to the root. An interval is associated with each node in the tree, as described in section 2.2. We calculate the $SCHED_V$ and REM_V sets as defined in section 2.2, but we calculate them using a pipeline. For each job i, $e_i = p_i$. If V is a leaf node with job i and interval $[r_i, r_{i+1})$, then job i is placed into $SCHED_V$ if $r_{i+1} - r_i \geq e_i$, or it is placed into REM_V if $r_{i+1} - r_i = 0$. Otherwise job i is split into two jobs, i_1 and i_2, such that i_1 has $p_{i_1} = r_{i+1} - r_i$ and i_2 has $p_{i_2} = p_i - p_{i_1}$. Then i_1 is placed into $SCHED_V$ and i_2 is placed into REM_V.

The algorithm runs in stages. At each stage, operations are performed at each node V in the tree. Let $SCHED_V(s)$ be the current solution set of jobs at node V in stage s. SAMP($SCHED_V(s)$) will be the sample list of $SCHED_V$ at stage s. If V is not external, SAMP($SCHED_V(s)$) equals every fourth job of $SCHED_V(s)$. If V becomes external at stage s, then SAMP($SCHED_V(s)$) equals every fourth job of $SCHED_V(s)$, SAMP($SCHED_V(s+1)$) equals every second job of $SCHED_V(s)$ and SAMP($SCHED_V(s+2)$) $= SCHED_V(s)$. Note that there is no need to calculate SAMP($SCHED_V(z)$) for $z > s+2$ as the parent of V now has its complete list. When $SCHED_V(s)$ becomes external at stage s, it is equivalent to the list $SCHED_V$ in Dekel

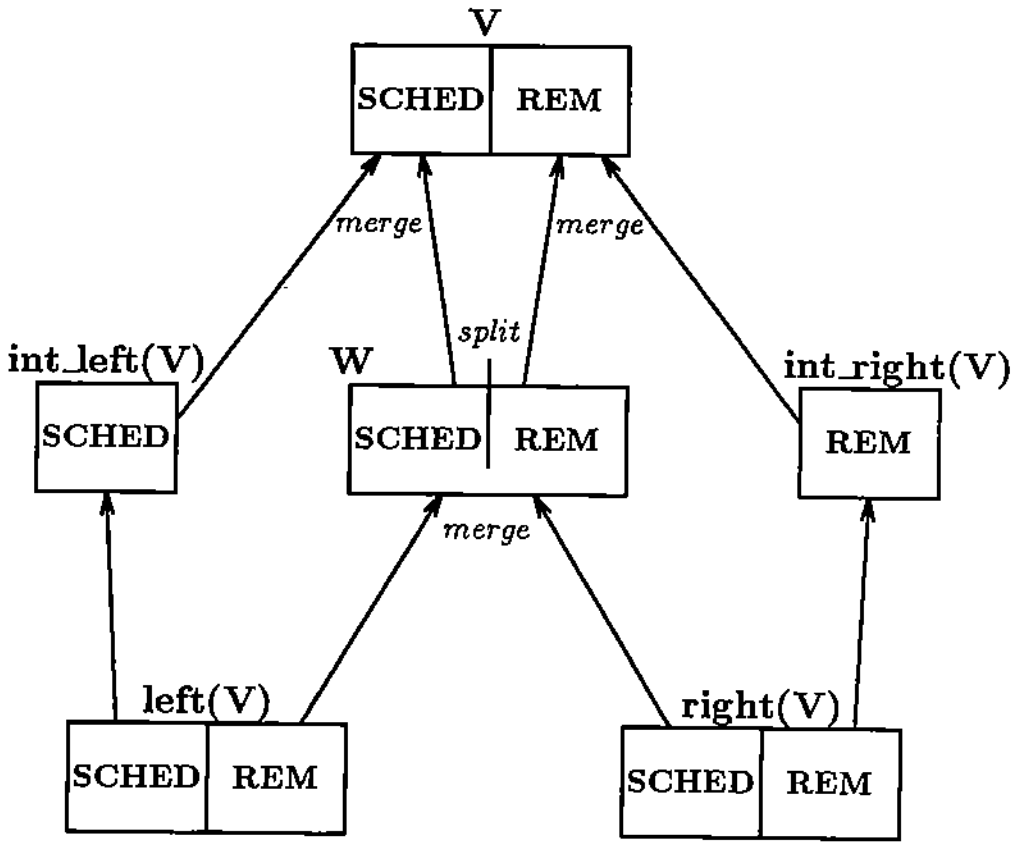


FIGURE 1

and Sahni's algorithm.

To handle the splitting of lists, we create an *intermediate* level between every level in the tree. Levels from the original tree will be called *regular* levels. Let V be a node in the original tree with left child $left(V)$ and right child $right(V)$. On the intermediate level between V and its children, there will be three nodes W, $int\_left(V)$ and $int\_right(V)$. $W(s)$ will be estimated by samples sent up from $REM_{left(V)}(s-1)$ and $SCHED_{right(V)}(s-1)$ and then $W(s)$ will be split into $SCHED_W(s)$ and $REM_W(s)$. $Int\_left(V)$ will receive sample lists from $SCHED_{left(V)}(s-1)$ and $int\_right(V)$ will receive sample lists from $REM_{right(V)}(s-1)$. Finally, $SCHED_V(s)$ and $REM_V(s)$ can be calculated. Figure 1 shows the flow between two regular levels and the intermediate level between them. Note that $int\_left(V)$ and $int\_right(V)$ are not necessary. Instead of these two nodes, smaller samples of $SCHED_{left(V)}$ and $REM_{right(V)}$

could be defined, and SAMP($SCHED_{left(V)}(s-1)$) would be sent to $SCHED_V(s)$ and SAMP($REM_{right(V)}(s-1)$) would be sent to $REM_V(s)$. We use $int\_left(V)$ and $int\_right(V)$ so that all the sample lists are defined the same, thus making it easier to describe the algorithms and the proofs.

Here is a description of one stage in the algorithm. Let V, $left(V)$, $right(V)$ and W be defined as above. The first step consists of calculating in parallel SAMP($SCHED_V(s-1)$) and SAMP($REM_V(s-1)$) for all nodes V. In the second step, for all nodes of type V, W, $int\_left(V)$ and $int\_right(V)$ calculate in parallel the following (we abbreviate $left$ by l and $right$ by r): $SCHED_V(s) = \text{SAMP}(SCHED_W(s-1)) \cup \text{SAMP}(SCHED_{int\_l(V)}(s-1))$, $SCHED_{int\_l(V)}(s) = \text{SAMP}(SCHED_{l(V)}(s-1))$, $REM_V(s) = \text{SAMP}(REM_W(s-1)) \cup \text{SAMP}(REM_{int\_r(V)}(s-1))$, $REM_{int\_r(V)}(s) = \text{SAMP}(REM_{r(V)}(s-1))$, and $W(s) = \text{SAMP}(REM_{l(V)}(s-1)) \cup \text{SAMP}(SCHED_{r(V)}(s-1))$. The third step consists of splitting $W(s)$ into $SCHED_W(s)$ and $REM_W(s)$ for all W in parallel.

The most difficult part of this problem lies in splitting $W(s)$ while using only $O(n)$ processors. $W(s)$ must be split into $SCHED_W(s)$ and $REM_W(s)$ using estimated partial sums of processing times, so neither $SCHED_W(s)$ nor $REM_W(s)$ is more than double their size from stage $s-1$. We give more detail on the sample lists and then explain how the split is calculated.

As soon as a $SCHED$ or REM list receives its first few jobs, it will approximately double in size at each stage following, becoming external when it has received all its jobs. The jobs start in the leaves of the tree with 0 or 1 jobs in each $SCHED_V$ and REM_V list, for V a leaf node. The jobs are pipelined up the tree level by level in sample lists. Unlike Cole's algorithm, the $SCHED_V$ and REM_V lists can be different sizes throughout the tree. The larger of the two will possibly receive jobs at an earlier stage. Both lists on receiving jobs approximately double in size from one stage to the next, with both lists becoming external (or complete) at the same time. For example, let V be a node in the tree at stage s such that when V becomes external, $|SCHED_V(s)| = 49$ and $|REM_V(s)| = 15$. Assuming no splitting of jobs has occurred, the sizes of these lists at previous stages are: $|SCHED_V(s-1)| = 24$, $|SCHED_V(s-2)| = 11$, $|SCHED_V(s-3)| = 5$, $|SCHED_V(s-4)| = 2$, $|SCHED_V(s-5)| = 0$, $|REM_V(s-1)| = 7$, $|REM_V(s-2)| = 3$,

$|REM_V(s-3)| = 1$, and $|REM_V(s-4)| = 0$.

When $W(s)$ is external, splitting $W(s)$ into $SCHED_W(s)$ and $REM_W(s)$ is easy. Let $[r_i, r_k)$ be the interval associated with $W(s)$, then $SCHED_W(s)$ can choose jobs from $W(s)$ whose partial sum of processing times is at most $r_k - r_i$. When $W(s)$ is external we can calculate which job should be preempted and then split that job into two jobs. All jobs $j \in W(s)$ with $e_j \leq r_k - r_i$ are placed into $SCHED_W(s)$. If there is no job j with $e_j = r_k - r_i$, then let x be the job with the smallest deadline that has $e_x > r_k - r_i$. Then x is split into two jobs, x_1 and x_2, so that

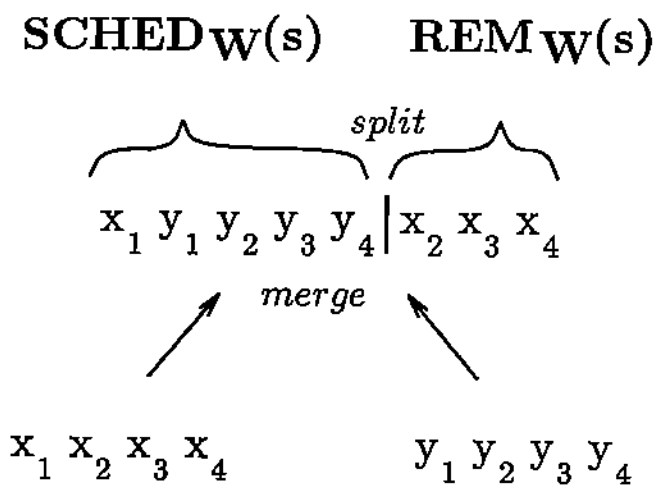
$$\sum_{j \in SCHED_W(s) \cup x_1} p_j \;=\; r_k - r_i$$

When we split job x, we split its processing time. Then $p_{x_1} = r_k - r_i - \sum p_j$ for all $j \in SCHED_W(s) - x_1$, $e_{x_1} = r_k - r_i$, and $p_{x_2} = p_x - p_{x_1}$. We must update the partial sums of all the jobs in $REM_W(s)$ since their partial sums in $W(s)$ included those jobs that are in $SCHED_W(s)$. Let j_L be the last job in $SCHED_W(s)$. Then subtract e_{j_L} from e_j for all $j \in REM_W(s)$. Note that at most one job is split at each leaf node and each W node. Thus, there are at most $3n$ jobs at the root at the end of phase 1.

Since $W(s)$ does not have all its jobs until it becomes external, the partial sums for jobs in $W(s)$ are estimated for all stages s such that $W(s)$ is not external. The split of $W(s)$ at each stage will be an estimate. We want to estimate the split of $W(s)$ so that $|SCHED_W(s)|$ and $|REM_W(s)|$ are at most double in size from one stage to the next. One attempt at estimating the split is to split $W(s)$ at the job with the estimated partial sum of processing times that is closest to $r_k - r_i$. All jobs $j \in W(s)$ with $e_j < r_k - r_i$ are placed into $SCHED_W(s)$ and all remaining jobs are placed into $REM_W(s)$. Since the partial sums of processing times are just estimates we don't need an exact fit into the interval until a node is external. Job splitting will occur only when nodes are external.

One problem with this estimate for the split is that we cannot guarantee that $|SCHED_W(s)|$ and $|REM_W(s)|$ will at most double in size from stage $s-1$ to stage s. For example, let $w_1, w_2, \ldots w_l$ be the jobs in $W(s)$. Suppose the split is between w_m and w_{m+1}. It is possible that many jobs in $W(s)$ from one of the lists SAMP($SCHED_{right(V)}$) or SAMP($REM_{left(V)}$) are bunched together, i.e. $w_{m-p}, w_{m-(p-1)}, \ldots w_m$ are all from

a)



b)

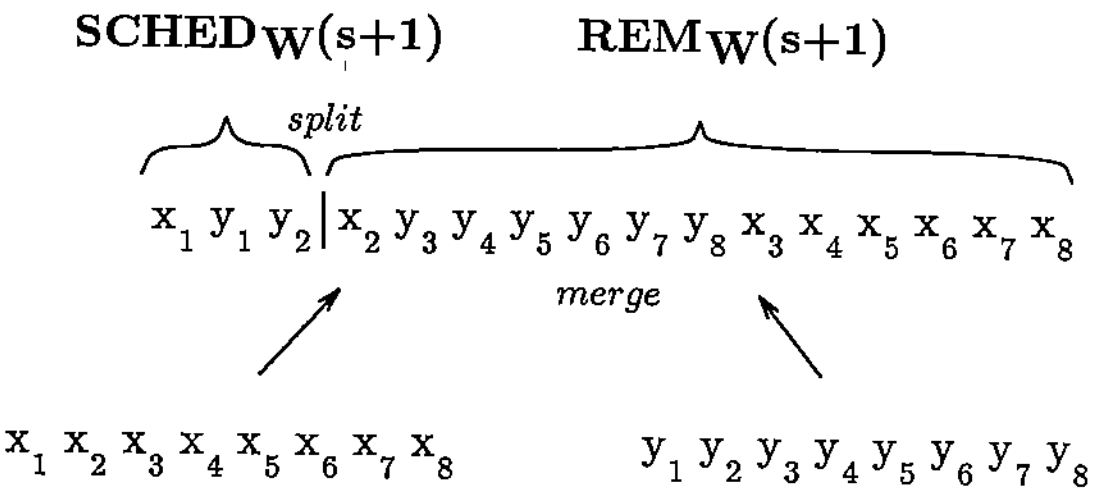
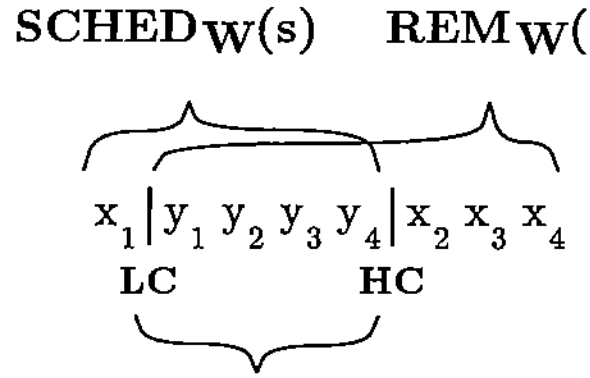


FIGURE 2

SAMP($SCHED_{right(V)}(s-1)$). Then $w_{m-(p+1)}$ and w_{m+1} are adjacent jobs from SAMP($REM_{left(V)}(s-1)$). If $w_{m-(p+1)}$ is small in comparison to w_{m+1}, then at stage $s+1$, the split position of $W(s+1)$ could be shifted to the left so that $|REM_W(s+1)| >> 2 * |REM_W(s)|$.

Figure 2 illustrates why we cannot estimate the split of $W(s)$ by splitting it at the job with the estimated partial sum of processing times that is closest to the size of W's interval. The x_i's and y_i's in figure 2 are the sample lists of jobs that are merged together to form

a)



b)

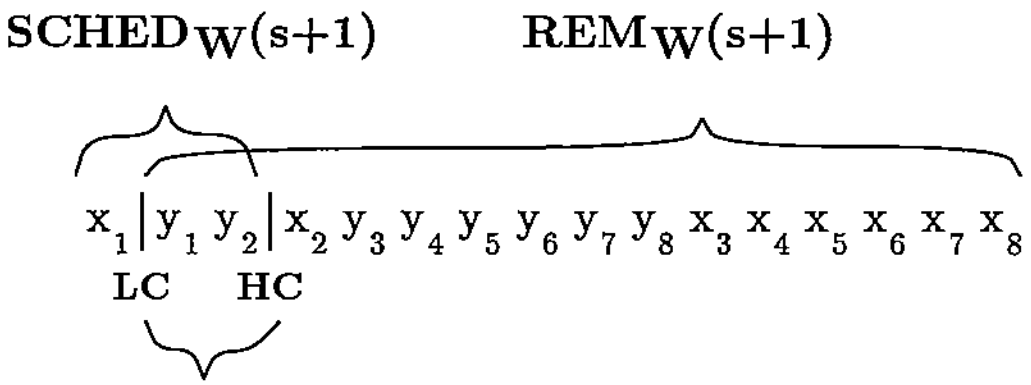


FIGURE 3

W, where x_i and y_i represent the ith element in their corresponding lists. Figure 2a shows the merge at stage s, and figure 2b shows the merge one stage later. The split in $W(s+1)$ is shifted far to the left of the split in $W(s)$ causing $|REM_W(s+1)| >> 2 * |REM_W(s)|$. To perform merges quickly, REM_W cannot be more than double in size from one stage to the next.

Now, we show how to split $W(s)$. Instead of estimating the split in just one position, estimate the split by a range of values. Make two cuts in $W(s)$, a high cut HC and a low cut LC. Again, let w_m be the job with the largest deadline that has $e_{w_m} < r_k - r_i$. Without loss of generality, w_m is from SAMP($SCHED_{right(V)}(s-1)$). Let w_j be the first job to

w_m's left that came from the other list SAMP($REM_{left(V)}(s-1)$). Similarly let w_h be the first job to w_m's right that came from the other list SAMP($REM_{left(V)}(s-1)$). Then the jobs $w_{j+1}, \ldots, w_m, \ldots, w_{h-1}$ all came from the same list SAMP($SCHED_{right(V)}(s-1)$) and they define the *region of uncertainty*. The exact split falls somewhere in this region. The low cut LC will be between jobs w_j and w_{j+1} and the high cut HC will be between jobs w_{h-1} and w_h. All jobs in the region of uncertainty will be duplicated and placed into both $SCHED_W(s)$ and $REM_W(s)$. In later stages as the interval of uncertainty shrinks, the LC and HC approach each other and will converge into one cut by the time this node becomes external. So $SCHED_W(s) = w_1, w_2, \ldots w_{h-1}$ and $REM_W(s) = w_{j+1}, \ldots w_l$. Figure 3 shows the region of uncertainty and the HC and LC cuts in $W(s)$ and $W(s+1)$ for the examples in figure 2. Here $|REM_W(s+1)| \leq 2 * |REM_W(s)| + 4$. By duplicating jobs, we are increasing the number of jobs being processed at any stage, so we show in lemma 6 that there are still $O(n)$ jobs being processed at any time.

Lemma 1: Phase one completes after $6 \log n$ stages.

Proof: Whenever a node becomes external, its parent becomes external three stages later. With the addition of the intermediate levels in the tree, there are now $2 \log n$ levels in the tree. Thus, phase one completes after $6 \log n$ stages. □.

Lemma 2: For each list of jobs L, $L = SCHED_V, REM_V, W, SCHED_W, REM_W$, etc., $|L(s)| \leq 2|L(s-1)| + 4$.

Proof: The proof is by induction on stage s. Basis ($s = 0$): This is trivial.

Induction Step: Assume that $|L(s)| \leq 2|L(s-1)| + 4$ is true for stage s. Show $|L(s+1)| \leq 2|L(s)| + 4$ is true at stage $s+1$.

$$|W(s+1)| = \left\lfloor \frac{|REM_{left(V)}(s)|}{4} \right\rfloor + \left\lfloor \frac{|SCHED_{right(V)}(s)|}{4} \right\rfloor$$

$$\leq \left\lfloor \frac{2 * |REM_{left(V)}(s-1)| + 4}{4} \right\rfloor + \left\lfloor \frac{2 * |SCHED_{right(V)}(s-1)| + 4}{4} \right\rfloor$$

$$\leq 2\left(\left\lfloor \frac{|REM_{left(V)}(s-1)|}{4} \right\rfloor + \left\lfloor \frac{|SCHED_{right(V)}(s-1)|}{4} \right\rfloor \right) + 4$$

$$= 2 * |W(s)| + 4$$

Thus $|W(s+1)| \leq 2*|W(s)|+4$. $SCHED_W(s+1)$ and $REM_W(s+1)$ are determined by the LC and HC cuts. The position of HC in $W(s+1)$ is less than or equal to twice the position of HC in $W(s)+4$. The position of LC in $W(s+1)$ is greater than or equal to twice the position of LC in $W(s)-4$. So $|SCHED_W(s+1)| \leq 2 * |SCHED_W(s)| + 4$ and $|REM_W(s+1)| \leq 2 * |REM_W(s)| + 4$. As these cuts approach each other, the lists $SCHED_W$ and REM_W are less than double in size from one stage to the next. It is easy to show that this lemma also holds for $SCHED_V(s+1)$, $REM_V(s+1)$, $SCHED_{int\_left(V)}(s+1)$ and $REM_{int\_right(V)}(s+1)$. So, for all lists L, $|L(s+1)| \leq 2 * |L(s)|$. □.

Lemma 3: For each list L, $L = SCHED_V, REM_V, W, SCHED_W, REM_W$, etc., SAMP($L(s)$) is a 3-cover of SAMP($L(s+1)$).

Proof: For the general merge problem, [ACG87] shows that if $[a,b]$ intersects $k+1$ elements in SAMP($L(s)$), then it intersects at most $8k+8$ elements in $L(s)$ for all $k \geq 1$ and $s \geq 1$. A similar proof using the size restrictions in Lemma 2 shows that if $[a,b]$ intersects $k+1$ elements in SAMP($W(s)$), then it intersects at most $8k+8$ elements in $W(s)$ for all $k \geq 1$ and $s \geq 1$. Thus, SAMP($W(s)$) is a 3-cover of SAMP($W(s+1)$), SAMP($SCHED_W(s)$) is a 3-cover of SAMP($SCHED_W(s+1)$), and SAMP($REM_W(s)$) is a 3-cover of SAMP($REM_W(s+1)$). Clearly, SAMP($SCHED_{int\_left(V)}(s)$) is a 3-cover of SAMP($SCHED_{int\_left(V)}(s+1)$) and SAMP($REM_{int\_right(V)}(s)$) is a 3-cover of SAMP($REM_{int\_right(V)}(s+1)$). Thus, SAMP($SCHED_V(s)$) is a 3-cover of

SAMP($SCHED_V(s+1)$) and SAMP($REM_V(s)$) is a 3-cover of SAMP($REM_V(s+1)$). □.

Lemma 4: Given two sample lists of jobs of size at most k each, they can be merged together in $O(1)$ time using $O(k)$ processors.

Proof: There are three merges: $SCHED_V(s)$ is the merge of SAMP($SCHED_W(s-1)$) and SAMP($SCHED_{int\_l(V)}(s-1)$), $REM_V(s)$ is the merge of SAMP($REM_W(s-1)$) and SAMP($REM_{int\_r(V)}(s-1)$), and $W(s)$ is the merge of SAMP($REM_{l(V)}(s-1)$) and SAMP($SCHED_{r(V)}(s-1)$). Let $L(s)$ be the current list of jobs for L at time s, $L(s+1)$ be the list of jobs at L at time $s+1$, and $Z(s+1) = \text{SAMP}(X(s)) \cup \text{SAMP}(Y(s))$ for lists X, Y, and Z. From Lemma 3, $L(s)$ is a c-cover of $L(s+1)$ for $L = Z$, SAMP(X), and SAMP(Y) and c a constant. Thus, we show that each job $y_i \in \text{SAMP}(Y(s))$ can calculate its rank in SAMP($X(s)$) in $O(1)$ time.

$Z(s)$ is a c-cover of $Z(s+1)$, so $Z(s)$ is a c-cover of SAMP($Y(s)$) and SAMP($X(s)$). For each element $y_i \in \text{SAMP}(Y(s))$ we calculate its rank in $Z(s)$ by assigning one processor to each job in $Z(s)$. Between any two adjacent jobs f and h in $Z(s)$, there are at most c jobs in SAMP($Y(s)$), so each processor in $Z(s)$ whose job f covers different jobs in SAMP($Y(s)$) than job h covers writes its rank to those c jobs in SAMP($Y(s)$). Suppose y_i's rank in $Z(s)$ is e, i.e. y_i is between elements e and g, $e, g \in Z(s)$. Find e and g's ranks in SAMP($X(s)$), say e' and g'. There will be at most c jobs between e' and g' in SAMP($X(s)$). So comparing y_i to these jobs we can find y_i's rank in SAMP($X(s)$) in $O(1)$ time using only $O(k)$ processors. □.

Lemma 5: Phase one takes $O(\log n)$ time.

Proof: There are at most $6 \log n$ stages. Each stage consists of merging sample lists and splitting W's, each of which are performed in $O(1)$ time. □.

Lemma 6: There are $O(n)$ processors used at each stage in phase one.

Proof: We count the number of jobs that exist at stage s, since one processor is assigned to each job. There may be more than n jobs at stage s due to job splitting and

job duplication, but not more than $O(n)$ jobs. When a level becomes external, there are no jobs on the levels below it, so we will focus on an external level and count the number of jobs on this level and on the levels above it. There are six cases to consider since there are two types of levels, regular and intermediate, and a different level becomes external every three stages. We give the proof for the case when level x is a regular level and it just became external. The other cases have similar proofs.

Suppose level x just became external at stage s, where x is a regular level. Level x just became external so level x contains at most $3n$ jobs. To count the number of jobs on some level y at stage s, we need to know the most recent level that was external whose jobs are now at level y. For example, counting the number of jobs on levels x, $x+1$, and $x+2$ at stage s depends on level $x-1$. Level $x-1$ became external at stage $s-3$. This level had at most $3n$ jobs at that time. At stage $s-2$, level $x-1$ sends at most $3n/4$ jobs to level x. At stage $s-1$, level x sends at most $3n/4^2$ jobs to level $x+1$. Level $x+1$ is an intermediate level. Splitting the W's on this level can increase the number of jobs on this level. The worst case split is when $|W| = k$, the first $k/2$ elements are from REM_Q and the remaining $k/2$ are from $SCHED_T$. When splitting W, the low cut LC will be at position 0 and the high cut HC will be at position $k/2$. $|SCHED_W| = k/2$ and $|REM_W| = k$, an $3/2$ increase in the number of jobs. Splitting W's can result in at most $(3/2) * (3n/4^2)$ jobs at level $x+1$ at stage $s-1$. At stage s, level $x+1$ sends at most $(3^2n)/(4^3 * 2)$ jobs to level $x+2$. Thus at stage s, levels x, $x+1$ and $x+2$ depend on level $x-1$ to calculate the maximum number of jobs on these levels. Level x contains at most $3n$ jobs, level $x+1$ contains at most $(3^2n)/(4 * 2^2)$ jobs and level $x+2$ contains at most $(3^2n)/(4^3 * 2)$ jobs. In general at stage s, levels $x+3(i-1)$, $x+3(i-1)+1$, $x+3(i-1)+2$ depend on level $x-i$ for calculating the number of jobs on these levels.

The number of jobs at stage s equals the sum of the jobs on level x and all levels above level x. So the number of jobs at stage s

$$\leq 3n + \frac{3^2 n}{4 * 2^2} + \frac{3^2 n}{4^3 * 2} + \frac{3^4 n}{4^4 * 2^3} + \frac{3^4 n}{4^5 * 2^4} + \frac{3^5 n}{4^7 * 2^4} + \frac{3^5 n}{4^8 * 2^4} + \frac{3^6 n}{4^9 * 2^6} + \frac{3^6 n}{4^{11} * 2^5} + \ldots$$

$$= 3n + \sum_{i=1}^{\infty} \frac{3^{(4i-2)}}{4^{(8i-7)} 2^{(4i-2)}} + \sum_{i=1}^{\infty} \frac{3^{(4i-2)}}{4^{(8i-5)} 2^{(4i-3)}} + \sum_{i=1}^{\infty} \frac{3^{(4i)}}{4^{(8i-4)} 2^{(4i-1)}} + \sum_{i=1}^{\infty} \frac{3^{(4i)}}{4^{(8i-3)} 2^{(4i)}}$$
$$+ \sum_{i=1}^{\infty} \frac{3^{(4i+1)}}{4^{(8i-1)} 2^{(4i)}} + \sum_{i=1}^{\infty} \frac{3^{(4i+1)}}{4^{(8i)} 2^{(4i)}}$$

$$\leq 3n + \frac{3^2 n}{4^2} + \frac{3^3 n}{4^3} + \frac{3^4 n}{4^4} + \frac{3^5 n}{4^5} + \frac{3^6 n}{4^6} + \frac{3^7 n}{4^7} + \frac{3^8 n}{4^8} + \frac{3^9 n}{4^9} + \ldots$$

$$= 3n + \left(\frac{3}{4}\right) n * \sum_{i=1}^{\infty} \left(\frac{3}{4}\right)^i = O(n)$$

□.

Theorem 1: Phase one of our algorithm that preemptively schedules jobs to run on one machine while minimizing the maximum lateness of the jobs, runs in $O(\log n)$ time using $O(n)$ processors.

3.3. Phase Two

Phase two of our algorithm calculates the schedule of jobs, using results from phase one. In phase one, each node in the tree is given the set of jobs *in its subtree* and a time interval. Using these, it calculates the solution set of jobs that can be scheduled in its associated interval. Phase two updates the solution sets at nodes by considering *all* the jobs in the tree when finding the schedule for a node's interval. Now, the nodes have the exact set of jobs to schedule in their associated intervals, but the jobs are still not in scheduled order. However, the schedule that minimizes the maximum lateness can be constructed by concatenating the NEW_$SCHED_V$ lists in the leaves of the tree.

Our phase two uses the information calculated from phase one to avoid extractions. In phase two of Dekel and Sahni's algorithm, they perform merges and extractions at

each node on a level. The parallel merging technique of Cole's does not apply to extracting jobs from a set. To calculate NEW_$SCHED_{left(V)}$, Dekel and Sahni extract from NEW_$SCHED_V$ those jobs that were not considered in $SCHED_{left(V)}$ in phase one. Using these extracted jobs and $SCHED_{left(V)}$, they calculate NEW_$SCHED_{left(V)}$. The extractions are not necessary. All the jobs that are extracted from NEW_$SCHED_V$ appear in REM_U from phase 1, where U is the left sibling of $left(V)$'s closest ancestor that is a right child. The leftmost jobs in R_U can be merged together with $SCHED_{left(V)}$ and used to calculate NEW_$SCHED_{left(V)}$.

Eliminating extractions, our phase two consist of merges and splits of lists. To calculate new solution sets, we can bypass all right nodes (except right leaf nodes which are a special case) and just work with the left nodes, calculating their new solution sets using merges and splits. We discuss the beginning of phase two, give the algorithm for a general node and then show how to apply Cole's parallel merge to our algorithm.

$SCHED$ and REM sets were calculated in phase one. In phase two, we calculate NEW_$SCHED$ sets. Phase two begins as follows. Let V be the root of the tree. $SCHED_V$ contains all the jobs sorted by deadlines, the solution for its interval. So, NEW_$SCHED_V$ = $SCHED_V$. Let $left(V)$ be the left child of V and $right(V)$ be the right child of V. $SCHED_{left(V)}$ contains the solution for $left(V)$'s interval when considering all the jobs in $left(V)$'s subtree. Since all the jobs in $right(V)$'s subtree have larger release times than the jobs in $left(V)$'s subtree, NEW_$SCHED_{left(V)} = SCHED_{left(V)}$. $REM_{left(V)}$ contains all the jobs in $left(V)$'s subtree that did not fit into $left(V)$'s interval. These jobs have release times less than any job in $right(V)$'s subtree, so they should be considered in the new solution sets in $right(V)$.

We do not need to calculate NEW_$SCHED_{right(V)}$ to update the solution sets in $right(V)$'s subtree. Suppose we merge together $REM_{left(V)}$ and $SCHED_{right(V)}$ forming NEW_$SCHED_{right(V)}$. The new information in NEW_$SCHED_{right(V)}$ would then be passed down through its subtree, but we would have to extract the jobs from $REM_{left(V)}$ that are in NEW_$SCHED_{right(V)}$. Let $left(right(V))$ be $right(V)$'s left child and $right(right(V))$ be $right(V)$'s right child. NEW_$SCHED_{right(V)}$ must be divided into new solutions for NEW_$SCHED_{left(right(V))}$ and NEW_$SCHED_{right(right(V))}$.

NEW_$SCHED_{left(right(V))}$ needs to consider those jobs in NEW_$SCHED_{right(V)}$ that were not in $SCHED_{right(V)}$, which are those jobs whose release times are less than the interval at $right(V)$, which are just those jobs from $REM_{left(V)}$. So by merging $SCHED_{left(right(V))}$ and $REM_{left(V)}$ together, NEW_$SCHED_{left(right(V))}$ can be calculated by chosing the jobs with minimum deadlines whose sum of processing times fits into $left(right(V))$'s interval. This will consist of one merge, $SCHED_{left(right(V))} \cup REM_{left(V)}$, and then one split, splitting the list so that $\sum p_i = left(right(V))$'s interval, for all $i \in$ NEW_$SCHED_{left(right(V))}$. . Notice that we did not need to calculate NEW_$SCHED_{right(V)}$ to calculate NEW_$SCHED_{left(right(V))}$, we can bypass NEW_$SCHED_{right(V)}$.

$Left(right(V))$'s children consider some of the jobs from $REM_{left(V)}$ in their NEW_$SCHED$ sets. Let $left(left(right(V)))$ be the left child of $left(right(V))$ and $right(left(right(V)))$ be the right child of $left(right(V))$. (From now on we will abbreviate *left* by l and *right* by r.) NEW_$SCHED_{l(r(V))}$ should be divided into new solutions for NEW_$SCHED_{l(l(r(V)))}$ and NEW_$SCHED_{r(l(r(V)))}$. NEW_$SCHED_{l(l(r(V)))}$ is formed by considering jobs in $SCHED_{l(l(r(V)))}$ and those jobs in NEW_$SCHED_{l(r(V))}$ that have a release time less than $l(r(V))$'s interval, which are some of the jobs from $REM_{l(v)}$. When $REM_{l(V)}$ and $SCHED_{l(r(V))}$ were merged together and then split, $REM_{l(V)}$ is really split in half with those jobs in the left half, call these $REM^{(1)}_{l(V)}$, going into NEW_$SCHED_{l(r(V))}$. $REM^{(1)}_{l(V)}$ is the set of jobs that have release time less than $l(l(r(V)))$'s interval and which we want to consider for NEW_$SCHED_{l(l(r(V)))}$. Thus to form NEW_$SCHED_{l(l(r(V)))}$, merge together $SCHED_{l(l(r(V)))}$ and $REM^{(1)}_{l(V)}$, and then split the list to fit into $l(l(r(V)))$'s interval. We do not need to extract any jobs from NEW_$SCHED_{l(r(V))}$.

The children of $R(r(V))$ consider some of the jobs from $REM_{l(V)}$ and $REM_{l(r(V))}$ in their NEW_$SCHED$ sets. Let $l(r(r(V)))$ be the left child of $r(r(V))$. We show how to calculate NEW_$SCHED_{l(r(r(V)))}$ by bypassing $r(r(V))$ and $r(V)$. NEW_$SCHED_{r(r(V))}$ is equal to those jobs in NEW_$SCHED_{r(V)}$ that are not in NEW_$SCHED_{l(r(V))}$. When NEW_$SCHED_{l(r(V))}$ was formed by merging together $REM_{l(V)}$ and $SCHED_{l(r(V))}$ and then splitting the result, those jobs in the right half of the split were not chosen. So

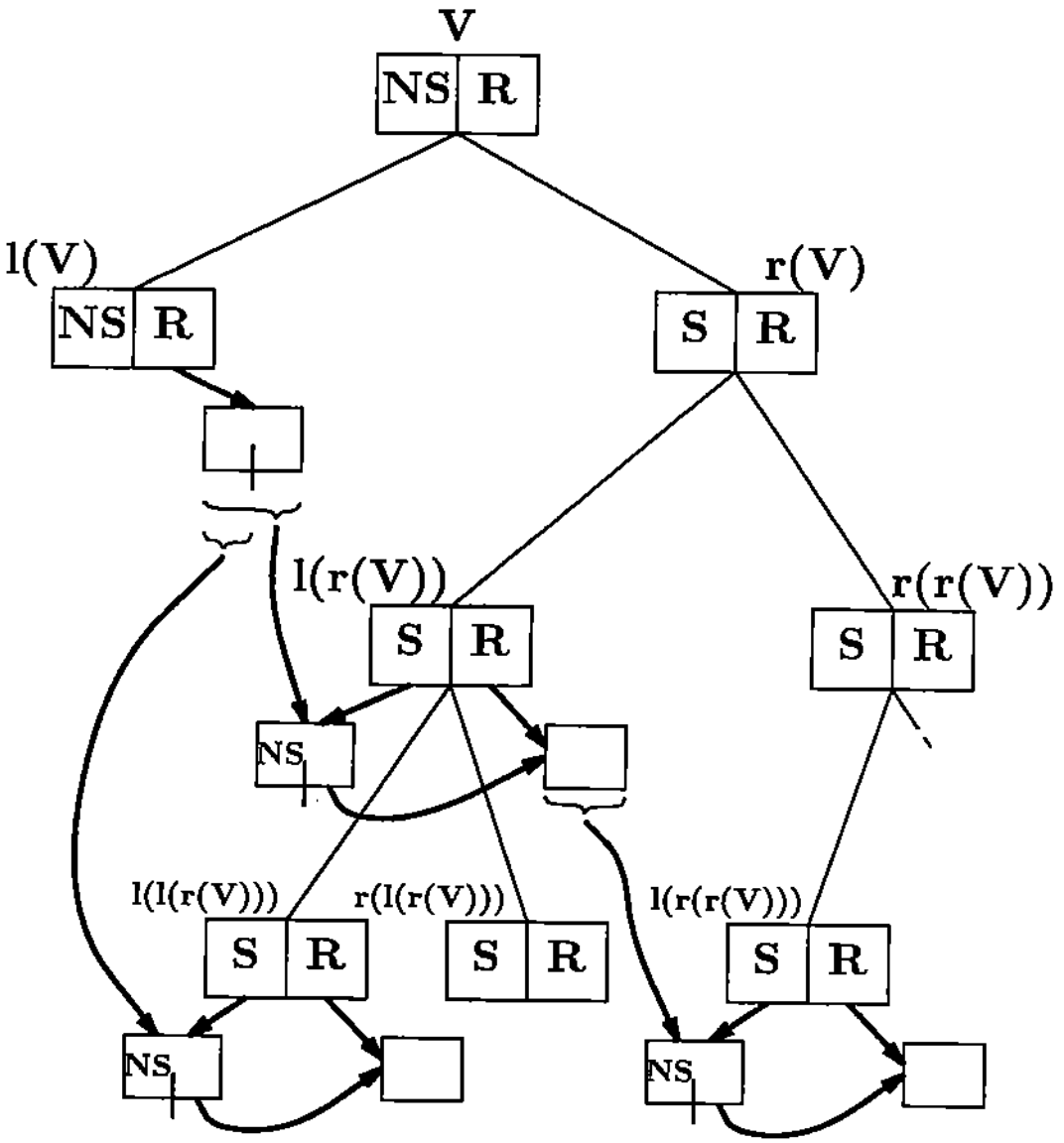


FIGURE 4

those jobs not chosen are merged together with the jobs in $REM_{l(r(V))}$ to form the set of jobs not chosen at $l(r(V))$ but which should be considered in $r(r(V))$'s subtree, call this $REM^{(0)}_{l(r(V))}$. All the jobs in $REM^{(0)}_{l(r(V))}$ would appear in NEW_$SCHED_{r(r(V))}$ if we calculated it. Instead, we can just bypass $r(r(V))$ and calculate NEW_$SCHED_{l(r(r(V)))}$ by merging together $REM^{(0)}_{l(r(V))}$ and $SCHED_{l(r(r(V)))}$ and then splitting this list. Figure 4 shows the flow of jobs in phase two starting with $left(V)$. The abbreviations used in

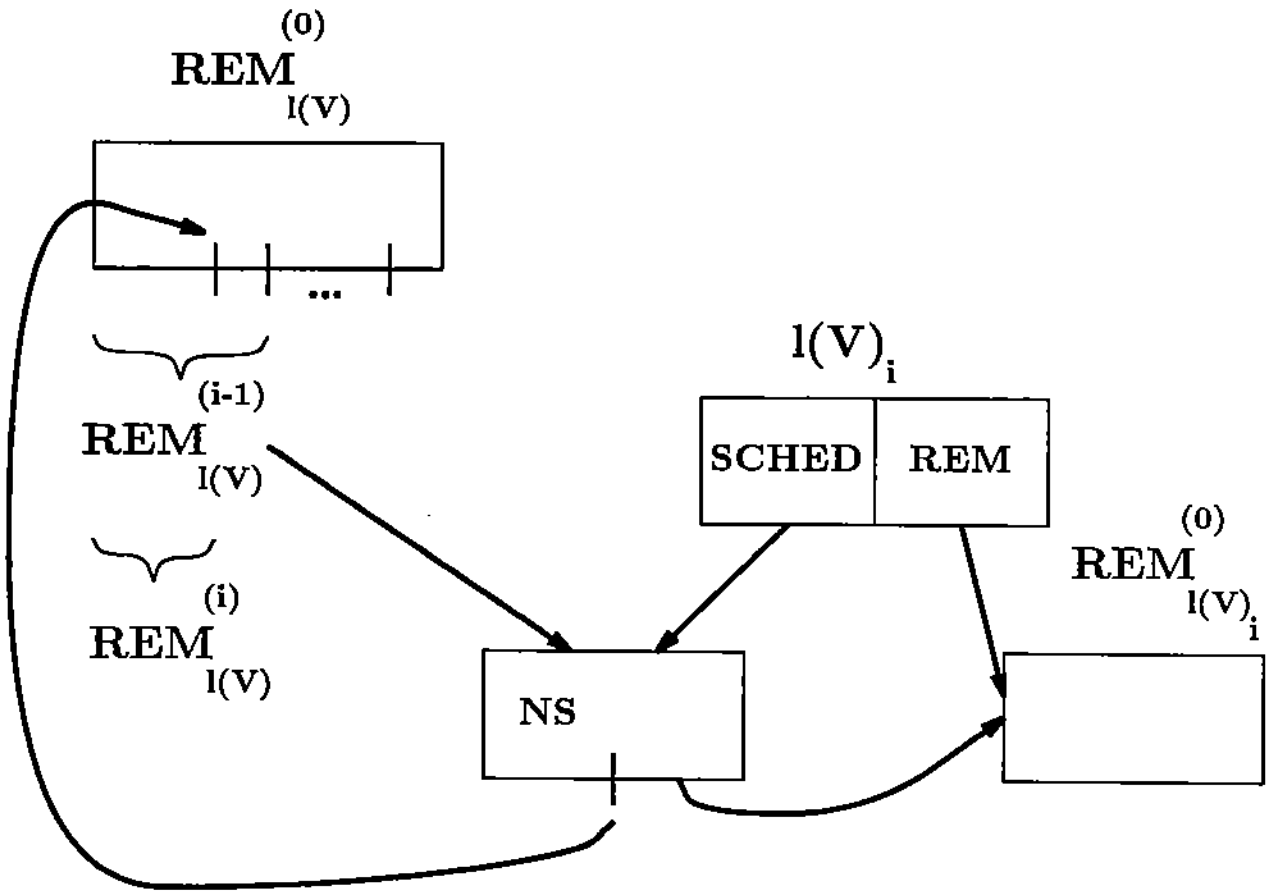


FIGURE 5

this figure are S for $SCHED$, R for REM, and NS for NEW_$SCHED$.

Here is the algorithm for a general node in phase two. Let $left(V)$ be a node in the tree with interval $[r_i, r_j)$ and lists $SCHED_{left(V)}$ and $REM_{left(V)}$ calculated from phase one. Assume that jobs in the tree to the left of $left(V)$'s subtree that have not been selected yet are merged together with $SCHED_{left(V)}$ and then split, the left half forming NEW_$SCHED_{left(V)}$ and the right half merging with $REM_{left(V)}$ to form $REM^{(0)}_{left(V)}$. Let $right(V)$ be the right sibling of $left(V)$. $Right(V)$ is a right node so we will ignore it, but note that the set of jobs in $REM^{(0)}_{left(v)}$ should be considered in $right(V)$'s subtree. Let $left(V)_1$ be $right(V)$'s left child, $left(V)_2$ be $left(V)_1$'s left child, $left(V)_3$ be $left(V)_2$'s left child, etc. To form NEW_$SCHED_{left(V)_i}$, merge $SCHED_{left(V)_i}$ and $REM^{(i-1)}_{left(v)}$ to form NEW_$left(V)_i$. Split NEW_$left(V)_i$ into NEW_$SCHED_{left(V)_i}$ and a right half

which is merged with $REM_{left(V)_i}$ to form $REM^{(0)}_{left(V)_i}$. When NEW$_left(V)_i$ is split, use this split to split $REM^{(i-1)}_{left(V)}$ so that $REM^{(i)}_{left(V)} = REM^{(i-1)}_{left(V)} \cap$ NEW$_SCHED_{left(V)_i}$. Figure 5 shows the flow of jobs for a general node $left(V)_i$.

Right leaf nodes are a special case. Suppose $left(V)$ is a left leaf node and $right(V)$ is its sibling right leaf node. $REM^{(0)}_{left(V)}$ is the set of jobs whose release times are less than $right(V)$'s interval and that have not been selected to run. So $SCHED_{right(V)}$ and $REM^{(0)}_{left(V)}$ are merged together and split with the left half of the split forming NEW$_SCHED_{right(V)}$.

That is the general algorithm for phase two. If the algorithm proceeds by levels, performing all the merges on each level before proceeding onward, the algorithm will spend $O(\log \log n)$ time on each level resulting in an $O(\log n \log \log n)$ time algorithm for phase two. Instead, we organize phase two so that we can apply the pipeline merge technique used in phase one by adding intermediate levels to the tree and setting up each list saved from phase one so that it sends samples of lists to be merged, each merge performed in $O(1)$ time.

An intermediate level is added between every two levels in the tree. From the description of the general algorithm, there are two merges and one split for each left node in the original tree. At node $left(V)_i$, $SCHED_{left(V)_i}$ and $REM^{(i-1)}_{left(V)}$ are merged together into NEW$_left(V)_i$ and then split, with the left half of the split forming NEW$_SCHED_{left(V)_i}$. The right half of the split is then merged with $REM_{left(V)_i}$ to form $REM^{(0)}_{left(V)_i}$. So NEW$_left(V)_i$ is formed and split at the level where $left(V)_i$ resides and $REM^{(0)}_{left(V)_i}$ is formed on the intermediate level directly below $left(V)_i$'s level. Similar to phase one, we have alternating levels performing merges and splits on one level and just merges on the next level.

When merging two lists, it is possible that one or both lists are complete at the start of the merge as they were calculated in phase one. To apply Cole's merging scheme, we need small samples of the complete lists which would double in size at each step. Let $SCHED_L$ be a complete list L of size k that was formed in phase one. $SCHED_L(1)$ in phase two is of size 1, the kth element of $SCHED_L$. $SCHED_L(2)$ is of size 2, the $k/2$th and the kth elements of $SCHED_L$. Continuing in this manner, $SCHED_L(j)$ is

of size 2^{j-1} and contains the elements in position $k/2^{j-1}, 2k/2^{j-1}, 3k/2^{j-1}, \ldots k$. The elements in $SCHED_L(j)$ can be picked out of $SCHED_L$ as they are needed. The number of processors needed for $SCHED_L(j)$ to perform merges is $|SCHED_L(j)|$.

At the end of phase one, we save the $SCHED$ and REM lists at every left node and leaf node. Not all the nodes begin calculating the NEW_$SCHED$ lists at the beginning of phase two, instead they must wait until they become active. Level two is the only active level when phase two begins since the root is not needed in any calculations and the intermediate level below the root is not defined. Level two which has one left node $left(V)$ calculates $SCHED_{left(V)}(1)$ and $REM^{(1)}_{left(V)}(1)$ at stage one. Every third stage, the highest nonactive level, either regular or intermediate, becomes active. Setting up phase two this way, the algorithm behaves in a similar manner to the algorithm for phase one.

Lemma 7: Phase two completes after $7 \log n$ stages.

Proof: Phase two begins by merging together $REM^{(0)}_{left(V)}$ and $SCHED_{left(right(V))}$, where $left(V)$ is the left child of the root and $left(right(V))$ is the left child of $left(V)$'s sibling. The two are merged together via samples in $\log n$ stages, becoming external at the $\log nth$ stage. From that point on, every three stages another level becomes external. There are $2 \log n$ levels so the algorithm takes at most $7 \log n$ stages to complete. □.

Lemma 8: Phase two takes $O(\log n)$ time.

Proof: There are at most $7 \log n$ stages. Each stage consists of merging and splitting lists in $O(1)$ time. □.

Lemma 9: There are $O(n)$ processors used at each stage in phase two.

Proof: We count the number of jobs that exist at stage s, since one processor is assigned to each job. Like lemma 6, we prove one of the six cases to consider. Assume that a regular level x just became external at stage s. All the lists in the levels above x became external at earlier stages except for those $REM_V(i)$ lists that are merging with

lists that are below level x. If we sum the sizes of all the lists formed at stage s, which are those lists below and including level x, this will include the jobs in $REM_V(i)$ lists since they form lists at levels below level x.

There will be at most 5n jobs in the leaves at the end of phase two. Phase 1 ended with at most 3n jobs at the root and there will be at most n splits at internal nodes and at most n splits in leaf nodes. We assume there are at most 5n jobs at level x, which has just become external. Counting up the sizes of all lists below level x is similar to the proof of Lemma 6.

$$Number\ of\ jobs \leq 5n + \frac{5*3n}{4*2^2} + \frac{5*3n}{4^3*2} + \frac{5*3^3n}{4^4*2^3} + \ldots$$

$$= 5n + \frac{5}{4}\left(\frac{3n}{2^2} + \frac{3n}{4^2*2} + \frac{3^3n}{4^3*2^3} + \ldots\right)$$

$$\leq 5n + \frac{5}{4}\left(\frac{3n}{4} + \frac{3^2n}{4^2} + \frac{3^3n}{4^3} + \ldots\right)$$

$$= 5n + \frac{5}{4}\left(\sum_{i=1}^{\infty}\left(\frac{3}{4}\right)^i\right) = O(n)$$

□.

Theorem 2: Phase two of our algorithm that preemptively schedules jobs to run on one machine while minimizing the maximum lateness of the jobs, runs in $O(\log n)$ time using $O(n)$ processors.

4. Applications

The technique of pipelining the merges of subproblems to improve an algorithm's running time can be applied to other job scheduling problems. In this section we apply this technique to get faster parallel algorithms for three job scheduling problems. Dekel and Sahni [DeS83a] [DeS83b] show that these problems are in NC and give parallel algorithms

for them that run in $O((\log n)^2)$ time and use $O(n)$ processors. Our algorithms are similar to the algorithm in section 3 and run in $O(\log n)$ time using $O(n)$ processors.

4.1. Problem 1

In the first job scheduling problem we are given n jobs with integer release times, deadlines and unit processing times. We want to minimize the maximum lateness when scheduling the jobs to run on m identical machines. If $m = 1$, the algorithm from section 3 can be used to solve this problem. There is no preemption and thus no job splitting due to the integer release times, so this is a simpler version of the problem in section 3. When $m > 1$, define interval $[r_i, r_j)$ to contain at most $m(r_j - r_i)$ jobs and again apply the algorithm from section 3. In both cases, our algorithm runs in $O(\log n)$ time using $O(n)$ processors. Atallah *et al.* [AGK88] also give a parallel algorithm for this problem that runs in $O(\log n)$ time and uses $O(n)$ processors. This is not necessarily optimal as Frederickson [Fre83] gives a linear sequential algorithm for this problem.

Corollary 1: Defining jobs by integer release times, deadlines and unit processing times, there is an algorithm for the job scheduling problem of minimizing the maximum lateness of jobs that schedules the jobs to run on m machines in $O(\log n)$ time using $O(n)$ processors.

4.2. Problem 2

In the second job scheduling problem we are given n jobs with the same release times ($r_i = 0$), integer deadlines, unit processing times, and a weight. A job is tardy if it completes after its deadline. We want to minimize the sum of the weights of tardy jobs when scheduling the jobs to run on one machine. In this case, sort the jobs by deadlines, define the intervals using the deadlines and place the jobs in the leaves of the binary tree. The algorithm is similar to phase 1 of the algorithm from section 3, at each node choosing the jobs with largest weights to fit into the intervals. At the root r, $SCHED_r$ contains the nontardy jobs and REM_r contains the tardy jobs. The schedule that minimizes the sum

of the weights of tardy jobs is obtained by sorting the jobs in $SCHED_r$ by nonincreasing deadlines and then appending the REM_r jobs to the end of the schedule. This algorithm runs in $O(\log n)$ time using $O(n)$ processors.

Corollary 2: Defining jobs by release times equal to zero, integer deadlines, unit processing times, and weights, there is an algorithm for the job scheduling problem of minimizing the sum of the weights of tardy jobs that schedules the jobs to run on one machine in $O(\log n)$ time using $O(n)$ processors.

4.3. Problem 3

In the third job scheduling problem we are given n jobs with the same release times ($r_i = 0$), arbitrary deadlines and unit processing times. We want to minimize the number of tardy jobs when scheduling the jobs to run on one machine. Monma [Mon82] gives a sequential algorithm for solving this problem that runs in $O(n)$ time. An algorithm similar to the algorithm in section 3 gives an $O(\log n)$ time algorithm that uses $O(n)$ processors.

Corollary 3: Defining jobs by release times equal to zero, integer deadlines, and unit processing times, there is an algorithm for the job scheduling problem of minimizing the number of tardy jobs that schedules the jobs to run on one machine in $O(\log n)$ time using $O(n)$ processors.

Acknowledgements: I would like to thank Greg Frederickson and Dah Jyh Guan for discussions and comments.

5. References

[ACG87] Atallah, M. J., Cole, R., and Goodrich, M. T., Cascading Divide-and-Conquer: A Technique for Designing Parallel Algorithms, *Proc. of the 28th Annual Symp. on Foundations of Computer Science* (1987), 151–160.

[AGK88] Atallah, M. J., Goodrich, M. T., and Kosaraju, S. R., On the Parallel Complexity of Evaluating Some Sequences for Set Manipulation Operations, *3rd International Workshop on Parallel Computation and VLSI Theory (AWOC)* (1988).

[BLR77] Brucker, P., Rinnooy Kan, A. H. G., and Lenstra, J. K., Complexity of Machine Scheduling Problems, *Ann. Discrete Math.* 1 (1977), 343–362.

[Col86] Cole, R., Parallel Merge Sort, *Proc. of the 27th Annual Symp. on Foundations of Computer Science* (1986), 511–516.

[DeS83a] Dekel, E., and Sahni, S., Binary Trees and Parallel Scheduling Algorithms, *IEEE Trans. on Computers* C-32 (1983), 307–315.

[DeS83b] Dekel, E., and Sahni, S., Parallel Scheduling Algorithms, *Operations Research* 31 (1983), 24–49.

[DeS84] Dekel, E. and Sahni, S., A Parallel Algorithm for Convex Bipartite Graphs and Applications to Scheduling, *Journal of Parallel and Distributed Computing 1* (1984), 185–205.

[DUW86] Dolev, D., Upfal E. and Warmuth, M., The Parallel Complexity of Scheduling with Precedence Constraints, *Journal of Parallel and Distributed Computing 3* (1986), 553–576.

[Fre83] Frederickson, G. N. Scheduling Unit-Time Tasks With Integer Release Times and Deadlines, *Inf. Proc. Letters* 16 (1983), 171–173.

[Fre88] Frederickson, G. N. Private Communication.

[GJST81] Garey, M. R., Johnson, D. S., Simons, B. B., and Tarjan, R. E., Scheduling Unit-Time Tasks with Arbitrary Release Times and Deadline, *SIAM Journal of Computing* 10 (1981), 256–269.

[HeM84] Helmbold, D. and Mayr, E., Fast Scheduling Algorithms on Parallel Computers, STAN-CS-84-1025, Department of Computer Science, Standford University (1984), 1–27.

[Hor74] Horn, W. A., Some Simple Scheduling Algorithms, *Naval Research Logistics Quarterly* 21 (1974), 177–185.

[Jac55] Jackson, J. K., Scheduling a Production Line to Minimize Tardiness, Management Science Research Project, Univ. California, Los Angelos, CA, Res. Rep. 43, 1955.

[Mar88] Martel, C. U., A Parallel Algorithm for Preemptive Scheduling of Uniform Machines, Technical Report, Computer Science Division, University of California at Davis, (1988), 1–21.

[Mon82] Monma, C. L., Linear-Time Algorithms for Scheduling on Parallel Processors, *Operations Research* 30, (1982), 116–124.

[Sim78] Simons, B. B., A Fast Parallel Algorithm for Single Processor Scheduling, *Proc. of the 19th Annual Symp. on Foundations of Computer Science* (1978), 246–252.

[Val75] Valiant, L. G., Parallelism in Comparison Problems, *SIAM Journal of Computing* 4, (1975), 348–355.