An NC Algorithm for Finding Minimum Weighted Completion Time Schedule on Series Parallel Graphs

S. Sunder Xin He*
Department of Computer Science
State University of New York at Buffalo
Buffalo, NY 14260
sunder@cs.buffalo.edu xinhe@cs.buffalo.edu

Abstract

We present a parallel algorithm for solving the minimum weighted completion time scheduling problem for transitive series parallel graphs. The algorithm takes $O(\log^2 n)$ time with $O(n^3)$ processors on a CREW PRAM, where n is the number of vertices of the input graph. This is the first NC algorithm for solving the problem.

1 Introduction

Because of their practical applications and connection to other combinatorial and optimization problems, scheduling problems have played an important role in the field of algorithm design. Sequential algorithms for solving a variety of scheduling problems have been studied and many important results are known. (See [6] for a survey.)

On the other hand, very little is known about the parallel complexity of scheduling problems. Dekel and Sahni [4] presented a parallel algorithm for a special case of the single processor scheduling problem: each job has a release time, a deadline and unit processing time (but no precedence constraints). The algorithm takes $O(\log^2 n)$ time and $O(n)$ processors. Dolev, Upfal and Warmuth [5] gave an NC algorithm for the k-processor scheduling problem where the precedence constraints are specified by a directed tree. Independently, Helmbold and Mayr [8] obtained similar results. The classical two processor scheduling problem has been studied by many researchers.

*Research supported in part by NSF grant CCR-9011214

Vazirani and Vazirani showed that the two processor scheduling problem is in RNC by using matching techniques [17]. Helmbold and Mayr [9] found the first deterministic NC algorithm for the problem. The complexity of the algorithm is extremely high: it takes $O(\log^2 n)$ time with $O(n^{10})$ processors. Recently, Jung, Spirakis and Serna [11] improved their result and obtained an algorithm with $O(\log^2 n)$ time using $O(n^3)$ processors.

In this paper, we consider another well studied problem in scheduling theory. We are given a single processor and a set $N = \{s_1, \ldots, s_n\}$ of *jobs* to be executed on the processor. The precedence constraints are specified by a directed transitive acyclic graph $G = (N, E)$, where each node of the graph represents a job. (In the sequel, we will use the words *nodes* and *jobs* interchangeably.) Each job $s_i \in N$ has a positive processing time $p(s_i)$ and a weight $w(s_i)$. If there is an arc $s_i \rightarrow s_j$ in G, s_i must be executed before s_j. A *schedule* is a topological ordering σ of G. More precisely, $\sigma : N \rightarrow \{1, \ldots, n\}$ specifies a sequence of the nodes of N such that if $s_i \rightarrow s_j$ is an edge in G, then $\sigma(s_i) < \sigma(s_j)$. If $\sigma(s_j) = k$ then s_j is the kth job to be executed on the processor. The *weighted completion time* of σ is defined to be $cost(\sigma) = \sum_{s_j \in N} w(s_j)C(s_j)$, where $C(s_j) = \sum_{\sigma(s_i) \leq \sigma(s_j)} p(s_i)$ is the *completion time* of the job s_j under the schedule σ. We want to find an *optimal schedule* that minimizes the weighted completion time. In the following discussion, we will use the words *schedule* and *sequence* interchangeably.

This problem has a history of slow progress. Smith [15] showed that, in the absence of precedence constraints (i.e. the edge set of the constraint graph G is empty), an optimal schedule can be found by sequencing the jobs in non-increasing order of the ratios $w(s_j)/p(s_j)$. Conway, Maxwell and Miller [3] developed a procedure for solving the problem when G is in the form of vertex disjoint paths and all $w(s_j) = 1$. Horn [10] proposed algorithms for the case when G is a rooted tree. Adolphson and Hu [2] showed that such an algorithm can be implemented in $O(n \log n)$ time. Sidney [14] contributed a number of interesting theorems which apply to the problem with arbitrary precedence constraints. Lawler [12] gave an $O(n \log n)$ algorithm for the case when G is a *transitive series parallel* graph. He also showed that for general constraint graphs the problem is NP-complete even if all $w(s_j) = 1$ or all $p(s_j) = 1$.

We present a parallel algorithm for finding a minimum weighted completion time schedule for transitive series parallel graphs. Our algorithm is a sophisticated application of the tree contraction technique [1, 13]. The algorithm runs in $O(\log^2 n)$ time with $O(n^3)$ processors. The parallel computation model used in this paper is a parallel random access machine (PRAM).

The model consists of a number of identical processors and a common memory. In each time unit, a processor can read a memory cell, perform an arithmetic or a logic operation, and write into a memory cell. Concurrent read is allowed. Concurrent write to the same memory cell by different processors is disallowed.

The present paper is organized as follows. In section 2, we give necessary definitions and briefly review the sequential algorithm in [12, 14]. In section 3, we outline our parallel algorithm. Section 4 discusses the implementation of the parallel algorithm.

2 Backgrounds and the Sequential Algorithm

We first define the class of the constraint graphs discussed in this paper.

Definition 1: The class of *transitive series parallel* (TSP) graphs is defined recursively as follows:

1. A digraph consisting of a single node is a TSP graph.

2. If $G_1 = (N_1, E_1)$ and $G_2 = (N_2, E_2)$ are TSP graphs, then:

- **(a)** The graph $G = G_1 \times G_2 = (N_1 \cup N_2, E_1 \cup E_2 \cup (N_1 \times N_2))$ is also a TSP graph. G is said to be formed by the *series composition* of G_1 and G_2.
- **(b)** The graph $G = G_1 \cup G_2 = (N_1 \cup N_2, E_1 \cup E_2)$ is also a TSP graph. G is said to be formed by the *parallel composition* of G_1 and G_2.

Every TSP graph G can be represented by a rooted ordered binary tree $T = (V_T, E_T)$ called the *decomposition tree* of G. (To avoid confusion, the members of V_T will be called *vertices*.) The leaf vertices of T represent the nodes of G. The internal vertices of T are marked by either "S" or "P" to represent series or parallel composition respectively. We adopt the convention that the left child of an "S" vertex precedes the right child. Figure 1a shows a TSP graph G. (The edges implied by transitivity are omitted). Figure 1b shows a decomposition tree T of G. For each vertex u of T, let $T(u)$ denote the subtree of T rooted at u. $G(u)$ denotes the subgraph of G induced by the leaf vertices in $T(u)$. Clearly, $G(u)$ is a TSP graph and $T(u)$ is a decomposition tree of $G(u)$.

Figure 1

Given a TSP graph G with n nodes and m edges, a decomposition tree of G can be constructed in $O(m+n)$ sequential time [16], or in $O(\log^2 n)$ time with $O(n+m)$ processors on a PRAM [7].

We next describe the sequential algorithm in [12] for solving the scheduling problem. The presentation of the algorithm has been tailored in order to facilitate the parallel implementation. The algorithm performs a bottom-up computation on the decomposition tree T of G. For each vertex u of T, the algorithm computes a sequence $\sigma(u)$, which is an optimal schedule for the TSP graph $G(u)$. For each leaf vertex v of T, $G(v)$ consists of only one node v. So the sequence $\sigma(v) = \{v\}$. Consider any vertex u of T such that both the left child v and the right child w of u are leaf vertices with the optimal sequences $\sigma(v)$ and $\sigma(w)$ having been computed. The algorithm computes an optimal sequence $\sigma(u)$ of $G(u)$ from $\sigma(v)$ and $\sigma(w)$. Then v and w are deleted from T and u becomes a new leaf of T. This process is repeated until only the root r of T remains. The sequence $\sigma(r)$ is an optimal sequence of the graph $G = G(r)$. The algorithm is as follows:

Algorithm 1: Sequential-Scheduling
Input: A decomposition tree T of a TSP graph G.
Output: An optimal sequence σ of G.

1. For each leaf node v of T, let $\sigma(v) = \{v\}$.

2. Repeat:

2.1 Pick a vertex u of T such that both the left child v and the right child w of u are leaf vertices.

2.2 If u is an "S" vertex then $\sigma(u)$ =SeriesChain$(\sigma(v), \sigma(w))$.

Else (u is a "P" vertex) $\sigma(u)$ =ParallelChain$(\sigma(v), \sigma(w))$.

2.3 Delete v and w from T.

Until only the root r of T remains.

End Sequential-Scheduling.

The functions SeriesChain and ParallelChain describe how to compute $\sigma(u)$ from $\sigma(v)$ and $\sigma(w)$, depending on the type of the vertex u.

Case 1: u is an "S" vertex. Because each node in $G(v)$ precedes every node in $G(w)$, the nodes in $G(v)$ must precede the nodes in $G(w)$ in the sequence $\sigma(u)$. Hence:

$$\text{SeriesChain}(\sigma(v), \sigma(w)) = \sigma(v)//\sigma(w)$$

where // denotes the sequence concatenation.

Case 2: u is a "P" vertex. Before describing the ParallelChain function, we need some definitions. For any sequence I over N, define the weight of I to be $w(I) = \sum_{s_i \in I} w(s_i)$ and the processing time of I to be $p(I) = \sum_{s_i \in I} p(s_i)$. Define the *ratio of* I to be $\rho(I) = w(I)/p(I)$.

Definition 2: Let I be a sequence over N. An *initial set* of I is a prefix subsequence of I. The *ρ-maximal* initial set of I is the longest initial set I^* of I such that $\rho(I^*) \geq \rho(I')$ for all initial sets I' of I.

Definition 3: Let I be a sequence over N. The *composite job decomposition* of I is a partition of I into subsequences $I_1, \ldots, I_k$ such that:

1. $I = I_1//I_2// \ldots //I_k$.
2. For each j $(1 \leq j \leq k)$, I_j is the ρ-maximal initial set of the sequence $I_j//I_{j+1}// \ldots //I_k$.

Note that $\rho(I_1) > \rho(I_2) > \ldots > \rho(I_k)$ by the definition of ρ-maximal initial set. Each I_j will be called a *composite job* of I. The ParallelChain function is computed by the following algorithm.

Algorithm 2: ParallelChain (I, J)

1. Construct the composite job decomposition $\{J_1, \ldots, J_l\}$ of J.
2. Construct the composite job decomposition $\{I_1, \ldots, I_k\}$ of I.
3. Merge the composite jobs $\{I_1, \ldots, I_k\}$ and $\{J_1, \ldots, J_l\}$ into a sequence by non-increasing ρ values. (Ties are broken as follows: If $\rho(I_x) = \rho(J_y)$, then place J_y after I_x.)
4. Output the resulting sequence.

End ParallelChain.

It was shown in [12] that once a composite job I is formed by the ParallelChain algorithm, I will appear as a contiguous subsequence in the final solution sequence. Therefore, during subsequent processing, each composite job can be treated as a single job with weight $w(I)$ and processing time $p(I)$. The correctness of Algorithm 1 is based on a general theory of module decomposition of graphs [14]. Lawler showed that Algorithm 1 (with slight modifications) can be implemented in $O(n \log n)$ sequential time [12].

Note that Algorithm 2 treats the sequences I and J symmetrically. (In the step 3, the ties can actually be broken arbitrarily without affecting the correctness of the algorithm. The convention used in step 3 is only for convenience). The ParallelChain algorithm can also be interpreted in a different way: The algorithm finds the composite jobs J_t's of J and merges them into the sequence I. Thus the sequence constructed by ParallelChain consists of the nodes of I and the composite jobs of J. The two interpretations of ParallelChain are equivalent. However, we find the second interpretation is more convenient for the presentation of the parallel implementation.

3 Outline of the Parallel Algorithm

Our parallel implementation of Algorithm 1 is an application of the tree contraction method [1, 13]. We first briefly review the tree contraction algorithm as presented in [1].

Let Q be a set and $D \subseteq \{f \mid f : Q \times Q \to Q\}$ be a set of two variable functions over Q. Let $U \subseteq \{g \mid g : Q \to Q\}$ be a set of unary functions over Q. A *computation tree* is a rooted binary tree T. Each leaf vertex v of T is associated with a label $L(v) \in Q$. Each internal vertex u of T is associated with a function $F(u) \in D$. Each edge e of T is associated with a function $F(e) \in U$.

Given such a computation tree T, the *binary tree computation* on T is defined as follows. Consider a vertex u of T such that both the left child v and the right child w of u are leaf vertices with associated labels $L(v)$ and $L(w)$ respectively. Let $e_1 = (v, u)$ and $e_2 = (w, u)$. Compute a label $L(u) \in Q$ as follows:

$$L(u) = F(u)[F(e_1)(L(v)), F(e_2)(L(w))]$$

Then v and w are deleted and u becomes a leaf vertex of T. This procedure is repeated until only the root r of T remains. The label $L(r)$ is the goal of the binary tree computation problem.

In order to solve the binary tree computation problem in parallel, the following **Prune** operation is introduced in [1]. Consider a vertex u of T with the left child v, the right child w and the parent z. Let $e_1 = (v, u), e_2 = (w, u)$ and $e_0 = (u, z)$. Let $f = F(u)$ and $g_i = F(e_i)$ for $i = 0, 1, 2$.

Suppose v is a leaf vertex. The operation Prune(v) performs the following actions: v and u are deleted from T. w becomes a child of z. The new edge $e' = (w, z)$ is associated with a unary function $F(e') = g'$ defined by:

$$g'(x) = g_0(f[g_1(L(v)), g_2(x)])$$

Figure 2 illustrates the Prune(v) operation.

Figure 2

Suppose w is a leaf vertex. The operation Prune(w) performs the following actions: w and u are deleted from T. v becomes a child of z. The new edge $e'' = (v, z)$ is associated with a unary function $F(e'') = g''$ defined by:

$$g''(x) = g_0(f[g_1(x), g_2(L(w))])$$

It is easy to check that the Prune operation does not affect the label $L(r)$ to be computed. The tree contraction algorithm in [1] works as follows: First number all leaf vertices of T from left to right as: $v_1, v_2, \ldots, v_k$. Then for each leaf vertex v_i with odd index i, perform Prune(v_i) in parallel. It can be shown that in $O(\log n)$ iterations, T is reduced to the root r and the label $L(r)$ will have been obtained.

It is easy to see that Algorithm 1 is an instance of the binary tree computation problem: Consider a TSP graph $G = (N, E)$ with the decomposition tree T. We can change T to be a computation tree as follows. The set Q is the set of all sequences over N. The initial label $L(v)$ for each leaf vertex v of T is the sequence consisting of a single node v. For each internal vertex u of T, the function $F(u)$ is either the function defined by SeriesChain (if u is an "S" vertex), or the function defined by ParallelChain (if u is a "P" vertex). For each edge e of T, the function $F(e)$ is initially the identity function. The label $L(r)$ of the root r of T gives the solution of the scheduling problem. Thus we can use the tree contraction algorithm to solve our problem. The crux of the algorithm is to find a proper form of the functions associated with the edges of T and to implement the Prune operation. We will represent the functions associated with the edges of T by the *edge operators* defined as follows.

Definition 4: An *edge operator* is a 3-tuple $(A, B, \mathcal{C})$ and denoted by $EO[A, B, \mathcal{C}]$. $A = \{a_1, \ldots, a_p\}$ and $B = \{b_1, \ldots, b_q\}$ are sequences over N and are called the left and the right sequence respectively. $\mathcal{C} = \{C_1, C_2, \ldots, C_f\}$ is a list. Each item $C_i \in \mathcal{C}$ is a sequence over N and has two pointers l_i and r_i associated with it. $l_i \in \{-\infty, 1, \ldots, p\}$ is a pointer to a node $a_{l_i} \in A$. (If $l_i = -\infty$, it points to a dummy node before a_1.) $r_i \in \{1, \ldots, q, +\infty\}$ is a pointer to a node $b_{r_i} \in B$. (If $r_i = +\infty$, it points

to a dummy node after b_q.) These pointers satisfies the following nesting property: if $1 \leq i < j \leq f$ then $l_i \geq l_j$ and $r_i \leq r_j$ (see Figure 3).

Figure 3

Let $Z = \{z_1, \ldots, z_t\}$ be a sequence over N. For $1 \leq i \leq j \leq t$, we use $Z[z_i, z_j]$ to denote the subsequence of Z consisting of the nodes between z_i and z_j including both z_i and z_j. $Z[z_i, z_j)$ denotes the subsequence of Z consisting of the nodes between z_i and z_j including z_i but excluding z_j. The symbols $Z(z_i, z_j]$ and $Z(z_i, z_j)$ are defined similarly.

Given a sequence X, the edge operator $EO[A, B, \mathcal{C}]$ returns a sequence $Z = EO[A, B, \mathcal{C}](X)$ computed by the following sequential procedure.

Algorithm 3: Sequential-EO[A,B,$\mathcal{C}$](X)

1. Form the sequence $Z_0 = A//X//B$.

To simplify the presentation, we use the following notation: Assume $Z_0 = \{z_1, \ldots, z_g\}$. Let $\mathcal{C} = \{C_1, \ldots, C_f\}$. For each $C_i \in \mathcal{C}$, the left pointer l_i points to the node z_{l_i} and the right pointer r_i points to the node z_{r_i}. (If $l_i = -\infty$ then l_i points to a dummy node before z_1. If $r_i = +\infty$, then r_i points to a dummy node after z_g.)

2. For $j = 1$ to f do:

2.1. Let $Y = Z_{j-1}(z_{l_j}, z_{r_j})$.

2.2. Y' =ParallelChain(Y, C_j).

2.3. In Z_{j-1}, replace the subsequence Y by Y'. Let Z_j be the resulting sequence.

3. Return $Z = Z_f$.

End Sequential-EO.

This algorithm uses the second interpretation of the ParallelChain algorithm. For each $1 \leq i \leq g + 1$, we call $Z_0(i - 1, i)$ the *slot* i. (The index 0 represents $-\infty$. The index $g + 1$ represents $+\infty$.) By calling ParallelChain(Y, C_j) at step 2.2, Sequential-EO decomposes each $C_j \in \mathcal{C}$ into the set $D_j = \{d_{j1}, \ldots, d_{jk_j}\}$ (for some k_j) of composite jobs and assigns each d_{jk} into a slot $Z_0(t - 1, t)$ with $l_j < t \leq r_j$. The sequence Z returned by Sequential-EO consists of the nodes of Z_0 and the composite jobs $\{d_{jk} \mid 1 \leq j \leq f,\ 1 \leq k \leq k_j\}$. Note that the slot for each d_{jk} is completely

determined from Z_0 and the composite jobs in $\bigcup_{i=1}^{j} D_i$. The sequence Z preserves the order of the nodes in Z_0. For each j, the composite jobs d_{jk}'s of C_j appear in Z in decreasing order of ρ values. In each slot $Z_0(t-1,t)$, the composite jobs appear in nonincreasing order of ρ values. If two composite jobs d_{xy} and d_{uv} are placed in the same slot and $\rho(d_{xy}) = \rho(d_{uv})$ and $x > u$, then d_{xy} appears after d_{uv} in the slot.

By this definition, $EO[\emptyset, \emptyset, \emptyset]$ is the identity operator.

Given two edge operators $EO[A_1, B_1, \mathcal{C}_1]$ and $EO[A_2, B_2, \mathcal{C}_2]$, we can construct a new edge operator $EO[A, B, \mathcal{C}]$ as follows. Let $A = A_1//A_2$, $B = B_2//B_1$ and $\mathcal{C} = \mathcal{C}_2//\mathcal{C}_1$. ($\mathcal{C}$ is the concatenation of the two lists). For each $C_i \in \mathcal{C}_1$, the left pointer l_i and the right pointer r_i are not changed. For each $C_i \in \mathcal{C}_2$ such that $l_i \neq -\infty$ and $r_i \neq +\infty$, the pointers l_i and r_i are not changed. For a $C_i \in \mathcal{C}_2$ with $l_i = -\infty$, let l_i point to the last node of A_1. (If $A_1 = \emptyset$, l_i remains $-\infty$.) If $r_i = +\infty$, let r_i point to the first node of B_1. (If $B_1 = \emptyset$, r_i remains $+\infty$).) It is easy to check that $EO[A, B, \mathcal{C}]$ defines a new edge operator and is the functional composition of $EO[A_1, B_1, \mathcal{C}_1]$ and $EO[A_2, B_2, \mathcal{C}_2]$. Namely for any sequence X, $EO[A, B, \mathcal{C}](X) = EO[A_1, B_1, \mathcal{C}_1](EO[A_2, B_2, \mathcal{C}_2](X))$. We will write $EO[A, B, \mathcal{C}] = EO[A_1, B_1, \mathcal{C}_1] \circ EO[A_2, B_2, \mathcal{C}_2]$.

We are now ready to describe the Prune operation. Let u be a vertex in T with the left child v and the right child w. Let z be the parent of u in T. Let $e_1 = (v, u)$, $e_2 = (w, u)$ and $e_0 = (u, z)$. For $i = 0, 1, 2$, suppose the edge e_i is associated with the edge operator $F(e_i) = EO[A_i, B_i, \mathcal{C}_i]$.

Suppose v is a leaf vertex with a label $L(v) = X$. When the operation Prune(v) is performed, v and u are deleted from T, w becomes a child of z and a new edge $e' = (w, z)$ is introduced. We need to define an edge operator $F(e')$ for e'. Depending on the type of u, there are two cases.

Case 1: u is an "S" vertex. Let $Z = EO[A_1, B_1, \mathcal{C}_1](X)$ and define $F(e') = EO[A_0, B_0, \mathcal{C}_0] \circ EO[Z//A_2, B_2, \mathcal{C}_2]$. (In $EO[Z//A_2, B_2, \mathcal{C}_2]$, for each $C_i \in \mathcal{C}_2$ with $l_i = -\infty$, let l_i point to the last node of Z.)

Case 2: u is a "P" vertex. Let $Z = EO[A_1, B_1, \mathcal{C}_1](X)$. Form a new edge operator $EO[A_2, B_2, \mathcal{C}_2']$ as follows: The list $\mathcal{C}_2'$ is obtained from $\mathcal{C}_2$ by adding an item $C' = Z$ at the end of $\mathcal{C}_2$. The left pointer l' for C' is set to be $-\infty$. The right pointer r' for C' is set to be $+\infty$. Define $F(e') = EO[A_0, B_0, \mathcal{C}_0] \circ EO[A_2, B_2, \mathcal{C}_2']$.

Suppose w is a leaf vertex with a label $L(v) = X$. When the operation Prune(w) is performed, w and u are deleted from T, v becomes a child of z and a new edge $e'' = (v, z)$ is introduced. We need to define an edge

operator $F(e'')$ for e''. Depending on the type of u, there are two cases.

Case 3: u is an "S" vertex. Let $Z = EO[A_2, B_2, \mathcal{C}_2](X)$ and define $F(e'') = EO[A_0, B_0, \mathcal{C}_0] \circ EO[A_1, B_1//Z, \mathcal{C}_1]$. (In $EO[A_1, B_1//Z, \mathcal{C}_1]$, for each $C_i \in \mathcal{C}_1$ with $r_i = +\infty$, let r_i point to the first node of Z.)

Case 4: u is a "P" vertex. Let $Z = EO[A_2, B_2, \mathcal{C}_2](X)$. Form a new edge operator $EO[A_1, B_1, \mathcal{C}_1']$ as follows: The list $\mathcal{C}_1'$ is obtained from $\mathcal{C}_1$ by adding an item $C' = Z$ at the end of $\mathcal{C}_1$. The left pointer l' for C' is set to be $-\infty$. The right pointer r' for C' is set to be $+\infty$. Define $F(e'') = EO[A_0, B_0, \mathcal{C}_0] \circ EO[A_1, B_1, \mathcal{C}_1']$.

By these definitions, it is easy to show that $F(e')$ and $F(e'')$ satisfy the properties required by the Prune operation. In order to implement the Prune operation, we need to construct the edge operators $F(e')$ and $F(e'')$. This involves the evaluation of an edge operator $EO[A, B, \mathcal{C}]$ on a sequence X and the computation of the composition of two edge operators. The composition of two edge operators requires only the concatenation of sequences. This can be done in $O(1)$ time. The evaluation of the edge operators, however, is a sequential process. In the next section, we show how to evaluate edge operators in parallel.

4 Evaluating Edge Operators in Parallel

We first describe a parallel algorithm for finding the composite job decomposition of a sequence $I = \{s_1, \ldots, s_p\}$. This algorithm is used as a subroutine by our parallel algorithm for evaluating edge operators.

Algorithm 4: Decompose(I)

1. For all i, j such that $1 \leq i \leq j \leq p$, compute the quantity $\sum_{k=i}^{j} w(s_k)$ and $\sum_{k=i}^{j} p(s_k)$. This can be done by using the prefix-sum algorithm.

2. For each subsequence of I starting at s_i, find the maximal index i_{max} such that $(\sum_{k=i}^{i_{max}} w(s_k))/(\sum_{k=i}^{i_{max}} p(s_k))$ is maximum.

3. Define a forest F on $\{1, \ldots, p\}$ with the following edges:

$\{(i, next_i) | 1 \leq i < p,\ i_{max} < p \text{ and } next_i = i_{max} + 1\}$.

4. Find the unique longest path of the forest F starting at 1 by the pointer jump technique. The vertices on this path give the indices of the first node of the composite jobs of I.

End Decompose.

The following is our parallel algorithm for evaluating an edge operator $EO[A, B, \mathcal{C}]$ on a sequence X.

Algorithm 5: Parallel-EO$[A, B, \mathcal{C}](X)$

1. Form the sequence $Z_0 = A//X//B$.

To simplify the presentation, we use the following notation: Assume $Z_0 = \{z_1, \ldots, z_g\}$. For each $C_i \in \mathcal{C} = \{C_1, \ldots, C_f\}$, the left pointer l_i points to the node z_{l_i} and the right pointer r_i points to the node z_{r_i}. (If $l_i = -\infty$ then l_i points to a dummy node before z_1. If $r_i = +\infty$, then r_i points to a dummy node after z_g.)

2. For each sequence $C_j \in \mathcal{C}$, let $D_j = \{d_{j1}, \ldots, d_{jk_j}\}$ (for some k_j) be the composite job decomposition of C_j obtained by calling Decompose(C_j).

3. For each d_{jk} $(1 \leq j \leq f,\ 1 \leq k \leq k_j)$, construct a sequence Z_{jk} as follows:

- Form the subsequence $Z_0(z_{l_j}, z_{r_j})$.
- Consider all nodes $d_{il} \in (\bigcup_{t=1}^{j} D_t) - \{d_{jk}\}$. If $\rho(d_{il}) \geq \rho(d_{jk})$, put d_{il} into the slot $Z_0(l_i, l_i + 1)$. (Namely, d_{il} is put into the leftmost allowable slot.) If $\rho(d_{il}) < \rho(d_{jk})$, put d_{il} into the slot $Z_0(r_i - 1, r_i)$. (Namely, d_{il} is put into the rightmost allowable slot.) Within each slot, order the composite jobs by nonincreasing ρ values. Break ties as follows: If $\rho(d_{xy}) = \rho(d_{uv})$ and $x > u$, then place d_{xy} after d_{uv}.

4. For each d_{jk}, compute Y_{jk} =ParallelChain(Z_{jk}, d_{jk}). Let K_{jk} be the slot assigned to d_{jk}. Namely d_{jk} is placed in the slot $Z_0(K_{jk} - 1, K_{jk})$.

5. Form the sequence $\overline{Z}$ by placing each composite job d_{jk} into the slot $Z_0(K_{jk} - 1, K_{jk})$. Within each slot, order the composite jobs by nonincreasing ρ values. Break ties as follows: If $\rho(d_{xy}) = \rho(d_{uv})$ and $x > u$, then place d_{xy} after d_{uv}.

6. Return $\overline{Z}$

End Parallel-EO.

We will show the sequence Z =Sequential-EO$[A, B, \mathcal{C}](X)$ and the sequence $\overline{Z}$ =Parallel-EO$[A, B, \mathcal{C}](X)$ are identical. By the remarks after Algorithm 3 and the step 5 of Algorithm 5, the composite jobs within each slot are ordered according to the same rule in Z and $\overline{Z}$. So it is enough to prove that the slot K_{jk} assigned to each composite job d_{jk} by Parallel-EO is identical to the slot assigned to d_{jk} by Sequential-EO.

The following lemma will be used repeatedly. Its proof is straightforward.

Lemma 1: Let $a, b, c, d, r > 0$.
(a) If $\frac{a}{b} \geq r$ and $\frac{c}{d} \geq r$, then $\frac{a+c}{b+d} \geq r$.
(b) If $\frac{a}{b} \leq \frac{a+c}{b+d}$, then $\frac{c}{d} \geq \frac{a+c}{b+d}$.
(c) If $\frac{a+c}{b+d} \geq r$ and $\frac{c}{d} < r$, then $\frac{a}{b} \geq r$. □

Definition 5: Let $U = \{u_1, \ldots, u_p\}$ be a sequence and $u_j \in U$.

1. We say u_j satisfies the **left constraint** with respect to U if $\rho(U[u_t, u_j)) \geq \rho(u_j)$ for all t with $1 \leq t \leq j-1$.

2. We say u_j satisfies the **right constraint** with respect to U if $\rho(U(u_j, u_t]) < \rho(u_j)$ for all t with $j+1 \leq t \leq p$.

The next lemma gives a key property of the ParallelChain algorithm.

Lemma 2: Let A and B be two sequences over N. Let $\{B_1, \ldots, B_p\}$ be the composite job decomposition of B. Let U = ParallelChain(A, B). Then the left and the right constraints are satisfied for each composite job B_d $(1 \leq d \leq p)$ with respect to U.

Proof : Suppose $U = \{u_1, \ldots, u_k\}$, where each u_i is either a node of A or a composite job of B. Consider a composite job B_d of B. Suppose B_d corresponds to u_j in U. We only prove the left constraint is satisfied for u_j. (The proof for the right constraint is similar.) Consider a node $u_t \in U$ with $1 \leq t \leq j-1$. We need to show $\rho(U[u_t, u_j)) \geq \rho(u_j)$.

Let $\{A_1, \ldots, A_q\}$ be the composite job decomposition of A. The sequence $U[u_1, u_j)$ can be divided into composite jobs $D_1, \ldots, D_l$ where each D_k is either a composite job of A (in this case D_k consists of several members of U) or a composite job of B (in this case D_k corresponds to a single member of U). The ParallelChain algorithm ensures:

$$\rho(D_1) \geq \rho(D_2) \geq \ldots \geq \rho(D_l) \geq \rho(u_j) \tag{1}$$

Suppose the composite job D_i contains u_t.

Case 1: u_t is the first job of D_i. (This includes the case where D_i is a composite job of B.) In this case, $U[u_t, u_j) = \bigcup_{k=i}^{l} D_k$. (See Figure 4.1). By (1) and Lemma 1 (a), we have: $\rho(U[u_t, u_j)) \geq \rho(u_j)$.

Figure 4

Case 2. u_t is not the first job of D_i. Suppose $D_i = U[u_r, u_s]$, where $r < t \leq s$. (See Figure 4.2). In this case, D_i is a composite job of A. By the definition of the composite job decomposition, we have:

$$\rho(U[u_r, u_t)) \leq \rho(U[u_r, u_s]) \tag{2}$$

By Lemma 1 (b), this implies:

$$\rho(U[u_t, u_s]) \geq \rho(U[u_r, u_s]) = \rho(D_i) \geq \rho(u_j) \tag{3}$$

Clearly, $U[u_t, u_j) = U[u_t, u_s] \cup D_{i+1} \cup \ldots \cup D_l$. By (1), (3) and Lemma 1 (a), we have: $\rho(U[u_t, u_j)) \geq \rho(u_j)$ as to be shown. □

Consider the evaluation of Sequential-EO$[A, B, \mathcal{C}](X)$. Suppose $Z_0 = A//X//B = \{z_1, \ldots, z_g\}$. Let Z_i be the sequence constructed by Sequential-EO during the ith iteration of the for loop. Let z_{l_i} and z_{r_i} be the nodes of Z_0 pointed to by the left pointer l_i and the right pointer r_i respectively. Lemma 2 implies that the left and the right constraints are satisfied for each composite job x of C_i with respect to the sequence $Z_i(z_{l_i}, z_{r_i})$. Consider the sequence Z_j $(j > i)$ constructed by Sequential-EO during the jth iteration of the for loop. The sequence $Z_j(z_{l_i}, z_{r_i})$ contains the nodes in $Z_i(z_{l_i}, z_{r_i})$ and might contain some composite jobs of some C_t with $j \geq t > i$. The following lemma states that the left and the right constraints are satisfied for each composite job x of C_i with respect to $Z_j(z_{l_i}, z_{r_i})$.

Lemma 3: Let $j > i$ be two integers. Then the left and the right constraints are satisfied for each composite job x of C_i with respect to the sequence $Z_j(z_{l_i}, z_{r_i})$.

Proof: The proof is by reverse induction on i. Consider the base case $i = j$. $Z_j(z_{l_j}, z_{r_j})$ is obtained by calling ParallelChain$(Z_{j-1}(z_{l_j}, z_{r_j}), C_j)$. By Lemma 2, the left and the right constraints are satisfied for each composite job x of C_j with respect to $Z_j(z_{l_j}, z_{r_j})$.

Suppose that for all t with $j \geq t > i$, the left and the right constraints are satisfied for each composite job of C_t with respect to $Z_j(z_{l_t}, z_{r_t})$. Consider a composite job x of C_i. We prove the left constraint is satisfied for x with respect to $Z_j(z_{l_i}, z_{r_i})$. (The proof for the right constraint is similar.) Let y be a node in $Z_j(z_{l_i}, x)$. We need to show $\rho(Z_j[y, x)) \geq \rho(x)$.

The sequence $Z_j[y,x)$ consists of the nodes in the sequence $Z_i[y,x)$ and (probably) some composite jobs of some C_t with $j \geq t > i$. Let $U = Z_j[y,x) - Z_i[y,x)$. If $U = \emptyset$ then $Z_j[y,x) = Z_i[y,x)$. By Lemma 2, we have $\rho(Z_j[y,x)) = \rho(Z_i[y,x)) \geq \rho(x)$.

Figure 5

Suppose $U \neq \emptyset$. Let $U = \{u_1, \ldots, u_q\}$ be listed from left to right. For $2 \leq k \leq q$, let $U_k = Z_j(u_{k-1}, u_k)$. Let $U_1 = Z_j[y, u_1)$ and $U_{q+1} = Z_j(u_q, x)$. (See Figure 5). For each k with $1 \leq k \leq q$, u_k is a composite job of some C_t with $t > i$. By the nesting property of the left and the right pointers of the edge operator, the sequence $Z_j(z_{l_t}, z_{r_t})$ contains $Z_j(z_{l_i}, z_{r_i})$ as a subsequence. Thus by induction hypothesis we have: $\rho(U_k) < \rho(u_{k-1})$ for $2 \leq k \leq q+1$ and $\rho(U_k) \geq \rho(u_k)$ for $1 \leq k \leq q$. Since $U_{q+1} = Z_j(u_q, x) = Z_i(u_q, x)$, by Lemma 2 we have $\rho(U_{q+1}) \geq \rho(x)$. Hence:

$$\rho(U_1) \geq \rho(u_1) > \rho(U_2) \geq \rho(u_2) > \ldots \geq \rho(u_q) > \rho(U_{q+1}) \geq \rho(x) \quad (4)$$

Note that some U_k might be empty. In this case, we can simply drop the item corresponding to an empty U_k from above inequalities. For example, if $U_2 = \emptyset$, then by the induction hypothesis (the left constraint for u_2), we have $\rho(u_1) \geq \rho(u_2)$. So the item $\rho(U_2)$ can be omitted in above inequalities without affecting its validity.

Clearly, $Z_j[y,x) = \{u_1, \ldots, u_q\} \bigcup \{U_1, \ldots, U_q, U_{q+1}\}$. By Lemma 1 (a) and (4), we have: $\rho(Z_j[y,x)) \geq \rho(x)$. This completes the induction. □

The following lemma establishes the correctness of the Parallel-EO algorithm.

Lemma 4: Consider any composite job d_{jk}. Suppose d_{jk} is assigned into the slot $Z_0(J-1, J)$ by Parallel-EO and is assigned into the slot $Z_0(I-1, I)$ by Sequential-EO. Then $I = J$.

Proof: Consider the sequence Y_{jk} constructed for d_{jk} by Parallel-EO and the sequence Z_j constructed by Sequential-EO during the jth iteration of the for loop. The slot J assigned to d_{jk} by Parallel-EO is completely determined from Y_{jk}. The slot I assigned to d_{jk} by Sequential-EO is completely determined from Z_j. Recall that for each composite job x of C_i with $i \leq j$, the position of x in Z_j is between z_{l_i} and z_{r_i}. For $x \neq d_{jk}$, if $\rho(x) \geq \rho(d_{jk})$, the position of x in Y_{jk} is in the slot $Z_0(l_i, l_i + 1)$. If $\rho(x) < \rho(d_{jk})$, the position of x in Y_{jk} is in the slot $Z_0(r_i - 1, r_i)$.

Toward a contradiction, assume $J < I$. Let b be the rightmost node of Z_j in the slot $Z_0(J-1, J)$ such that $\rho(b) \geq \rho(d_{jk})$. (If no such b exists, let

b be the node z_{J-1} in Z_0). Let u be the leftmost node in $Z_j(b, d_{jk}]$ such that $\rho(u) \geq \rho(d_{jk})$ and u appears in Y_{jk} before d_{jk}. Suppose the position of u in Z_j is in the slot $Z_0(K-1, K)$. (If no such u exists, let $u = d_{jk}$ and $K = I$). Let c be the rightmost node of Y_{jk} in the slot $Z_0(K-1, K)$ such that $\rho(c) \geq \rho(d_{jk})$. (If no such c exists, let $c = z_{K-1}$.) (See Figure 6.1). Let $A = Y_{jk}(d_{jk}, c)$ and $B = Z_j(b, u)$. Note that in Z_j, b and u cannot be in the same slot by the definition of b and u. Therefore $A \cap B$ contains at least one node $z_J \in Z_0$. Hence $A \cap B \neq \emptyset$.

Figure 6

Let $U = B - A$ and $W = A - B$. Define:
$U_1 = \{x \in U \mid \rho(x) \geq \rho(d_{jk})\}$.
$U_2 = \{x \in U \mid \rho(x) < \rho(d_{jk})\}$.
$W_1 = \{x \in W \mid \rho(x) \geq \rho(d_{jk})\}$.
$W_2 = \{x \in W \mid \rho(x) < \rho(d_{jk})\}$.

First we show $U_1 = \emptyset$. Suppose not, let $x \in U_1$. Since $\rho(x) \geq \rho(d_{jk})$ and $x \in B - A$, x appears in Y_{jk} before the node d_{jk}. However, u is the leftmost node in $U(b, d_{jk}]$ with this property. Hence such an x cannot exist and we must have $U_1 = \emptyset$.

Suppose u is a composite job of C_i for some $i \leq j$. Let z_{l_i} and z_{r_i} be the nodes pointed to by the left pointer l_i and the right pointer r_i respectively.

Next we examine the position of the nodes of W_2 in Z_j. Suppose $W_2 = \{w_1, w_2, \ldots, w_t\}$ listed from left to right in Z_j.

Claim: w_1 appears in Z_j after z_{l_i}. w_t appears in Z_j before b.

Proof of the claim: Suppose w_1 is a composite job of $C_{i'}$ for some $i' \leq j$. Let $z_{l_{i'}}$ and $z_{r_{i'}}$ be the nodes of Z_0 pointed to by the left pointer $l_{i'}$ and the right pointer $r_{i'}$ respectively. Since $\rho(w_1) < \rho(d_{jk})$, w_1 is placed in Y_{jk} by Parallel-EO in the slot $Z_0(r_{i'} - 1, r_{i'})$. Since the position of w_1 in Y_{jk} is in a slot between the slot J and the slot K, we have: $J \leq r_{i'} \leq K$.

Since $\rho(w_1) < \rho(d_{jk})$ and c is the rightmost node of Y_{jk} in the slot $Z_0(K-1, K)$ such that $\rho(c) \geq \rho(d_{jk})$, w_1 cannot be in the slot $Z_0(K-1, K)$ in Y_{jk}. Hence $r_{i'} < K$.

Since the position of u in Z_j is in the slot $Z_0(K-1, K)$, we must have $K \leq r_i$. Therefore:

$$r_{i'} < K \leq r_i \tag{5}$$

By the nesting property of the left and the right pointers of the edge operator, (5) implies $l_{i'} \geq l_i$. Thus w_1 must appear after z_{l_i} in Z_j.

Since $\rho(w_t) < \rho(d_{jk})$ and $w_t \in A - B$, clearly, w_t must appear in Z_j before b. This completes the proof of the claim.

By the claim, $w_1 \in Z_j(z_{l_i}, u)$. Thus by Lemma 3, we have:

$$\rho(Z_j[w_1, u)) \geq \rho(u) \geq \rho(d_{jk}) \tag{6}$$

Let $B_p = Z_j(w_p, w_{p+1})$ for $1 \leq p \leq t-1$ and let $B_t = Z_j(w_t, b]$. (See Figure 6.2). By Lemma 3, we have:

$$\rho(B_p) < \rho(w_p) < \rho(d_{jk}), \qquad for \quad 1 \leq p \leq t \tag{7}$$

Let $B' = B \cup \{w_1, \ldots, w_t\}$. Note that $B' = Z_j[w_1, u) - (\bigcup_{p=1}^{t} B_p)$. By (6), (7) and Lemma 1 (c), we get:

$$\rho(B') \geq \rho(d_{jk}) \tag{8}$$

Since $\rho(x) \geq \rho(d_{jk})$ for all $x \in W_1$, by (8) and Lemma 1 (a), we have:

$$\rho(B' \cup W_1) \geq \rho(d_{jk}) \tag{9}$$

Since $\rho(x) < \rho(d_{jk})$ for all $x \in U_2$, by (9) and Lemma 1 (c), we have:

$$\rho(B' \cup W_1 - U_2) \geq \rho(d_{jk}) \tag{10}$$

Note that the fact $A \cap B \neq \emptyset$ ensures $B' \cup W_1 - U_2 \neq \emptyset$. So the application of Lemma 1 (c) is valid.

Clearly, $A = B \cup (W_1 \cup W_2) - (U_1 \cup U_2) = B' \cup W_1 - U_2$. Hence $\rho(A) \geq \rho(d_{jk})$. However, by applying Lemma 2 to the sequence Y_{jk}, we must have $\rho(A) < \rho(d_{jk})$. This contradiction shows that the assumption $J < I$ is wrong. Thus $J \geq I$. Similarly, we can show $J \leq I$. Hence $J = I$. □

In summary, we have the following:

Theorem 5: The minimum weighted completion time scheduling problem for TSP graphs can be solved in $O(\log^2 n)$ time with $O(n^3)$ processors on a PRAM, where n is the number of vertices of the input graph.

Proof: Since the sequences constructed by Sequential-EO and Parallel-EO are identical, our implementation of Algorithm 1 is correct.

The implementation of Algorithm 1 is an application of the tree contraction algorithm. The construction of the composite job decomposition of a sequence I can be done by calling Decompose(I) which takes $O(\log p)$ time with $O(p^2)$ processors (where p is the number of nodes in I). Hence

ParallelChain(I, J) can also be done within the same time and processor bounds.

In the algorithm Parallel-EO$[A, B, \mathcal{C}](X)$, the step 4 dominates the complexity. This step (and the entire Algorithm 5) can be implemented on a PRAM using $O(p^3)$ processors and $O(\log p)$ time, where p is the total number of nodes in $A, B, \mathcal{C}$, and X. This implies that the Prune operation can be implemented in $O(\log p)$ time with $O(p^3)$ processors (where p is the total number of nodes of the edge operators involved in the operation). At any stage of the tree contraction algorithm, the Prune operation is performed on a set of leaf vertices of the decomposition tree T. The total number of nodes involved in all these operations is bounded by n. Therefore each stage of the tree contraction algorithm can be done in $O(\log n)$ time with $O(n^3)$ processors. The tree contraction algorithm terminates in $O(\log n)$ stages. This gives the claimed time and the processor bounds. □

References

[1] K. Abrahamson, N. Dadoun, D. G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *J. Algorithms*, 10:287–302, 1989.

[2] D. Adolphson and T. C. Hu. Optimal linear ordering. *SIAM Journal Of Applied Math.*, 25:403–423, 1973.

[3] R. W. Conway, W. L. Maxwell, and L. W. Miller. *Theory Of Scheduling.* Addison-Wesley, Reading, Mass., 1967.

[4] E. Dekel and S. Sahni. A parallel matching algorithm for convex bipartite graphs and applications to scheduling. *Journal of Parallel and Distributed Computing*, 1:185–205, 1984.

[5] D. Dolev, E. Upfal, and M. Warmuth. Scheduling trees in parallel. In P. Bertolazzi and F. Luccio, editors, *VLSI: Algorithms and Architectures*, 1985.

[6] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling : a survey.

[7] Xin He. Parallel recognition and decomposition of two terminal series parallel graphs. *Information and Computation*, 75:15–38, 1987.

[8] D. Helmbold and E. Mayr. Fast scheduling algorithms on parallel computers. In F. Prepparata, editor, *Advances in Computing Research*, volume 4. 1987.

[9] David HelmBold and Ernst Mayr. Two processor scheduling is in nc. *SIAM Journal Of Computing*, 16:747–759, August 1987.

[10] W. A. Horn. Single machine job sequencing with treelike precedence ordering and linear delay penalties. *SIAM Journal Of Applied Math.*, 23:189–202, 1972.

[11] H. Jung, P. Spirakis, and M. Serna. A parallel algorithm for two processors precedence constrained scheduling. In *Proceedings of ICALP*, 1991.

[12] Eugene L. Lawler. Sequencing jobs to minimize total weighted completion time subject to precedence constraints. In B.Alspach, P. Hell, and D.J. Miller, editors, *Algorithmic Aspects Of Combinatorics*, volume 2 of *Annals of Discrete Mathematics*, pages 75–90. North Holland Publishing Company, 1978.

[13] G. L. Miller and J. Reif. Parallel tree contraction and its applicaton. In *26th IEEE FOCS*, pages 478–489, 1985.

[14] Jeffrey B. Sidney. Decomposition algorithms for single-machine sequencing with precedence relations and deferral costs. *Operations Research*, 23:283–298, March-April 1975.

[15] W. E. Smith. Various optimizers for single stage production. *Naval Res. Logist. Quart.*, 3:59–66, 1956.

[16] J. Valdez, R. E. Tarjan, and E. L. Lawler. The recognition of series parallel graphs. In *11th ACM STOC*, pages 1–12, 1979.

[17] U. V. Vazirani and V. V. Vazirani. Two processor scheduling is in rnc. In *17th ACM STOC*, pages 11–21, 1985.

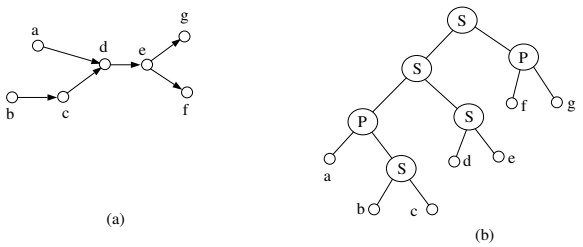


Figure 1. (a) A TSP graph and (b) its decomposition tree.

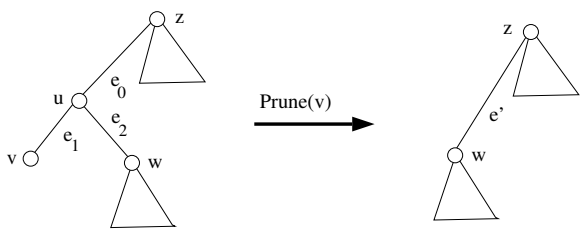
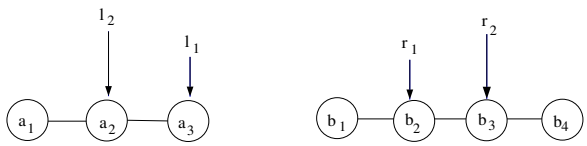


Figure 2. Prune(v) operation.



$l_1 = 3,\ l_2 = 2,\ r_1 = 2, r_2 = 3$

Figure 3: EO[$\{a_1, a_2, a_3\}, \{b_1, b_2, b_3, b_4\}, \{C_1, C_2\}$]

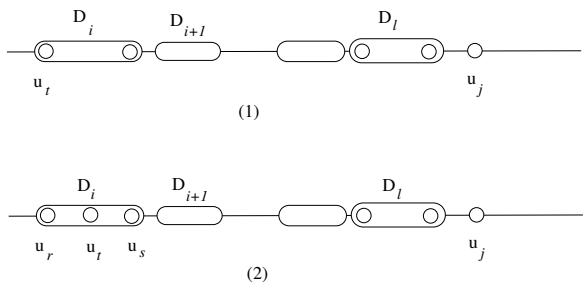


Figure 4. The proof of Lemma 2.

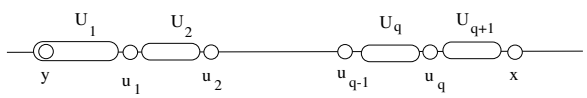


Figure 5. The proof of Lemma 3.

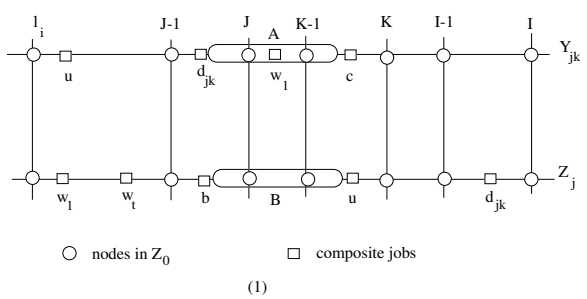


Figure 6. The proof of Lemma 4.