Algorithmica (2003) 35: 321–345
DOI: 10.1007/s00453-002-1004-3

Algorithmica

Detecting Race Conditions in Parallel Programs that Use Semaphores[1]

Philip N. Klein,[2] Hsueh-I Lu,[3] and Robert H. B. Netzer[2]

Abstract. We address the problem of detecting race conditions in programs that use semaphores for synchronization. Netzer and Miller showed that it is NP-complete to detect race conditions in programs that use many semaphores. We show in this paper that it remains NP-complete even if only two semaphores are used in the parallel programs.

For the tractable case, i.e., using only one semaphore, we give two algorithms for detecting race conditions from the trace of executing a parallel program on p processors, where n semaphore operations are executed. The first algorithm determines in $O(n)$ time whether a race condition exists between any two given operations. The second algorithm runs in $O(np \log n)$ time and outputs a compact representation from which one can determine in $O(1)$ time whether a race condition exists between any two given operations. The second algorithm is near-optimal in that the running time is only $O(\log n)$ times the time required simply to write down the output.

Key Words. Race conditions, Parallel programs, Semaphores, Tractability, Scheduling.

1. Introduction. Race detection is crucial in developing and debugging shared-memory parallel programs [5], [7], [11], [16]–[18]. Explicit synchronization is usually added to such programs to coordinate access to shared data. For example, when using a semaphore, a V-operation increments the semaphore, and a P-operation waits until the semaphore is greater than zero and then decrements the semaphore. P-operations are typically used to wait (synchronize) until some condition is true (such as a shared buffer becoming nonempty), and V-operations typically signal that some condition is now true. Race conditions result when this synchronization does not force concurrent processes to access data in the expected order. One way to detect races in a program dynamically is to trace its execution and analyze the traces afterward. A central part of dynamic race detection is to compute from the trace the order in which shared-memory accesses were guaranteed by the execution's synchronization to have executed. Accesses to the same location not guaranteed to execute in some particular order are considered a race. When programs use semaphore operations for synchronization, some operations (belonging to different processes) could have potentially executed in an order different than what was traced.

In this paper we address the tractability of detecting race conditions from the traces of parallel programs that use semaphores. Let p be the number of processors used to execute the parallel program, and let n be the total number of semaphore operations

[1] Preliminary versions of this paper appeared in [12] and [13]. This research was supported in part by National Science Foundation Presidential Young Investigator Award CCR-9047466.

[2] Department of Computer Science, Brown University, Providence, RI 02912-1910, USA. {klein,rn}@cs.brown.edu.

[3] Institute of Information Science, Academia Sinica, Taipei 115, Taiwan. hil@iis.sinica.edu.tw. www.iis.sinica.edu.tw/~hil/. Most of the results of this paper were done when this author was a Ph.D. student at Brown University.

Received October 28, 2000; revised November 3, 2001. Communicated by F. Thomson Leighton.
Online publication January 13, 2003.

performed in the execution. The trace can then be represented by a directed n-node graph G consisting of p disjoint chains, each represents the sequence of semaphore operations executed by a processor. A *schedule* of G is a linear ordering of all nodes in G consistent with the precedence constraints imposed by the arcs of G. A prefix of a schedule of G is a *subschedule* of G. A subschedule of G is *valid* if at each point in the subschedule, the number of V operations is never exceeded by the number of P operations for each semaphore (i.e., all semaphores are always nonnegative). If the trace indicates that v preceded w in the actual execution, but a valid subschedule[4] exists in which w precedes v, then v and w could have executed in either order, i.e., there is a *race condition* between v and w. Netzer and Miller showed that detecting race conditions in parallel programs that use multiple semaphores is NP-complete [15]. Researchers have developed exact algorithms for cases where the problem is efficiently solvable (programs that use types of synchronization weaker than semaphores such as post/wait/clear) [8], [9], [14], and heuristics for the multiple semaphore case [4], [10]. The complexity for the case of a constant number of semaphores was unknown. In the present paper we show that the problem remains NP-complete even if only two semaphores are used in the parallel program.

For the case of using only one semaphore in parallel programs, we give two algorithms. The first algorithm detects in $O(n)$ time whether a race condition exists between any two operations. The second algorithm computes in $O(np \log n)$ time a compact representation, from which one can determine whether a race condition exists between any two operations in $O(1)$ time. Our results are based on reducing the problem of determining whether a valid subschedule exists in which w precedes v to the problem of *Sequencing to Minimize Maximum Cumulative Cost* (*SMMCC*). Given an acyclic directed graph G with costs on the nodes, the *cumulative cost* of the first i nodes in a schedule of G is the sum of the cost of these nodes. Thus, minimizing the maximum cumulative cost is an attempt to ensure that the cumulative cost stays low throughout the schedule. The SMMCC problem is NP-complete in general even if the node costs are restricted to ± 1 [1], [6]. Abdel-Wahab and Kameda [2] presented an $O(n^2)$-time algorithm for the special case that G is a series-parallel graph. (The time bound was later improved to $O(n \log n)$ by the same authors [3].) As part of this solution, they gave an $O(n \log p)$-time algorithm applicable when G consists of p disjoint chains. The existence problem of a valid *schedule* in which v precedes w can be reduced to the SMMCC problem in a chain graph augmented with one interchain edge. We add an edge from w to v, assign costs to the nodes ($+1$ if the node is a P-operation, -1 if a V-operation), and compute the minimum maximum cumulative cost. Clearly, the cost is nonpositive if and only if there is a valid schedule. The augmented chain graph is not series-parallel, so the algorithms of Abdel-Wahab and Kameda [2], [3] are not applicable. We show that the SMMCC problem can nevertheless be solved in polynomial time. In fact, for the special case of interest, that in which the costs are ± 1, we give a linear-time algorithm.

The rest of the paper is organized as follows. Section 2 gives the preliminaries. Section 3 gives the algorithm for a single pair of nodes. Section 4 gives the algorithm for all pairs of nodes. Section 5 sketches the proof for showing that race-condition detection is NP-complete if two semaphores are used in the parallel program.

[4] We consider subschedules rather than schedules because deadlocks might happen during the execution of parallel programs.

2. Preliminaries. Suppose G is an acyclic graph with node costs. We introduce some terminology having to do with schedules, mostly adapted from [2]. A *segment* of a schedule is a consecutive subsequence. Let $H = v_1 v_2 \cdots v_m$ be a sequence of nodes. The *cost* of H, denoted $c(H)$, is the sum of the costs of its nodes. The *height of a node* v_ℓ *in* H is defined to be the sum of the costs of the nodes v_1 through v_ℓ. The *height of* H, denoted $h(H)$, is the maximum of zero and the maximum height of the nodes in H. A node of maximum height in H is called a *peak*. A node of minimum height in H is called a *valley*. The *reverse height* of H, denote $\tilde{h}(H)$, is the height of H minus the cost of H. Note that height and reverse height are nonnegative. A schedule of G is *optimal* if its height is minimum over all schedules of G. We use $h(G)$ to denote the height of its optimal schedule.

A sequence $C = v_1 v_2 \cdots v_m$ of nodes of G is called a *chain* of G if the only edges in G incident on these nodes are $v_0 v_1, v_1 v_2, \ldots, v_{m-1} v_m, v_m v_{m+1}$, where v_0 and v_{m+1} are other nodes, denoted $pred(C)$ and $succ(C)$, respectively. We use $start(C)$ to denote v_1 and $end(C)$ to denote v_m. Note that C could be a single node.

We use $[v, w]_G$ to denote the chain of G starting from v and ending at w. Let $[v, -]_G$ denote the longest chain of G starting from v, and $[-, v]_G$ the longest chain of G ending at v. If it is clear from the context which graph is intended, then we may omit the subscript G. Note that the above notation might not be well defined for any acyclic graph G, but it is so when G is composed of disjoint chains, which is the case of interest in this paper.

Suppose H is a chain of G containing a peak v_ℓ such that (1) every node of H preceding v_ℓ has nonnegative height in H, and (2) every node of H following v_ℓ has height in H at least the cost of H. In this case we call H a *hump*, and we say v_ℓ is a *useful peak* of H. This definition is illustrated in Figure 1. We say a hump is an *N-hump* if its cost is negative, a *P-hump* if its cost is nonnegative.

We are concerned primarily with graphs G consisting of disjoint chains $C_1, C_2, \ldots, C_p$. For convenience, we assume that G contains an *initial pseudonode* ($\perp$), preceding all nodes, and a *terminal pseudonode* ($\top$), following all nodes, each of cost zero. Thus, $pred(v)$ could be $\perp$ and $succ(v)$ could be $\top$.

For the rest of the section we describe the properties of humps in schedules, mostly adapted from [2].

2.1. *Hump Decomposition.* As part of their scheduling algorithm for series-parallel graphs, Abdel-Wahab and Kameda [3] show that in linear time a sequence of nodes

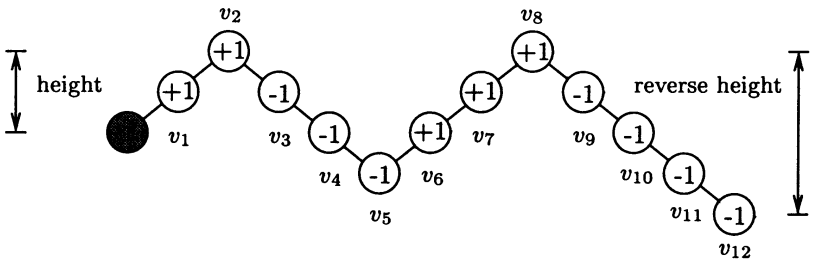


Fig. 1. A hump H of 12 nodes: $v_1, v_2, \ldots, v_{12}$. The cost of each node is in the circle. By definition $c(H) = -2$, $h(H) = 2$, and $\tilde{h}(H) = 4$. Both of v_2 and v_8 are peaks of H, but only v_2 is useful.

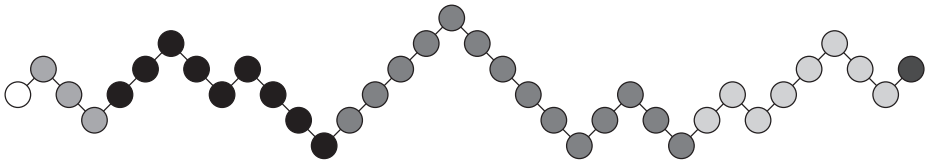


Fig. 2. A chain decomposed into two N-humps and three P-humps.

can be decomposed into a set of humps by an algorithm DECOMP(). It takes a chain as input and outputs a set of disjoint subchains such that every subchain is a hump. The output of DECOMP(C) is unique, although the output is not necessarily the only hump decomposition of C. An example is shown in Figure 2. The chain is decomposed by DECOMP() into two N-humps and three P-humps. For a chain C, we say H is a *hump of* C if $H \in$ DECOMP(C). It can be proved that DECOMP() has the following properties.

HUMP-DECOMPOSITION PROPERTIES.

1. Suppose $H_1, H_2 \in$ DECOMP(C) and H_1 precedes H_2 in C. If $c(H_1) \geq 0$, then $c(H_2) \geq 0$ and $\tilde{h}(H_1) > \tilde{h}(H_2)$. If $c(H_2) < 0$, then $c(H_1) < 0$ and $h(H_1) < h(H_2)$.
2. If v is the first valley of $[u, w]$, then DECOMP($[u, v]$) (respectively, DECOMP($[succ(v), w]$)) consists of N-humps (respectively, P-humps) only.
3. Let C and C' be two disjoint chains, whose humps are respectively $H_1, H_2, \ldots, H_k$ and $H_{k+1}, H_{k+2}, \ldots, H_\ell$ in order. Then, for some $1 \leq i \leq k$ and $k \leq j \leq \ell$, the humps of CC' are

$$H_1, H_2, \ldots, H_i, (H_{i+1} \cdots H_j), H_{j+1}, \ldots, H_\ell$$

in order.

The third property implies that

$$\{end(H):\ H \in \text{DECOMP}(CC')\} \subseteq \{end(H):\ H \in \text{DECOMP}(C)\} \cup \{end(H):\ H \in \text{DECOMP}(C')\}.$$

It will turn out that once we decompose a chain into humps, we need not be concerned with the internal structure of these humps. For each hump H we need only store $c(H)$ and $h(H)$. Thus, a chain consisting of ℓ humps can be represented by a length-ℓ sequence of pairs $(c(H), h(H))$. We call this sequence the *hump representation* of the chain. Using the third hump-decomposition property, one could straightforwardly derive the hump representation of C_1C_2 from the hump representation of C_1 and that of C_2. In particular, if we are given DECOMP(C) and DECOMP(C'), then computing DECOMP(CC') takes $O(|\text{DECOMP}(C)| + |\text{DECOMP}(C')|)$ time.

2.2. *Hump Clustering.* The following lemma concerns an operation on a schedule called *clustering* the nodes of a hump. Suppose H is a hump of G, and let v be a useful peak of H. Let S be a schedule of G. If all the nodes of H are consecutive in S, then we

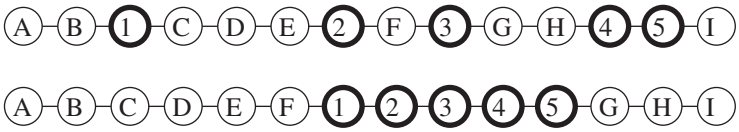


Fig. 3. The second sequence of nodes is obtained from the first one by clustering the nodes 1–5 to node 3.

say H is *clustered in* S. If every hump of G is clustered in S, then we say the schedule S is *clustered*. If a hump is not clustered in a schedule, then we can modify the schedule to make it so. To *cluster the nodes of* H *to* v is to change the positions of nodes of H other than v so that all the nodes of H are consecutive, and the order among nodes of H is unchanged. An example is shown in Figure 3.

LEMMA 2.1 (see [2]). *Let G be an acyclic graph with node costs and let H be a hump of G. Suppose S is a schedule of G. If T is obtained from S by clustering all nodes in H to a useful peak of H, then T is a schedule of G and $h(T) \leq h(S)$.*

An example is shown in Figure 4. The height of the schedule in Figure 4(c) is smaller than that of the schedule in Figure 4(b). Two clustered schedules of the graph in Figure 4(a) are shown in Figure 4(d), (e). It follows from Lemma 2.1 that there is always an optimal schedule of G which is clustered.

2.3. *Standard Order.* A series $S_1 \cdots S_m$ of subsequences of nodes is in *standard order* if it satisfies the following properties.

STANDARD ORDER PROPERTIES.

- The series consists of S_i's with negative costs, followed by S_i's with nonnegative costs.
- The S_i's with negative costs are in nondecreasing order of height; and the S_i's with nonnegative costs are in nonincreasing order of reverse height.

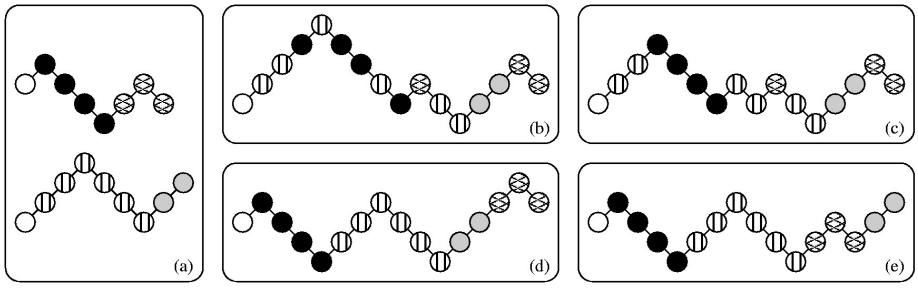


Fig. 4. (a) A graph G consists of two chains. The first chain contains an N-hump followed by a P-hump. The second chain contains two P-humps. (b) A schedule for G of height four. (c) The schedule obtained from the previous one by clustering the N-hump to its useful peak. (d) A clustered schedule of G of height two. This one is obtained from the previous schedule by clustering every hump. (e) A clustered schedule of G with minimum height.

If the humps of a chain are $H_1, H_2, \ldots, H_m$ in order, then the series $H_1 H_2 \cdots H_m$ is in standard order by the first hump-decomposition property.

LEMMA 2.2 (see [2]). *Let A, B, S_1, and S_2 be subsequences of nodes. Suppose $S = S_1ABS_2$ and $T = S_1BAS_2$. If the series BA is in standard order, then $h(S) \geq h(T)$.*

For example, the sequence in Figure 4(d) is a clustered schedule of the graph in Figure 4(a). Note that the series of the last two humps in the schedule is not in standard order: the reverse height of the first hump (zero) is less than that of the second hump (one). The schedule in Figure 4(e) obtained by exchanging those two clustered humps has height one less than that of the schedule in Figure 4(d).

2.4. *Hump Merging.* A schedule of G is in *standard form* if it is clustered and its series of humps of G is in standard order. Let T be any schedule of G in standard form. Recall that by Lemma 2.1 there is always an optimal schedule S of G which is clustered. The humps of G, while clustered in both T and S, may not be in the same order. However, any two humps of the same chain of G must be in the same order in T and in S, else either T or S is not a schedule. Take two consecutive humps in S that are from different chains and that are not in the same order as in T, and exchange their positions. By Lemma 2.2, the resulting ordering has height no more than S. By a series of such exchanges, we eventually obtain T from S. It follows that the height of T is no more than that of S, and hence that T is optimal. This argument shows that every schedule in standard form is an optimal schedule of G.

Let $I = \{H_1, H_2, \ldots, H_m\}$, where the series $H_1 H_2 \cdots H_m$ is in standard order. Suppose MERGE(I) returns a sequence of nodes obtained by concatenating all humps in I into standard order. Namely, MERGE(I) $= H_1 H_2 \cdots H_m$. Assume for uniqueness that MERGE() breaks ties in some arbitrary but fixed way. By the above argument we have the following lemma.

LEMMA 2.3 (see [2]). *The output of*

$$\text{MERGE}\left(\bigcup_{1 \leq i \leq p} \text{DECOMP}(C_i)\right)$$

is an optimal schedule of G.

An example is shown in Figure 4. Since the schedule in Figure 4(e) is clustered and its series of humps is in standard order, it is an optimal schedule of the graph in Figure 4(a). Abdel-Wahab and Kameda [2] showed that MERGE($\bigcup_{1 \leq i \leq p}$ DECOMP(C_i)) can be obtained in $O(n \log p)$ time. Note that the output of function MERGE() may not be unique. Without loss of generality, however, we may define MERGE() more restrictively as follows to make its output unique for the same G. Suppose G is composed of disjoint chains, $C_1, C_2, \ldots, C_p$ and $I = \bigcup_{1 \leq i \leq p}$ DECOMP(C_i). Define MERGE(I) $= H_1 H_2 \cdots H_m$, where $\{H_1, H_2, \ldots, H_m\} = I$ and the series $H_1 H_2 \cdots H_m$ is in standard order. Furthermore, if $H_i H_j$ and $H_j H_i$ are both in standard order, where $C_{i'}$ contains H_i, $C_{j'}$ contains H_j, and $i' < j'$, then H_i precedes H_j in MERGE(I).

3. Algorithm for Single Pair. A vector $\Gamma = (x_1, x_2, \ldots, x_p)$ of p nodes is called a *cut* of G if each x_i is either $\perp$ or a node in C_i. We call x_i the ith *cutpoint* of Γ. The *prefix subgraph* $G[\Gamma]$ of G is the subgraph $\bigcup_{1 \le i \le p}[-, x_i]$. Therefore, the problem we address can be reduced to finding a cut such that the valid schedule of the prefix subgraph determined by the cut has the minimal maximum cumulative cost. Let h be the maximum cumulative cost of the optimal subschedule that contains v and w. If h is zero, then a valid subschedule exists (i.e., the optimal valid subschedule.) If h is positive, then there is no valid subschedule because the maximum cumulative cost of any valid subschedule is greater than or equal to h and is thus positive, too. The rest of the section shows that a best cut can be found in linear time.

Since we will frequently encounter two cuts that differ at only one cutpoint, let NEWCUT(Γ, i, u) denote a cut Γ' with

$$\Gamma'(\ell) = \begin{cases} \Gamma(\ell) & \text{if} \quad \ell \neq i, \\ u & \text{if} \quad \ell = i. \end{cases}$$

A *j-schedule* of $G[\Gamma]$ is a schedule of $G[\Gamma]$ whose last node is $\Gamma(j)$. We use $h_j(G[\Gamma])$ to denote the height of an optimal j-schedule of $G[\Gamma]$. Suppose $\Gamma(j) \neq \perp$. One can compute $h_j(G[\Gamma])$ for a given Γ as follows. Let $\Gamma' =$ NEWCUT$(\Gamma, j, pred(\Gamma(j)))$. Clearly, if S is an optimal schedule of $G[\Gamma']$, then $S\Gamma(j)$ is an optimal j-schedule of $G[\Gamma]$. It follows that

$$h_j(G[\Gamma]) = \max\{h(G[\Gamma']), c(G[\Gamma']) + h(\Gamma(j))\}.$$

Note that $h(G[\Gamma])$ and $h_j(G[\Gamma])$ are both nonnegative. We use $v \to w$ to signify that there is a valid subschedule of G in which v precedes w. Let $v \not\to w$ signify that $v \to w$ is not true. Note that neither $\to$ nor $\not\to$ is a partial order.

3.1. *Basic Idea.* Every valid subschedule of G is a valid schedule of a prefix subgraph $G[\Gamma]$ for some cut Γ of G. Therefore, $v \to w$ if and only if there is a cut Γ of G such that $G[\Gamma]$ has a valid schedule in which v precedes w. Let h^* be the minimum of $h(G[\Gamma] \cup \{vw\})$ over all $G[\Gamma]$'s that contain v and w. It follows that $v \to w$ if and only if $h^* = 0$. Hence, the problem of determining whether $v \to w$ is reduced to computing the minimum height of a set of chain graphs each augmented with an interchain arc. Clearly, two immediate questions arise. (1) How do we compute the height of $G[\Gamma] \cup \{vw\}$, which is not even serial-parallel? (2) How do we cope with the fact that there could be an exponential number of prefix subgraphs that contain v and w?

Let v and w be contained in two disjoint chains C_i and C_j, respectively. The following observation will ease the situation. Suppose S is a subschedule of G containing w. Let S' be the subschedule of G obtained from S by discarding all nodes succeeding w in S. Clearly, $h(S') \le h(S)$. Therefore, without loss of generality the minimum of $h(G[\Gamma] \cup \{vw\})$ can be computed over only cuts Γ with $\Gamma(j) = w$. Moreover, we can let w always be the last node of a subschedule by considering only the minimum-height j-schedule of each $G[\Gamma]$ that contains v. The first question above is no longer an issue.

It turns out that the second question is not an issue, either. We will show that in order to obtain the minimum-height of all those j-schedules, it suffices to consider only $O(\sqrt{n})$ cuts. In particular each of those $O(\sqrt{n})$ cuts is uniquely determined by its jth cutpoint.

```
Function MinHeight(v, w)
1  C_i  := the chain containing v;
2  C_j  := the chain containing w;
3  Γ(j) := w;
4  h*   := ∞;
5  I_0  := {v} ∪ Decomp([succ(v), −]);
6  For every Γ(i) ∈ {end(H) : H ∈ I_0} do
7     S*:= Best(j, {i, j}, Γ);
8     h*:= min{h*, h(S*)};
9  Return h*;
```

```
Function Best(j, F, Γ)
1  I   := ⋃_{k∈F,k≠j} Decomp([−, Γ(k)]);
2  J   := Decomp([−, pred(Γ(j))]);
3  K   := ⋃_{k∉F} Decomp(C_k);
4  s_1 := max{h(H) : H ∈ I ∪ J, c(H) < 0};
5  S+  := Merge({H ∈ I ∪ J : c(H) ≥ 0});
6  s_2 := h(S+Γ(j));
7  s   := max{s_1, s_2};
8  K_s := {H ∈ K : h(H) < s, c(H) < 0};
9  S_s := Merge(I ∪ J ∪ K_s);
10 Return S_sΓ(j);
```

Fig. 5. The algorithm for computing $h(G[\Gamma^*] \cup \{vw\})$ for a best cut Γ^* of G corresponding to vw.

3.2. *The Algorithm.* The algorithm takes v and w as inputs. Let C_i contain v and C_j contain w. The algorithm proceeds iteratively with different cutpoint $\Gamma(i)$ such that $\Gamma(i)$ does not precede v. In each iteration the algorithm calls the function BEST() to obtain a minimum-height j-schedule for $G[\Gamma]$ over all cuts Γ with the designated cutpoints in C_i and C_j. By comparing the heights of these j-schedules with respect to different $\Gamma(i)$'s, the algorithm outputs the minimum height of j-schedules for $G[\Gamma]$ over all Γ such that $\Gamma(j) = w$ and $\Gamma(i)$ does not precede v. In Figure 5 we give the algorithm to compute $h(G[\Gamma^*] \cup \{vw\})$, where Γ^* is a best cut of G corresponding to vw.

Function BEST() is the essential part of the algorithm. Based on the given subset F of $\{1, 2, \ldots, p\}$ and the given cut Γ, it looks for a best cut Γ^* corresponding to vw such that $\Gamma^*(k) = \Gamma(k)$ for every $k \in F$. (In the case in which we are interested, $F = \{i, j\}$.) An optimal j-schedule of $G[\Gamma^*]$ is then returned. Note that for every $k \notin F$, $\Gamma^*(k)$ depends on a value s, which is the maximum of s_1 and s_2. Each of s_1 and s_2 is determined simply by chains with indices in F and their designated cutpoints. Namely, the choices of $\Gamma^*(k)$'s for different $k \notin F$ are mutually independent. This is the key to our efficient algorithm.

In BEST(), we do not explicitly specify cutpoints of Γ^*. Instead, we work on the hump representation of subchains and every cutpoint is implicitly specified by an $end(H)$ for some hump H. Specifically, Step 1 ensures $\Gamma^*(k) = \Gamma(k)$ for every $k \in F, k \neq j$. Steps 3 and 8 ensure $\Gamma^*(k) = end(H)$, where H is the highest N-hump of all C_k with $h(H) < s$ and $k \notin F$. Since we are considering j-schedules, $\Gamma^*(j)$ is specified slightly differently. Although in Step 2 the subchain of C_j is only up to $pred(\Gamma(j))$, $\Gamma^*(j)$ is still $\Gamma(j)$, since j-schedule $S^*\Gamma(j)$ is returned in Step 10.

3.3. *Correctness.* We answer the following two questions in this subsection:

1. Why is it sufficient to try for $\Gamma(i)$ only those nodes in $\{end(H)\colon\ H \in I_0\}$?
2. Why does BEST(j, F, Γ) return an optimal j-schedule of $G[\Gamma^*]$ with $\Gamma^*(k) = \Gamma(k)$ for every $k \in F$?

LEMMA 3.1. *Let Γ be a cut of G. Suppose $[x, z]$ is a subchain of G containing $\Gamma(i)$. Let H be the hump of $[x, z]$ containing $\Gamma(i)$. Let y be the first valley of $[pred(H)$,*

$\Gamma(i)]$. *If*

$$\Gamma_1(k) = \begin{cases} \Gamma(k) & \text{if} \quad k \neq i, \\ pred(H) & \text{if} \quad k = i \quad \text{and} \quad y = pred(H), \\ end(H) & \text{if} \quad k = i \quad \text{and} \quad y \neq pred(H), \end{cases}$$

then $h_j(G[\Gamma_1]) \leq h_j(G[\Gamma])$.

PROOF. Straightforward. □

Note that the $pred(H)$ in the above lemma is always an $end(H')$ for some hump H' in I_0, which is defined in Step 5 of MINHEIGHT(). Therefore, Lemma 3.1 answers the first question.

By definitions of I, J, and K_s it is not difficult to see that the sequence returned by BEST(j, F, Γ) is an optimal j-schedule of $G[\Gamma^*]$ for some cut Γ^* such that $\Gamma^*(k) = \Gamma(k)$ for every $k \in F$. The correctness of MINHEIGHT() thus relies on the following lemma, which answers the second question.

LEMMA 3.2. *Let* Γ *be a cut. Let* F *be a subset of* $\{1, 2, \ldots, p\}$ *containing* j. *If* $S^* =$ BEST(j, F, Γ), *then* $h(S^*) \leq h_j(G[\Gamma])$.

The rest of the subsection proves Lemma 3.2. Let $F_\ell = \{1, \ldots, \ell - 1, \ell + 1, \ldots, p\}$. The following lemma is a special case of Lemma 3.2, in which F is composed of $p - 1$ numbers.

LEMMA 3.3. *Let* Γ *be a cut. If* $S^* =$ BEST(j, F_ℓ, Γ) *for some* $\ell \neq j$, *then* $h(S^*) \leq h_j(G[\Gamma])$.

PROOF. Define Γ_1 by

$$\Gamma_1(k) = \begin{cases} \Gamma(k) & \text{if} \quad k \neq \ell, \\ \text{the first valley of } [-, \Gamma(\ell)] & \text{if} \quad k = \ell. \end{cases}$$

Then it is not difficult to see $h_j(G[\Gamma_1]) \leq h_j(G[\Gamma])$. Let Γ' be the cut with $h(S^*) = h_j(G[\Gamma'])$, i.e., S^* is a j-schedule of $G[\Gamma']$. By definition of BEST(), Γ' and Γ_1 could differ only at the ℓth position. Clearly, it suffices to show $h_j(G[\Gamma']) \leq h_j(G[\Gamma_1])$.

Let $w = \Gamma_1(j)$. Let $L =$ DECOMP($[-, \Gamma_1(k)]$). Define

$$S = \text{MERGE}(I \cup J \cup L),$$

where I and J are defined in Steps 1 and 2 of BEST(). Clearly, Sw is an optimal j-schedule of $G[\Gamma_1]$. Thus, $h(Sw) = h_j(G[\Gamma_1])$. By choice of $\Gamma_1(\ell)$, L contains no P-hump. Hence, by the uniqueness assumption of MERGE(), we could write $Sw = S_1 S^+ w$, where S^+ is defined in Step 5 of BEST(). We prove $h_j(G[\Gamma']) \leq h_j(G[\Gamma_1])$ by showing that $\Gamma'(\ell)$ succeeds $\Gamma(\ell)$ if and only if $h_j(G[\Gamma']) \leq h(Sw)$ as follows.

Case 1: $\Gamma'(\ell)$ *succeeds* $\Gamma(\ell)$. Since L contains no P-hump, each hump of $[-, \Gamma_1(\ell)]$ appears in S_1. Therefore, $S_1 S' S^+ w$ is a j-schedule of $G[\Gamma']$, where $S' = [succ(\Gamma_1(\ell))$,

$\Gamma'(\ell)]$. We show $h(S_1S'S^+w) \leq h(S_1S^+w)$. Now $h(S_1S'S^+w) = \max\{h(S_1), c(S_1) + h(S'), c(S_1S') + h(S^+w)\}$. Clearly,

$$h(S_1) \leq h(S_1S^+w). \tag{1}$$

By definition of F, the K_s defined in Step 8 of BEST() is composed of the N-humps of C_ℓ that have heights less than s. Therefore, by choice of $\Gamma'(\ell)$ every hump of $[-, \Gamma'(\ell)]$ has height less than s. It follows from the standard order of humps in S' that $h(S') < s$. By Step 7 of BEST(), $s = \max\{s_1, s_2\}$. If $s = s_2 = h(S^+w)$, as defined in Step 6 of BEST(), then $c(S_1) + h(S') < c(S_1) + h(S^+w)$. If $s = s_1 = h(H^*)$, where H^* is a highest N-hump in $I \cup J$, then we could write $S_1 = S_2H^*S_3$. It follows that

$$\begin{aligned} c(S_1) + h(S') &= c(S_2H^*S_3) + h(S') \\ &< c(S_2) + h(H^*) \\ &\leq h(S_2H^*) \\ &\leq h(S_1). \end{aligned}$$

Therefore, in either case we have

$$c(S_1) + h(S') < h(S_1S^+w). \tag{2}$$

By choice of $\Gamma'(\ell)$, $c(S') < 0$. Hence,

$$\begin{aligned} c(S_1S') + h(S^+w) &< c(S_1) + h(S^+w) \\ &\leq h(S_1S^+w). \end{aligned} \tag{3}$$

Combining (1), (2), and (3), we obtain $h(S_1S'S^+w) \leq h(Sw)$.

Case 2: $\Gamma'(\ell)$ *precedes* $\Gamma_1(\ell)$. Let $S' = [succ(\Gamma'(\ell)), \Gamma_1(\ell)]$. By choice of $\Gamma'(\ell)$, it is not difficult to see

$$\text{DECOMP}([-, \Gamma_1(\ell)]) = \text{DECOMP}([-, \Gamma'(\ell)]) \cup \text{DECOMP}(S').$$

By choice of $\Gamma_1(\ell)$, DECOMP(S') contains only N-humps of heights no less than s. Note that every N-hump in $I \cup J$ has height no more than s. By the standard form of S, we know that S' is a suffix of S_1. Therefore, we could write $Sw = S_2S'S^+w$. Removing S' from Sw, we obtain a j-schedule S_2S^+w of $G[\Gamma']$. We show $h(S_2S^+w) \leq h(Sw)$.

Now $h(S_2S^+w) = \max\{h(S_2), c(S_2) + h(S^+w)\}$. Clearly,

$$h(S_2) \leq h(S_2S'S^+w) = h(Sw). \tag{4}$$

Since each hump of S' has height no less than s, $h(S') \geq s$. Hence, $h(S'S^+w) \geq h(S') \geq s \geq s_2 = h(S^+w)$. It follows that

$$\begin{aligned} c(S_1) + h(S^+w) &\leq c(S_1) + h(S'S^+w) \\ &\leq h(Sw). \end{aligned} \tag{5}$$

Combining (4) and (5), we obtain $h(S_1S^+w) \leq h(Sw)$. □

```
Procedure CUTTRANS(Γ, Γ*)
1  ℓ := 0;
2  While Γ* ≠ Γ do
3     ℓ := (ℓ mod p) + 1;
4     If ℓ ∉ F
5        S := BEST(j, F_ℓ, Γ);
6        Γ′ := the cut such that S is an optimal j-schedule of G[Γ′];
7        Γ := Γ′;
```

Fig. 6. The algorithm transforms Γ to Γ^*. We prove Lemma 3.2 by showing that this algorithm always terminates.

Now we are ready to prove Lemma 3.2.

PROOF OF LEMMA 3.2. Recall that $S^* = \text{BEST}(j, F, \Gamma)$. Let Γ' be the cut such that S^* is a j-schedule of $G[\Gamma']$. (S^* is certainly an optimal j-schedule of $G[\Gamma']$.) We use the algorithm in Figure 6 to prove the lemma. Procedure CUTTRANS() proceeds with iterations, in which the value of ℓ varies among $\{1, \ldots, p\}$. If $\ell \notin F$, then the value of $\Gamma(\ell)$ is updated. Since S is an optimal j-schedule of $G[\Gamma']$, it follows from Lemma 3.3 that $h_j(G[\Gamma']) \leq h_j(G[\Gamma])$ always holds during the while-loop. If we could show that CUTTRANS() always terminates, then the lemma is proved.

Let s_1^*, s_2^*, and s^* be the s_1, s_2, and s in the execution of $\text{BEST}(j, F, \Gamma)$. Let s_1, s_2, and s be those in the execution of $\text{BEST}(j, F_\ell, \Gamma)$. The values of s_1, s_2, and s change as the while-loop of CUTTRANS() proceeds. We show that Γ eventually becomes Γ' by arguing that s eventually becomes s^*.

Since $F \subseteq F_\ell$, $s_1 \geq s_1^*$ always holds. By definition of BEST(), whenever Step 7 of CUTTRANS() is finished, $[-, \Gamma(\ell)]$ contains only N-humps. Thus, after the first p iterations of the while-loop, $[-, \Gamma(\ell)]$ contains no P-hump for every $\ell \notin F$. Henceforth, $s_2 = s_2^*$ and therefore $s = \max\{s_1, s_2\} \geq \max\{s_1^*, s_2^*\} = s^*$. If $s > s^*$, then $s = s_1 > s^*$. Since $s_1 > s^*$, there must be an N-hump H in $\bigcup_{k \notin F} \text{DECOMP}([-, \Gamma(k)])$ such that $h(H) = s_1$. Since $s = s_1$, in the next iteration when C_ℓ contains H, $\Gamma(\ell)$ will be moved before H by definition of BEST(). It follows that the value of s is nonincreasing and s will become s^*. Once $s = s^*$, in the following p iterations, $\Gamma(k)$ will be moved to $\Gamma'(k)$ for every $k \notin F$. The algorithm then terminates. □

3.4. *Implementation.* Recall that $\text{DECOMP}(C)$ runs in time linear in $|C|$, the length of chain C. It follows that the time complexity of Steps 1–5 and Step 9 of MINHEIGHT() is $O(n)$. Suppose the order of nodes assigned to $\Gamma(i)$ in the for-loop is the same as their order in C_i. In the subsection we focus on implementing BEST() such that the for-loop runs in time $O(n)$.

Number of Iterations. The following lemma ensures that the size of I_0 is $O(\sqrt{|C_i|})$. It follows that the number of iterations is $O(\sqrt{n})$.

LEMMA 3.4. *Suppose C is a chain with node costs ± 1. The number of humps in* DECOMP(C) *is* $O(\sqrt{|C|})$.

PROOF. Since the costs of nodes are either $+1$ or -1, a hump of height ℓ contains at least ℓ nodes. For the same reason, a hump of reverse height ℓ contains at least ℓ nodes. By the first hump decomposition property, the heights of the N-humps in DECOMP(C) are different, and so are the reverse heights of the P-humps in DECOMP(C). If there are n_1 N-humps and n_2 P-humps in DECOMP(C), then $|C| = \Omega(n_1^2 + n_2^2) = \Theta((n_1 + n_2)^2)$. This proves the lemma. □

Compact Representation of Humps. For the sake of efficiency, we do not deal with the internal structure of humps in BEST(). It suffices to represent each hump H by a pair $(c(H), h(H))$ and work on the compact representation of humps. Therefore, each of the I, J, and K computed in the first three steps is a set of pairs. Clearly, each of these three steps takes $O(n)$ time. However, the contents of J and K do not change in different iterations. Thus, Steps 2 and 3 need only be executed once.

By $F = \{i, j\}$, we have $I =$ DECOMP($[-, \Gamma(i)]$). Suppose I_t and Γ_t are the I and Γ in the tth iteration for some $t \geq 2$. By the order of nodes assigned to $\Gamma(i)$, we need not recompute DECOMP($[-, \Gamma_t(i)]$) from scratch. In the tth execution of Step 1, $[-, \Gamma_t(i)]$ is obtained by appending a hump $[succ(\Gamma_{t-1}(i)), \Gamma_t(i)]$ to $[-, \Gamma_{t-1}(i)]$. By the argument following the hump decomposition properties in Section 2.1, the tth execution of Step 1 takes $O(|I_{t-1}|)$ time. By Lemma 3.4, the time complexity of all executions of Step 1 is $O(n + \sqrt{n} \times \sqrt{n}) = O(n)$.

Priority Tree. To compute s_1 efficiently, we resort to a *priority tree*, a complete binary tree with $n + 1$ leaves.[5] Each leaf keeps two values, *count* and *maxheight*. The cost of the $(h + 1)$st leaf is the number of N-humps of height h in $I \cup J$. The maxheight of the $(h + 1)$st leaf is zero (respectively, h), if its count is zero (respectively, nonzero). The maxheight of an internal node is the maximum maxheight of its children. It follows that the maxheight of the root of a priority tree is the correct value of s. The priority tree can be built in time $O(n)$. Whenever a hump is added to or deleted from $I \cup J$, the priority tree can be updated in time $O(\log n)$. Since J is fixed, to compute s_1 in tth iterations for every $t \geq 2$, we add humps in $I_t - I_{t-1}$ to $I \cup J$, remove humps in $I_{t-1} - I_t$ from $I \cup J$, and update the priority tree. By the third hump decomposition property, we have

$$\sum_{2 \leq t \leq q_i} |I_t - I_{t-1}| + |I_{t-1} - I_t| = O(\sqrt{|C_i|}), \tag{6}$$

where q_i is the number of humps in C_i. Hence, the time complexity of all executions of Step 4 is $O(n + \sqrt{n} \times \log n) = O(n)$.

Hump Tree. To obtain the value of s_2, it is not necessary to know the value of S^+. We need only to obtain the height of $S^+\Gamma(j)$. Similarly, the actual value of S_s is irrelevant. What we compare in Step 8 of MINHEIGHT() is the height of $S_s\Gamma(j)$. We need a data structure to compute these two heights efficiently.

[5] Note that there are other ways to implement Step 4 to run in linear time. However, the necessity of a priority tree will become clear when we address the implementation of the all-pairs algorithm.

Let L be a set of humps such that $h(H) \leq n$ and $\tilde{h}(H) \leq n$ for every $H \in L$. A *hump tree* T for L is a binary tree composed of two complete binary subtrees. Each subtree has $n+1$ leaves. Let T_N be the left subtree and T_P be the right subtree. The $(h+1)$st leaf of T_N associates with the set of N-humps of height h in L. The $(h+1)$st leaf of T_P associates with the set of P-humps of reverse height $n-h$ in L. Let T_x be the subtree of T rooted at x. Let L_x be the set of humps associated with leaves of T_x. Define $h(T_x) = h(\text{MERGE}(L_x))$ and $c(T_x) = c(\text{MERGE}(L_x))$. Clearly, when $L = I \cup J$, $h(T_P) = h(S^+)$ and $c(T_P) = c(S^+)$. When $L = I \cup J \cup K_s$, $h(T) = h(S_s)$ and $c(T) = c(S_s)$. The heights of $S^+\Gamma(j)$ and $S_s\Gamma(j)$ can then be computed by

$$\begin{aligned} h(S^+\Gamma(j)) &= \max\{h(S^+), c(S^+) + h(\Gamma(j))\}, \\ h(S_s\Gamma(j)) &= \max\{h(S_s), c(S_s) + h(\Gamma(j))\}. \end{aligned}$$

We keep $h(T_x)$ and $c(T_x)$ in x for every node x of T. Therefore, the hump tree T takes $O(n)$ space. We show how to compute $h(T_x)$ and $c(T_x)$ for every node x from leaves to root. When x is a leaf of T, the humps in L_x have the same height if x is in T_N, and the same reverse height if x is in T_P. It is not difficult to see that $c(T_x) = \sum_{H \in L_x} c(H)$; and

$$h(T_x) = \begin{cases} 0 & \text{if} \quad L_x = \emptyset, \\ h & \text{if} \quad x \text{ is the } (h+1)\text{st leaf of } T_N, \\ c(T_x) - h & \text{if} \quad x \text{ is the } (n-h+1)\text{st leaf of } T_P. \end{cases}$$

When x is an internal node of T, $h(T_x)$ and $c(T_x)$ can be computed by the information kept in the children of x. Suppose y and z are the left and right children of x, respectively. For any H in L_y and H' in L_z, by the way we associate humps with leaves, the series HH' is in standard order. Hence,

$$\begin{aligned} h(T_x) &= \max\{h(T_y), c(T_y) + h(T_z)\}, \\ c(T_x) &= c(T_y) + c(T_z). \end{aligned}$$

It follows that the hump tree T for L can be built in time $O(n + |L|)$.

Once T is built, inserting a hump to L can be done efficiently. Suppose we insert H to L. For the case that H is an N-hump, if $L_x = \emptyset$, then let $h(T_x) = h$; otherwise, add $c(H)$ to $c(T_x)$, where x is the $(h(H)+1)$st leaf of T_N. If H is a P-hump, then we add $c(H)$ to both $c(T_x)$ and $h(T_x)$, where x is the $(n - \tilde{h}(H) + 1)$st leaf of T_P. To update T, we simply update the internal nodes on the path from x to the root of T. Deleting a hump from L can be done similarly by replacing every addition with a subtraction. Clearly, both insertion and deletion take time $O(\log n)$.

To compute the heights of $S^+\Gamma(j)$ and $S_s\Gamma(j)$, we need not maintain a hump tree for $I \cup J$ and another hump tree for $I \cup J \cup K_s$. Suppose K^- is the set of N-humps in K, i.e., $K^- = \{H \in K\colon c(H) < 0\}$. It suffices to maintain a hump tree T for $I \cup J \cup K^-$. Since there is no P-hump in K^-, it is still true that $h(T_p) = h(S^+)$ and $c(T_p) = c(S^+)$. Although the hump tree is not for $I \cup J \cup K_s$, the values of $h(S_s)$ and $c(S_s)$ can be efficiently obtained by the procedure in Figure 7. Procedure REMOVERANGE() acts as if the N-humps of heights no less than s are removed from the hump tree for $I \cup J \cup K^-$. Therefore, the resulting $h(T)$ and $c(T)$ are $h(S_s)$ and $c(S_s)$, respectively.

```
Procedure REMOVERANGE(T, s)
1  y := the sth leaf of T_N;
2  While y is not the root of T_N do
3     x := the parent of y;
4     If y is the left child of x then
5        (h(T_x), c(T_x)) := (h(T_y), c(T_y));
6     else
7        Recompute h(T_x) and c(T_x);
8     y := x;
9  Recompute h(T) and c(T);
```

Fig. 7. Let T be the hump tree for $I \cup J \cup K^-$. This procedure acts as if the N-humps of heights no less than s are removed from the hump tree.

Clearly, REMOVERANGE() takes $O(\log n)$ time. Since we maintain the hump tree for $I \cup J \cup K^-$ in every iteration, we use $O(\log n)$ space to keep the modified information of T. After obtaining the information we need, we restore the hump tree for $I \cup J \cup K^-$ in time $O(\log n)$.

Let I_t be the I in the tth iteration for any $t \geq 1$. To obtain the hump tree for $I_t \cup J \cup K^-$ from $I_{t-1} \cup J \cup K^-$, we need to insert the humps in $I_t - I_{t-1}$ to T and remove the humps in $I_{t-1} - I_t$ from T. Since each insertion and deletion takes $O(\log n)$ time, it follows from (6) that the overall time complexity for obtaining the hump tree from that of previous iterations is $O(\sqrt{n} \times \log n)$. Recall that building a hump tree for L takes $O(n + |L|)$ time. Since there are n nodes in G, $|I_1 \cup J \cup K^-| = O(n)$. It follows that the time complexity for building a hump tree for $I_1 \cup J \cup K^-$ is $O(n)$.

By the above arguments we implement BEST() such that the overall time complexity of the while-loop in MINHEIGHT() is $O(n)$. We therefore have the following theorem.

THEOREM 3.5. *Suppose G is a graph consisting of p disjoint chains comprising n nodes, where each node represents either a P-operation or a V-operation. For any two nodes v and w of G, one can determine in $O(n)$ time whether there is a valid subschedule in which v precedes w.*

4. Algorithm for All Pairs. In this section we show how to determine the $\rightarrow$ relations for all pairs of nodes in G. The linear-time algorithm for a single pair of nodes, applied to all $O(n^2)$ pairs, takes time $O(n^3)$. Fortunately, there is a *compact representation* of this information. To represent this information, it is sufficient that we indicate, for each node v, and for each chain C not containing v, the first node w in C such that v precedes w in some valid subschedules. This representation has size $O(np)$, where n is the number of nodes and p is the number of chains. The representation can be used to determine in constant time whether there is a race between two given operations

```
Procedure CHAINPAIR(i, j)
1  (v, w) := (end(C_i), end(C_j));
2  Repeat
3     If w = ⊥ then h := 1;
4                else h := MINHEIGHT(v, w);
5     If h > 0  then first_j(v) := succ_j(w);
6                     v := pred(v);
7                else w := pred(w);
8  Until v = ⊥;
```

Fig. 8. The algorithm that computes $first_j(v)$ for every $v \in C_i$.

v and w, assuming that the input p chains are schedulable.[6] To determine whether v can precede w, we obtain the first node in w's chain that could be preceded by v in some valid subschedules. If this first node is numbered later than w, then v can precede w. Otherwise, v cannot precede w. We therefore consider the complexity of constructing such a representation. Clearly, it can be constructed by a sequence of calls to the algorithm of Theorem 3.5. We show how to do much better; in fact the time required by our algorithm is only $O(\log n)$ times the time required simply to write down the output.

4.1. *The Algorithm.* Let $first_j(v)$ denote the first node in C_j that could be preceded by v in some valid subschedule of G. The output of the all-pairs algorithm is thus the value of $first_j(v)$ for every node v and $1 \leq j \leq p$. Note that $first_j(v)$ could be $\top$, which means that none of nodes in C_j can be preceded by v in any valid subschedule of G.

We describe first the procedure CHAINPAIR(i, j) which computes $first_j(v)$ for every $v \in C_i$. The all-pairs algorithm simply calls CHAINPAIR(i, j) for every $1 \leq i, j \leq p$. For convenience, let $succ_j(w) = succ(w)$ for every $w \in C_j$ and let $succ_j(\perp) = start(C_j)$. Procedure CHAINPAIR$(i, j)$ is shown in Figure 8. The algorithm starts with letting v be $end(C_i)$ and letting w be $end(C_j)$. The repeat-loop proceeds by replacing w with $pred(w)$. Once MINHEIGHT(v, w) is not zero, the algorithm reports $succ_j(w)$ as $first_j(v)$. After replacing v with $pred(v)$, the repeat-loop continues the same procedure to search for new $first_j(w)$.

4.2. *Correctness.* By induction on v we show that CHAINPAIR(i, j) correctly computes $first_j(v)$ for every $v \in C_i$.

When $v = end(C_i)$, procedure CHAINPAIR(i, j) keeps replacing w with $pred_j(w)$ until $w = \perp$ or MINHEIGHT$(v, w) > 0$. If $w = \perp$, then $h(G \cup \{vw'\}) = 0$ for every $w' \in C_j$. Thus, $first_j(v) = succ_j(\perp) = succ_j(w) = start(C_j)$ is correct. If

[6] Since the p chains represent a trace of a parallel program, the assumption holds. For arbitrary p chains, one can determine whether they are schedulable using the algorithm in [2].

MINHEIGHT$(v, w) > 0$, then $v \not\to w$. It follows that $v \not\to w'$ for every w' precedes w in C_j. Since MINHEIGHT$(v, succ_j(w)) = 0$, $v \to succ_j(w)$. Therefore, $succ_j(w)$ is the correct value of $first_j(v)$. This confirms the induction basis.

Suppose the procedure CHAINPAIR(i, j) correctly reports $succ_j(w)$ as the value of $first_j(succ_i(v))$ in a certain iteration of the repeat-loop. We need to show that in the remaining iterations $first_j(v)$ will also be correctly computed. Since $succ_i(v) \to succ_j(w)$, $v \to succ_j(w)$. It follows that $v \to w'$ (and thus MINHEIGHT$(v, w') = 0$) for every w' succeeding w in C_j. In other words, to locate the first node in C_j that could be preceded by v, it suffices to start testing from w. For the same reason as above, CHAINPAIR(i, j) reports the correct value of $first_j(w)$. The correctness is therefore ensured.

4.3. *Implementation.* We show in this subsection how to implement CHAINPAIR(i, j) to run in time $O((|C_i| + |C_j|) \log n)$. It then follows that the time complexity of the all-pairs algorithm is $O(np \log n)$.

Suppose each time before we call CHAINPAIR(i, j), we have the hump tree for $I \cup J \cup K^-$, where

$$I = \text{DECOMP}(C_i),$$

$$J = \text{DECOMP}([-, pred(end(C_j))]),$$

$$K^- = \left\{ H \in \bigcup_{\substack{1 \le k \le p \\ k \ne i, j}} \text{DECOMP}(C_k) \colon c(H) < 0 \right\}.$$

It follows from Section 3.4 that the first call to MINHEIGHT(v, w) can be computed in time $O(\log n)$, since only one $\Gamma(i)$ need be considered. In each of the remaining iterations of the repeat-loop, we either replace v with $pred(v)$ or replace w with $pred(w)$. The remaining lemma guarantees that to compute each of the following MINHEIGHT(v, w), we need only try v as the cutpoint of C_i.

LEMMA 4.1. *Consider any iteration of the repeat-loop in* CHAINPAIR(i, j). *When the algorithm computes* $h =$ MINHEIGHT(v, w), v *is the only cutpoint of* C_i *that could make* h *zero.*

PROOF. By definition of CHAINPAIR(), when computing MINHEIGHT(v, w), $first_j(succ_i(v))$ always succeeds w in C_j. Assume for a contradiction that u is a node succeeding v in C_i such that there is a cut Γ of G where $\Gamma(i) = u$, $\Gamma(j) = w$, and $h_j(G[\Gamma]) = 0$. It follows that $u \to w$ and thus $succ_i(v) \to w$. This contradicts the fact that $first_j(succ_i(v))$ succeeds w in C_j. □

THEOREM 4.2. *Suppose* G *is as in Theorem* 3.5. *The compact representation of the relation "*v *precedes* w *in some valid subschedules" can be constructed in* $O(np \log n)$ *time and* $O(n)$ *space.*

PROOF. Note that in each iteration of the repeat-loop, either v or w is moved by one position. Since the costs of v and w are ± 1, by the first hump decomposition property the

number of humps updated in $I \cup J \cup K^-$ between two consecutive iterations is a constant. Thus, each execution of MINHEIGHT(v, w) takes only time $O(\log n)$. Since the number of iterations of the repeat-loop is $O(|C_i| + |C_j|)$, each execution of CHAINPAIR(i, j) takes time

$$O((|C_i| + |C_j|) \times \log n). \tag{7}$$

It remains to show how to build the hump tree efficiently for each execution of CHAINPAIR(i, j).

The very first hump tree can be constructed in time

$$O(n). \tag{8}$$

Consider the moment when CHAINPAIR(i, j) is just finished and the all-pairs algorithm is about to call CHAINPAIR(i_1, j_1). Since all humps in $I \cup J$ have been deleted during the execution of CHAINPAIR(i, j), the current T is the hump tree for the N-humps in $\bigcup_{1 \le k \le p; k \ne i, j}$ DECOMP(C_k). In order to obtain the hump tree for CHAINPAIR(i_1, j_1), we have to add the N-humps in DECOMP$(C_i) \cup$ DECOMP(C_j), delete the N-humps in DECOMP(C_{j_1}) from T, and then insert the humps in

$$\{H \in \text{DECOMP}(C_{i_1}):\ c(H) \ge 0\} \cup \text{DECOMP}([-, pred(end(C_{j_1}))])$$

to T. The hump decomposition can be done in time

$$O(|C_i| + |C_j| + |C_{i_1}| + |C_{j_1}|). \tag{9}$$

The insertion and deletion of humps can be done in time

$$O((\sqrt{|C_i|} + \sqrt{|C_j|} + \sqrt{|C_{i_1}|} + \sqrt{|C_{j_1}|}) \times \log n). \tag{10}$$

By (7), (8), (9), and (10), the overall time complexity of the all-pairs algorithm is

$$O(n) + \sum_{1 \le i, j \le p} (O(|C_i| + |C_j|) + O(\sqrt{|C_i|} + \sqrt{|C_j|}) \times \log n + O(|C_i| + |C_j|) \times \log n),$$

which is $O(np \log n)$. □

5. NP-Completeness. In this section we sketch the proof for the following theorem.

THEOREM 5.1. *The race-condition detection problem for a parallel program that uses more than one semaphore is NP-complete.*

The proof is by reduction from the NP-complete uniform-cost SMMCC problem, where the node costs are restricted to ± 1 [6]. The reduction has three steps. Given an SMMCC problem for a uniform-cost graph G_0 of n nodes, we construct $O(\log n)$ chain graphs

with $n + 2$ semaphores. The first step of the reduction shows that the SMMCC problem for G_0 can be reduced to determining whether each of those $O(\log n)$ chain graphs has a valid schedule. The second step shows that each of those $O(\log n)$ chain graphs can be *simulated* by a chain graph with only two semaphores. In other words, the simulated chain graph has a valid schedule if and only if the simulating chain graph has a valid schedule. The last step shows that the simulating chain graph has a valid schedule if and only if $v \to w$, for some v and w, in the same chain graph. We elaborate the details of the reduction in the Appendix.

Acknowledgments. We thank the anonymous referees for their helpful remarks that significantly improved the presentation of the paper.

Appendix. Let G be a chain graph. Each node of G is an operation on a semaphore. An operation on semaphore S is either $+S$, incrementing the value of S by one, or $-S$, decrementing the value of S by one. A subschedule of G is *valid* if the value of each semaphore is always nonpositive during the execution of the subschedule. Let v and w be two nodes of G. If there exists a subschedule of G in which v precedes w, then we say $v \to w$. Clearly, determining whether $v \to w$ is in NP. If G is allowed to use more than one semaphore, then we prove the NP-hardness by a three-step reduction from the uniform-cost SMMCC problem.

A.1. *First Step.* Let G_0 be an acyclic directed graph of n nodes, $v_1, v_2, \ldots, v_n$. The cost of each node is either $+1$ or -1. Suppose we would like to know whether $h(G_0) \leq \ell$. We construct a chain graph G_1 composed of $2n + 2$ chains of operations on $n + 2$ semaphores, and argue that G_1 has a valid schedule if and only if $h(G_0) \leq \ell$. Note that $0 \leq h(G_0) \leq n$. Therefore, $h(G_0)$ can be obtained by $O(\log n)$ queries of whether a chain graph of $n + 2$ semaphores has a valid schedule.

Let n^+ be the number of nodes with positive costs. Let n^- be the number of nodes with negative costs. Clearly, $n^+ - n^-$ is the sum of node costs of G_0. Let d_i be the number of outgoing arcs of G_0 from v_i. The $n + 2$ semaphores for G_1 are $S_1, S_2, \ldots, S_n, S_\alpha, S_\beta$. Let the $2n + 2$ chains of G_1 be $C_1, \ldots, C_{n+1}$, and $C'_1, \ldots, C'_{n+1}$, all initially empty. We construct G_1 from G_0 by the procedure CONSTRUCT() in Figure 9, which runs in polynomial time. Without loss of generality we can assume that $\ell - n^+ + n^-$, the number in the penultimate statement of the procedure CONSTRUCT, is nonnegative, since otherwise $h(G_0) > \ell$ is immediately concluded.

An example is shown in Figure 10. The intuition is as follows. The (only) operation for S_α in C_i corresponds to v_i, where the "sign" of S_α reflects the cost of v_i. We use the first n semaphores, $S_1, \ldots, S_n$, to enforce the execution of these n operations for S_α to obey the precedence constraints imposed by G_0. In Figure 10, for instance, in order to reach the $-S_\alpha$ in C_4, we have to unlock the $+S_2$ (and $+S_3, +S_5$) in the same chain first. Since the only $-S_2$ is after the $+S_\alpha$ in C_2, we know the $+S_\alpha$ in C_2 must be executed before the $-S_\alpha$ in C_4.

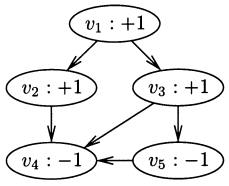
The $-S_\beta$'s at the end of $C_1, \ldots, C_n$ are to ensure that as long as the last $+S_\beta$ in C_{n+1} is executed, all operations in $C_1, \ldots, C_n$ are already executed. The function of those ℓ

```
CONSTRUCT(G_0)
1   For i := 1 to n do
2     For j := 1 to n do
3       If v_j v_i is an arc of G_0 then
4         Append a +S_j to C_i.
5     If the cost of v_i is +1 then
6       Append a +S_α to C_i.
7     else (i.e., the cost of v_i is −1)
8       Append a −S_α to C_i.
9       Append a +S_α and −S_α to C'_i.
10    Append d_i copies of −S_i to C_i.
11    Append a −S_β to C_i.
12  Append n copies of +S_β to C_{n+1}.
13  Append ℓ − n^+ + n^− copies of +S_α to C_{n+1}.
14  Append ℓ copies of −S_α to C'_{n+1}.
```

Fig. 9. The procedure constructs a chain graph G_1 such that G_1 has a valid schedule if and only if $h(G_0) \leq \ell$.

copies of $-S_\alpha$ in C'_{n+1} is clear: the larger ℓ, the easier for G_1 to have a valid schedule. The purpose of the $+S_\alpha$, $-S_\alpha$ pairs in $C'_1, \ldots, C'_n$ and those $\ell - n^+ + n^-$ copies of $+S_\alpha$'s at the end of C_{n+1} will become clear as we proceed. Basically they are used to ensure that G_1 has some kind of "pairwise" schedule, as long as G_1 has a valid schedule. One can verify that there are the same number of $+S_i$'s and $-S_i$'s in G_1, for each $1 \leq i \leq n+2$.

For the rest of the subsection, we prove that $h(G_0) \leq \ell$ if and only if G_1 has a valid schedule. An implication of the following proofs is that G_1 has a valid schedule if and only if it has a valid schedule executable by some procedure PAIRWISE, which will be given in the proofs.



C_1	C_2	C_3	C_4	C_5	C_6	C'_4	C'_5	C'_6
			$+S_2$		$+S_\beta$	$+S_\alpha$	$+S_\alpha$	$-S_\alpha$
			$+S_3$		$+S_\beta$	$-S_\alpha$	$-S_\alpha$	$-S_\alpha$
	$+S_1$	$+S_1$	$+S_5$	$+S_3$	$+S_\beta$			
					$+S_\beta$			
$+S_\alpha$	$+S_\alpha$	$+S_\alpha$	$-S_\alpha$	$-S_\alpha$	$+S_\beta$			
$-S_1$	$-S_2$	$-S_3$		$-S_5$				
$-S_1$		$-S_3$						
$-S_\beta$	$-S_\beta$	$-S_\beta$	$-S_\beta$	$-S_\beta$	$+S_\alpha$			

Fig. 10. An example for the first step of the reduction. Suppose we would like to determine whether $h(G_0) \leq 2$, where G_0 is the graph on top. We then construct, by CONSTRUCT, the chain graph G_1 at the bottom. Note that there are one $+S_\alpha$ at the end of C_6 and two $-S_\alpha$ in C'_6, according to the last two statements of CONSTRUCT. It follows from Lemmas A.1(1) and A.2 that that exists a valid schedule of the chains at the bottom if and only if the height of the graph on the top is at most two.

LEMMA A.1.

1. *If G_1 has a valid subschedule containing the last $+S_\alpha$ of C_{n+1}, then $h(G_0) \leq \ell$.*
2. *If G_1 has a valid schedule, then $h(G_0) \leq \ell$.*

PROOF. Clearly, it suffices to prove the first statement, since the second statement follows immediately from the first statement.

Let X be a valid subschedule of G_1 as described in the lemma. We show $h(G_0) \leq \ell$. Let O_i be the operation of S_α in C_i. Since X is valid and contains the last $+S_\alpha$ of C_{n+1}, X must contain all the operations in $C_1, \ldots, C_n$. Therefore, every O_i, $1 \leq i \leq n$, is in X.

Suppose the order of those O_i's in X is $O_{k_1}, O_{k_2}, \ldots, O_{k_n}$. By the definition of CONSTRUCT, if v_j is reachable from v_i in G_0, then O_j does not precede O_i in X. It follows that the sequence $Y = v_{k_1} v_{k_2} \cdots v_{k_n}$ is a schedule of G_0. Therefore, it suffices to show $h(Y) \leq \ell$.

Assume $h(Y) > \ell$ for a contradiction. If we count only those O_i's as the operations for S_α in X, then the maximum value of S_α would be greater than ℓ during the execution of X. Note that there are $\ell + n^-$ other $-S_\alpha$'s in $C'_1, \ldots, C'_{n+1}$, which are the only hope for bringing the maximum value of S_α down to zero. By the construction of $C'_1, \ldots, C'_n$, however, we know n^- of those $-S_\alpha$'s have to be preceded in X by n other $+S_\alpha$'s. It follows that even if we count all operations for S_α together, the maximum value of S_α would be greater than zero during the execution of X. This contradicts the fact that X is a valid schedule of G_1. □

LEMMA A.2. *If $h(G_0) \leq \ell$, then G_1 has a valid schedule.*

PROOF. Let $Y = v_{k_1} v_{k_2} \cdots v_{k_n}$ be a schedule of G_0 with $h(Y) \leq \ell$. Let m_i be the sum of costs of $v_{k_1}, \ldots, v_{k_i}$. Clearly, $m_n = n^+ - n^-$, which is the sum of node costs of G_0. Since $h(Y) \leq \ell$, we know that $m_i \leq \ell$ for every $1 \leq i \leq n$. We claim that G_1 can be executed by the procedure PAIRWISE in Figure 11.

Note that in the schedule of G_1 executed by PAIRWISE, each operation $-S_i$ is immediately followed by an operation $+S_i$. Not every chain graph has such a "pairwise" schedule, however, we show that G_1 does. We first show that the first for-loop of PAIRWISE can be finished for G_1. Specifically, suppose the following claim holds:

CLAIM. *For each $1 \leq i \leq n$, the i*th *iteration of the first for-loop of* PAIRWISE *can be executed for G_1. Furthermore, after executing the i*th *iteration,*

- *the remaining operations in C_{k_i} are d_{k_i} copies of $-S_{k_i}$'s followed by a $+S_\beta$; and*
- *there are $\ell - m_i$ copies of $-S_\alpha$'s available in $C'_1, \ldots, C'_{n+1}$.*

It is then not hard to see that after the execution of the first for-loop of PAIRWISE, the remaining operation in each C_i is a $-S_\beta$. Therefore, the second for-loop of PAIRWISE can be finished, since there are n copies of $+S_\beta$'s available in C_{n+1}.

By Lemma A.1, we know that after executing the first For-loop, the number of $-S_\alpha$'s in $C'_1, \ldots, C'_{n+1}$ is $\ell - m_n$, which is equal to the number of $+S_\alpha$'s at the end of C_{n+1}. Therefore, the last for-loop of PAIRWISE can be finished. The lemma is proved.

```
Procedure PAIRWISE
1   For k := k_1, k_2, ..., k_n do
2     For j := 1 to n do
3       If v_j v_k is an arc of G_0 then
4         Execute a -S_k in C_j.
5         Execute the +S_k in C_k.
6     If O_k = +S_α then
7       Execute one of the -S_α's
8         in C'_1, C'_2, ..., C'_{n+1}.
9       Execute the +S_α in C_k.
10    else (i.e., O_k = -S_α)
11      Execute the -S_α in C_k.
12      Execute the +S_α in C'_k.
13  For i := 1 to n do
14    Execute the -S_β in C_i.
15    Execute a +S_β in C_{n+1}.
16  For i := 1 to ℓ - m_n do
17    Execute a -S_α in C'_1, ..., C'_{n+1}.
18    Execute a +S_α in C_{n+1}.
```

Fig. 11. Procedure PAIRWISE.

It remains to prove the above claim by induction on i. For convenience we abbreviate k_i to k for the rest of the proof. When $i = 1$, we know v_k does not have any incoming arcs from other nodes. Therefore, the for-loop with index j in the first iteration does not execute any operation. We then consider the if-statement.

- If $O_k = -S_\alpha$, then $c(v_k) = -1$, and thus $m_1 = -1$. There is a $+S_\alpha$ in C'_k by the definition of CONSTRUCT. We can execute the else-part of the if-statement without problem. Since the second operation in C'_k is a $-S_\alpha$, these two steps increase the number of $-S_\alpha$'s available in $C'_1, \ldots, C'_{n+1}$ by one.
- If $O_k = +S_\alpha$, then $c(v_k) = 1$, and thus $m_1 = 1$. Since v_k is the first node in Y, $h(Y)$ is at least one, and thus $\ell \geq 1$. We can therefore execute the then-part of the if-statement without problem. The number of $-S_\alpha$'s available in $C'_1, \ldots, C'_{n+1}$ is decreased by one.

Clearly, after executing the first iteration, in which the only executed operation in C_k is O_k, the remaining operations in C_k are exactly as that described in the claim. Note that before executing the first iteration, the number of available $-S_\alpha$'s is ℓ by the definition of CONSTRUCT. Therefore, after executing the first iteration, the number of available $-S_\alpha$'s is exactly $\ell - m_1$. This confirms the inductive basis.

Let i' be an integer with $1 < i' \leq n$. Assume that the claim holds for every $1 \leq i < i'$. We show it holds for $i = i'$. Consider the ith iteration. Note that for every j such that $v_j v_k$ is an arc of G_0, O_j must have been executed. By the inductive hypothesis we know those d_j copies of $-S_j$'s are already available before executing the ith iteration. Therefore, the for-loop with index j will proceed without problem, since there are exactly d_j copies of

$+S_j$'s in G_1 by the definition of CONSTRUCT. We then consider the if-statement:

- If $O_k = -S_\alpha$, then $m_i = m_{i-1} - 1$. We know there is a $+S_\alpha$ in C'_k. Thus, the else-part can proceed without problem. Since the second operation in C'_k is a $-S_\alpha$, these two steps increase the number of available $-S_\alpha$'s in $C'_1, \ldots, C'_{n+1}$ by one.
- If $O_k = +S_\alpha$, then $m_i = m_{i-1} + 1$. The inductive hypothesis says that the number of $-S_\alpha$'s available in $C'_1, \ldots, C'_{n+1}$ is $\ell - m_{i-1}$ before executing the ith iteration. That number is at least one since $\ell - m_{i-1} - 1 = \ell - m_i \geq 0$. Therefore, the then-part of the if-statement can be executed without problem. The number of available $-S_\alpha$'s in $C'_1, \ldots, C'_{n+1}$ is decreased by one.

Therefore, the ith iteration can be executed, and thus the remaining operations in C_k are as required.

It follows from the inductive hypothesis that the number of available $-S_\alpha$'s in $C'_1, \ldots, C'_{n+1}$ is $\ell - m_{i-1}$. By the above case analysis we see that the number is exactly $\ell - m_i$ after executing the ith iteration. The claim is proved. □

If G_1 has a valid schedule, then by Lemma A.1(2) we know $h(G_0) \leq \ell$. It then follows from the proof of Lemma A.2 that G_1 has a valid schedule executable by PAIRWISE. Therefore, we have the following lemma.

LEMMA A.3. *G_1 has a valid schedule if and only if G_1 has a valid schedule executable by* PAIRWISE.

A.2. *Second Step*. In this subsection we show that the G_1 constructed in the first step can be simulated by another chain graph G_2, which uses only two semaphores, T_1 and T_2. G_2 has $2n + 3$ chains. The first chain, denoted C_0, is composed of two $-T_1$'s and two $-T_2$'s. The remaining $2n + 2$ chains are obtained from those of G_1 as follows. We replace every operation $-S_i$ (and $+S_i$) by a *unit* $-U_i$ (and $+U_i$) for each $1 \leq i \leq n+2$. Each unit, $-U_i$ or $+U_i$, is a sequence of operations on T_1 and T_2, as shown in Figure 12. We also denote those $2n + 2$ chains of G_2 by $C_1, \ldots, C_{n+1}$ and $C'_1, \ldots, C'_{n+1}$. Clearly, G_2 can be constructed in polynomial time.

Note that the sequence of operations in each unit is arranged such that only a $-U_i$ and a $+U_i$ can "unlock" each other. To be more specific, suppose each of T_1 and T_2 has initial

$+T_1$	$+T_1$
$+T_2$	$+T_2$
$-T_1$ ⎫	$+T_1$ ⎫
$+T_2$ ⎪	$-T_2$ ⎪
⋮ ⎬ $i+1$ pairs	⋮ ⎬ $i+1$ pairs
$-T_1$ ⎪	$+T_1$ ⎪
$+T_2$ ⎭	$-T_2$ ⎭
$-T_2$	$+T_2$
$-T_2$	$+T_2$
$-T_2$	$-T_1$
$-T_2$	$-T_1$

Fig. 12. The sequence of operations for a $-U_i$ is at the left and that for a $+U_i$ is at the right, for any $1 \leq i \leq n+2$.

value -2, which will be the case if the four operations in C_0 are executed. Consider a graph U_{ij} for some $1 \leq i, j \leq n+2$ composed of two units, $-U_i$ and $+U_j$, each forms a single chain. One can easily verify that U_{ij} has a valid schedule if $i = j$. Moreover, after executing all the operations of U_{ii}, the values of T_1 and T_2 go back to -2.

We claim that G_1 has a valid schedule if and only if G_2 has a valid schedule. The only-if part is straightforward. Suppose G_1 has a valid schedule. By Lemma A.3, G_1 has a valid schedule executable by PAIRWISE. Note that we can execute the four operations of C_0 first, which decrease the value of both semaphores down to -2. Clearly, the remaining $2n+2$ chains of units can be completely pairwisely executed by following the sequence of corresponding operations in G_1 executed by PAIRWISE. Therefore, G_2 has a valid schedule.

It takes some added work to prove the other direction of the above claim. A unit is *active* if its third operation is executed. A unit is *finished* (and thus inactive) if its fifth-to-last operation is executed. Suppose G_2 has a valid schedule. Consider the sequence of the units of G_2 that become active in the valid schedule. It follows from the following lemma that the corresponding sequence of operations of G_1 is a valid schedule of G_1. In fact it is "pairwise", since in the schedule each $-S_i$ is immediately followed by a $+S_i$.

LEMMA A.4. *Consider the execution of a valid subschedule.*

1. *When there is no active unit, the next unit that becomes active must be a $-U_i$ for some $1 \leq i \leq n+2$.*
2. *Before that active $-U_i$ is finished, a $+U_i$ must become active.*
3. *No unit will become active unless these two active units are finished.*

PROOF. At the beginning of the valid schedule, no unit is active. We show the first statement of the lemma holds. At this moment there are two $-T_1$'s and two $-T_2$'s available (in C_0). They are our only hope for activating any unit, since each unit is guarded by two $+T_1$'s and two $+T_2$'s. Assume for a contradiction that the first unit becoming active is a $+U_i$ for some $1 \leq i \leq n+2$. Note that as soon as the first $+U_i$ becomes active, at least two $+T_1$'s are already executed. Since at most two $-T_1$'s are executed so far, there is no way to activate any other unit. The execution thus cannot proceed.

When the first unit $-U_i$ becomes active, one can see that the second statement of the lemma holds by verifying the following:

- The active $-U_i$ will not be finished unless another unit becomes active, since otherwise the execution will be blocked by some $+T_2$'s.
- The next active unit must be a $+U_j$ for some $1 \leq j \leq n+2$, since otherwise the execution will be blocked by some $+T_2$'s.
- If $i < j$, the execution will be blocked by some $+T_1$'s. If $i > j$, then the execution will be blocked by some $+T_2$'s. Therefore, the next active unit must be a $+U_i$.

When those two units are active, in order to activate other units, we can only hope for the $-T_1$'s at the end of the active $+U_i$. In order to reach those $-T_1$'s, the preceding consecutive $+T_2$'s must be penetrated. Hence, at least two $-T_2$'s at the end of the active $-U_i$ must be executed first. Therefore, those two active units $-U_i$ and $+U_i$ must be

finished before any other unit becomes active. This confirms the third statement of the lemma.

Note that as soon as the active $+U_i$ is finished (and so must be the active $-U_i$), the situation is exactly the same as the situation at the very beginning of the execution. Namely we have two $-T_1$'s and two $-T_2$'s available, which are again our only hope for activating any other units. Therefore, all the above argument follows inductively. The lemma is proved. □

A.3. *Third Step*. Let v be the first operation of the C_0 in G_2. Let w be the last operation of the C_{n+1} in G_2. We claim that $v \to w$ if and only if G_2 has a valid schedule. Note that v is always the first node in any valid subschedule of G_2. The if-part of the claim holds trivially. It remains to prove the only-if-part of the claim.

Let X be a valid subschedule of G_2 in which v precedes w. Consider the sequence of the units of G_2 that become active while executing X. It follows from Lemma A.4 that the corresponding sequence of operations of G_1 is a valid subschedule of G_1, which definitely contains the last $+S_\alpha$ of the C_{n+1} in G_1. Therefore, G_1 has a valid schedule by Lemmas A.1(2) and A.2. Finally it follows from the claim in Section A.2 that G_2 has a valid schedule.

References

[1] H. M. Abdel-Wahab. Scheduling with Application to Register Allocation and Deadlock Problems. Ph.D. thesis, University of Waterloo, 1976.

[2] H. M. Abdel-Wahab and T. Kameda. Scheduling to minimize maximum cumulative cost subject to series-parallel precedence constraints. *Operations Research*, 26(1):141–158, 1978.

[3] H. M. Abdel-Wahab and T. Kameda. On strictly optimal schedules for the cumulative cost-optimal scheduling problem. *Computing*, 24:61–86, 1980.

[4] P. A. Emrath, S. Ghosh, and D. A. Padua. Event synchronization analysis for debugging parallel programs. In *Proceedings of Supercomputing*, pages 580–588, Reno, NV, 1989.

[5] P. A. Emrath, S. Ghosh, and D. A. Padua. Detecting nondeterminacy in parallel programs. *IEEE Software*, 9(1):69–77, 1992.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability—A Guide to the Theory of NP-Completeness*. Freeman, San Francisco, CA, 1979.

[7] K.-S. Ha, E.-K. Ryu, and K.-Y. Yoo. Space-efficient first race detection in shared memory programs with nested parallelism. In J. Fagerholm, J. Haataja, J. Järvinen, M. Lyly, P. Råback, and V. Savolainen, editors, *Proceedings of the 6th International Conference on Applied Parallel Computing and Advanced Scientific Computing*, pages 253–263, Espoo, Finland, 2002. Lecture Notes in Computer Science 2367, Springer-Verlag, Berlin.

[8] D. P. Helmbold and C. E. McDowell. A Class of Synchronization Operations that Permit Efficient Race Detection. Technical Report UCSC-CRL-93-29, University of California at Santa Cruz, 1993.

[9] D. P. Helmbold and C. E. McDowell. A taxonomy of race conditions. *Journal of Parallel and Distributed Computing*, 33:159–164, 1996.

[10] D. P. Helmbold, C. E. McDowell, and J.-Z. Wang. Analyzing traces with anonymous synchronization. In *Proceedings of the International Conference on Parallel Processing*, pages II70–II77, St. Charles, IL, August 1990.

[11] A. Itzkovitz, A. Schuster, and O. Zeev-Ben-Mordehai. Toward integration of data race detection in dsm systems. *Journal of Parallel and Distributed Computing*, 59(2):180–203, 1999.

[12] P. N. Klein, H.-I Lu, and R. H. B. Netzer. Race-condition detection in parallel computation with semaphores. In J. Díaz and M. Serna, editors, *Proceedings of the 4th Annual European Symposium*

on Algorithms, pages 445–459, Barcelona, Spain, 1996. Lecture Notes in Computer Science 1136, Springer-Verlag, Berlin.

[13] H.-I Lu, P. N. Klein, and R. H. B. Netzer. Detecting race conditions in parallel programs that use one semaphore. In F. K. H. A. Dehne, J.-R. Sack, N. Santoro, and S. Whitesides, editors, *Proceedings of the* 3*rd Workshop on Algorithms and Data Structures*, pages 471–482, Montréal, Canada, 1993. Lecture Notes in Computer Science 709, Springer-Verlag, Berlin.

[14] R. H. B. Netzer and S. Ghosh. Efficient race condition detection for shared-memory programs with post/wait synchronization. In *Proceedings of the International Conference on Parallel Processing*, pages II242–II246, St. Charles, IL, August 1992.

[15] R. H. B. Netzer and B. P. Miller. On the complexity of event ordering for shared-memory parallel program executions. In *Proceedings of the International Conference on Parallel Processing*, pages II93–II97, August 1990.

[16] R. H. B. Netzer and B. P. Miller. What are race conditions? Some issues and formalizations. *ACM Letters on Programming Languages and Systems*, 1(1):74–88, 1992.

[17] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[18] M. L. Simmons, A. H. Hayes, J. S. Brown, and D. A. Reed, editors. *Debugging and Performance Tuning for Parallel Computing Systems*. IEEE Computer Society Press, Los Alamitos, CA, 1996.