Detecting Race Conditions in Parallel Programs that Use one Semaphore

Hsueh-I Lu Philip N. Klein and Robert H. B. Netzer

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-93-29
June 1993

Appears in Workshop on Algorithms and Data Structures (WADS'93), August 11-13, Montreal, Canada

Detecting Race Conditions in Parallel Programs that Use One Semaphore

Hsueh-I Lu Philip N. Klein* Robert H. B. Netzer

Department of Computer Science, Brown University, Providence, RI 02912, USA**

Abstract. We address a problem arising in debugging parallel programs, detecting race conditions in programs using a single semaphore for synchronization. It is NP-complete to detect races in programs that use many semaphores. For the case of a single semaphore, we give an algorithm that takes $O(n^{1.5}p)$ time, where p is the number of processors and n is the total number of semaphore operations executed. Our algorithm constructs a representation from which one can determine in constant time whether a race exists between two given events.

1 Introduction

Race condition detection is a crucial aspect of developing and debugging shared-memory parallel programs. Explicit synchronization is usually added to such programs to coordinate access to shared data, and *race conditions* result when this synchronization does not force concurrent processes to access data in the expected order. One way to dynamically detect races in a program 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,[1] some operations (belonging to different processes) could have potentially executed in an order different than what was traced. In this paper, we present fast algorithms for computing the order in which an execution's semaphore operations *could have* executed for the only case when it is tractable: programs that use a single semaphore. Our algorithms can be used to exactly detect race conditions in executions of such programs. Past work has shown that exactly detecting races in programs that use multiple semaphores is NP-complete [9], and has developed exact algorithms for other cases where the problem is efficiently solvable (programs that use types of synchronization weaker than semaphores) [6,8], and heuristics for the multiple semaphore case [4,7]. The complexity for the case of a single semaphore has been an open question.

Our main results are two fast algorithms, and are the first known polynomial-time algorithms that allow exact race detection in programs that use semaphores. Our goal was to solve this problem as efficiently as possible, since parallel programs are typically long-running, and the resulting large traces must be analyzed quickly by our algorithms. Our first algorithm determines

* Research supported by NSF grant CCR-9012357 and NSF PYI award CCR-9157620, together with PYI matching funds from Thinking Machines Corporation and Xerox Corporation. Additional support provided by DARPA contract N00014-91-J-4052 ARPA Order No. 8225.

** Email addresses: {hil,pnk,rn}@cs.brown.edu.

[1] 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 non-empty), and V operations typically signal that some condition is now true.

whether any two given semaphore operations could have executed in a different order than during execution (and can be used to detect whether a race exists between any two particular events) and runs in time and space linear in the total number of semaphore operations. Our second algorithm answers this question for all pairs of operations (and can detect all races in the execution) and runs in $O(n^{1.5}p)$ time, where n is the number of operations and p is the number of processes in the execution.

Computing the order in which an execution's semaphore operations could have executed requires solving a scheduling problem. To determine whether any two operations, a and b, where guaranteed to have executed in a fixed order, we must determine if some *valid* schedule of the execution's operations exists in which b precedes a. A valid schedule is an interleaving of the p processes' semaphore operations that honors the semantics of semaphore-style synchronization; i.e., a linear ordering of the operations such that at each point in the ordering, the number of V operations is never exceeded by the number of P operations (meaning that the semaphore is always nonnegative). Then, if the trace indicates that a preceded b in the actual execution, but a valid schedule exists in which b precedes a, then a and b could have executed in either order.

This scheduling problem can be reduced to a special case of *sequencing to minimize maximum cumulative cost (SMMCC)*. Given an acyclic directed graph G with costs on the nodes, a schedule is a topological ordering of the nodes; i.e. an ordering of the nodes consistent with the arcs. The cumulative cost of the first i nodes of such a schedule is just the sum of the cost of these nodes. Thus minimizing the maximum cumulative cost is an attempt to make sure that the cumulative cost stays low throughout the schedule. The SMMCC problem is NP-complete in general [1,5], but Abdel-Wahab and Kameda present an $O(n^2)$-time algorithm for the special case where G is a series-parallel graph [2] (the time bound was later improved to $O(n \log n)$ [3]). As part of this solution, they give an $O(n \log p)$-time algorithm applicable when G is a chain graph, a graph consisting of a union of p disjoint directed paths.

The problem we address can be reduced to the SMMCC problem in a chain graph augmented with one inter-chain edge. In particular, suppose we want to determine whether there is a valid schedule in which b precedes a. We add an edge from b to a, 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. The cost is zero if there is a valid schedule, and positive otherwise. The augmented chain graph is not series-parallel, so the algorithm of Abdel-Wahab and Kameda is 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.

Theorem 1. *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 a and b of G, one can determine in $O(n)$ time whether there is a valid schedule in which b precedes a.*

In the application, parallel debugging, it is important to exactly detect all races. Hence we need to determine the above for all pairs of nodes a and b. Fortunately, there is a *compact representation* of this information. The relation "a precedes b in all valid schedules" (written $a \prec b$) is a partial order consistent with the original chain graph. To represent this information, it is sufficient that we indicate, for each node a, and for each chain C not containing a, the first node b in C such that a precedes b in all valid schedules. 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 a and b. A race exists if either operation can precede the other. To determine whether b can precede a, we determine the first node in b's chain that is preceded by a in all valid

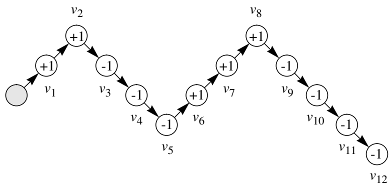

Fig. 1. A hump H of 12 nodes: $v_j, \ldots, v_{j+11}$. The cost of each node is in the circle. By definition $s(H) = -2$, $h(H) = 2$, and $\tilde{h}(H) = 4$. Both of v_{j+1} and v_{j+7} are peaks of H, but only v_{j+1} is useful.

schedules. If this first node is numbered later than b, then b can precede a. If not, b cannot precede a.

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 1. We show how to do much better; in fact the time required by our algorithm is only $O(\sqrt{n})$ times the time required simply to write down the output.

Theorem 2. *Suppose G is as in Theorem 1. The compact representation of the relation "a precedes b in all valid schedules" can be constructed in $O(n^{1.5}p)$ time.*

2 Definitions and Lemmas

For the remainder of the paper, we shall use G' to denote an acyclic graph, and G to denote a chain graph whose node costs are ± 1. Let $\mu(G')$ denote the lowest maximum cumulative cost of a schedule of G'. Thus we shall be interested in computing $\mu(G \cup \{ab\})$ for pairs a, b of nodes of G.

Now we introduce some terminology having to do with schedules. Much of this terminology is adapted from that in [2]. A *segment* of a schedule is a consecutive subsequence. Let $H = v_j v_{j+1} \cdots v_k$ be a sequence consisting of some of the nodes of G. The *cost* of H, denoted $s(H)$, 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 height of any node in H. A node of maximum height in H is called a *peak*. 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 *chain* is a connected graph with all indegrees and outdegrees at most one. We are concerned primarily with graphs G consisting of the disjoint union of chains. For convenience, we assume in such a graph the existence of an *initial pseudonode*, preceding all nodes, and a *terminal pseudonode*, following all nodes, each of cost zero.

Suppose H is a chain in G' containing a peak v_ℓ such that (1) every node of H preceding v_ℓ has nonnegative height, and (2) every node of H following v_ℓ has height at least the cost of H. In this case, we call H a *hump* in G', 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 positive, and an L*-hump* if its cost is zero. As part of their scheduling algorithm for series-parallel graphs, Abdel-Wahab and Kameda show that in linear time a sequence of nodes can be decomposed into a series of humps. It can be proved that the humps have the following properties.

Hump decomposition properties:

– The series consists of N-humps, followed by at most one L-hump, followed by P-humps.

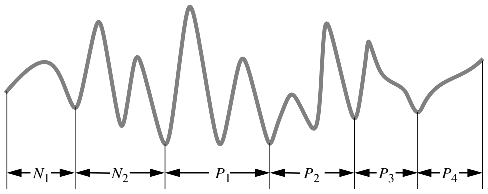

Fig. 2. A chain which is decomposed into two N-humps, one L-hump, and three P-humps.

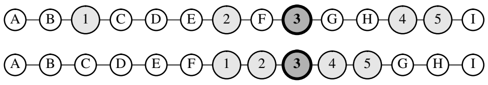

Fig. 3. The second sequence of nodes is obtained from the first one by clustering the numbered nodes to node 3.

– The N-humps are in increasing order of height, and the P-humps are in decreasing order of reverse height. Furthermore, for any two N-humps in the series, the reverse height of the one ahead is less than the height of the one behind. For any two P-humps in the series, the reverse height of the one ahead is greater than the height of the one behind.
– Every peak of the sequence occurs at or before the first P-hump.

For example, the chain in Figure 2 is decomposed into two N-humps, one L-hump, and three P-humps. When we refer to *the* humps of a chain or a graph, we mean the humps obtained by this hump decomposition process. We refer the process of carrying out this decomposition on a sequence as *constructing the humps* of the sequence. A node is called a *hump boundary* of the sequence if it is the last node of some hump of the sequence. Let hb(S) denote the set of hump boundaries of some sequence S. One can prove the following property.
Hump boundary property: For sequences S_1 and S_2, we have hb$(S_1S_2) \subseteq$ hb$(S_1) \cup$ hb(S_2). Furthermore, if hb(S_1S_2) contains the last node of S_1, then hb$(S_2) \subseteq$ hb(S_1S_2).

It will turn out that once we decompose a sequence of nodes into humps, we need not be concerned with the internal structure of these humps. We need only store the heights, costs, and boundaries of the humps. Thus a sequence consisting of ℓ humps can be represented by a length-ℓ sequence of triples (height, cost, boundary). We call this sequence the *hump representation* of the sequence. Using the hump boundary properties, one can straightforwardly derive the hump representation of S_1S_2 from the hump representation of S_1 and that of S_1, and can derive the hump representation of S_2 from that of S_1S_2.

Three lemmas are useful to our results. They are all generalizations of lemmas in [2]. The first lemma concerns an operation on a schedule called *clustering* the nodes of a hump. Suppose H is a hump in 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, we say H is *clustered in* S. Consider the humps obtained in applying hump decomposition to the chains of G; if each of these humps is clustered in S, we say the schedule S is *clustered*. If a hump is not clustered in a schedule, 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 3. *Let G' be an acyclic graph with costs on nodes and H be a hump in 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 $h(T) \leq h(S)$.*

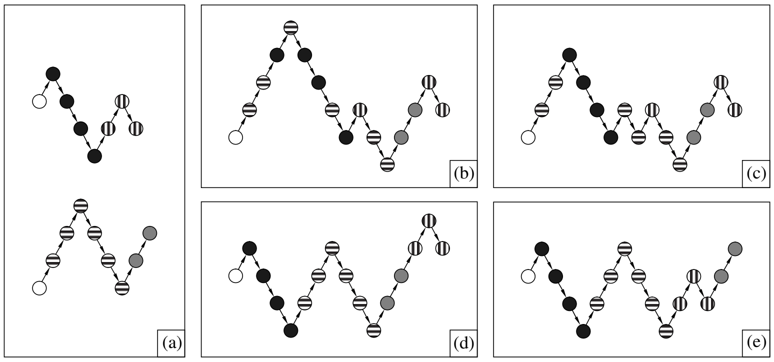

Fig. 4. (a) The graph G is composed of two chains. The first chain is composed of an N-hump followed by a P-hump. The second chain is composed of an L-hump followed by a P-hump. (b) A schedule for G of height four. (c) The schedule obtained from the previous one by clustering the N-hump to its useful 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 of minimum height.

The proof is a modification of the proof in [2] of a similar lemma that requires G' to be a chain graph. An example is shown in Figure 4. The height of the schedule in (c) is smaller than that of the schedule in (b). It follows from Lemma 3 that there is always a clustered optimal schedule of G. Two clustered schedules of the graph in Figure 4-(a) are shown in Figure 4-(d) and (e). The latter is optimal.

Consider a series $S_1 \dots S_m$ of subsequences. It is in *standard form* if it satisfies the following properties:

Standard form properties:

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

Lemma 4. *Let G' be an acyclic graph with node costs. Let A, B, S_1, S_2, and S_3 be subsequences of nodes in G'. Suppose $S = S_1AS_2BS_3$, $T_1 = S_1BAS_2S_3$, and $T_2 = S_1S_2BAS_3$. If the series of BA is in standard form then either $h(S) \geq h(T_1)$ or $h(S) \geq h(T_2)$.*

The proof of Lemma 4 is a modification of a similar lemma in [2] that requires A and B to be humps and requires S, T_1, and T_2 to be schedules of G'.

The following lemma is immediate from Lemma 4; we simply let S_2 be empty.

Lemma 5. *Let G' be an acyclic graph with node costs. Let A, B, and S_2 be subsequences of nodes in G'. Suppose $S = S_1ABS_2$ and $T = S_1BAS_2$. If the series of BA is in standard form then $h(S) \geq h(T)$.*

A schedule of G is in *standard form* if it is clustered and its series of humps in G is in standard form. Let T be any schedule of G in standard form. Recall that by Lemma 3 there is always an optimal schedule S of G which is clustered. The humps of G, while clustered both in T and in 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 5, the resulting ordering has height no more than S. By a series of such exchanges, we eventually obtain T from S. It follows that T's height 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. The algorithm of Abdel-Wahab and Kameda for chain graphs G consists in decomposing the chains into humps and then merging the orderings of humps into a series of humps in standard form; they show this can be done in $O(n \log p)$ time. Let us call this algorithm the *hump-merging algorithm*.

3 The Linear-time Algorithm for a Single Pair

In this section we shall present an algorithm of computing $\mu(G \cup \{e\})$, where $e = ba$, and a and b are nodes of G. The algorithm consists of two steps. The first step merges $G \cup \{e\}$ into three chains with an extra arc. The second step computes the minimal maximum cumulative cost of the merged graph.

3.1 Merging $p-2$ Chains into One Chain

We use $C(a)$ and $C(b)$ to denote the chains containing a and b, respectively. We assume $C(a) \neq C(b)$, for otherwise either no schedule exists or $\mu(G \cup \{e\}) = \mu(G)$, depending on the relative order of a and b. In the first step of our algorithm we use the hump-merging algorithm to find an optimal schedule C^* for the other $p-2$ chains. Let $G^* = C(a) \cup C(b) \cup C^* \cup \{e\}$. The following theorem implies that the minimal maximum cumulative cost of $G \cup \{e\}$ is the same as that of G^*.

Theorem 6. *Let G' be an acyclic graph with node costs, and let C_1 and C_2 be two parallel chains in G'. Suppose G'' is the graph obtained from G' by merging C_1 and C_2 using the hump-merging algorithm. The minimal maximum cumulative cost of G' is equal to that of G''.*

Proof. Since G'' is more restrictive than G', clearly the cost of a schedule for G'' is at least that for G'. To prove the reverse inequality, we first use Lemma 3 to show that, without loss of generality, an optimal schedule for G' can be assumed to consist of the humps of G' in standard form. By Lemmas 4 and 5, we can assume moreover that the humps occur in the same order as in the merge of C_1 and C_2. The resulting schedule is also a schedule for G'', proving the theorem. □

The following observation is the basis for our linear-time algorithm for merging.

Observation 1 *Let C be a chain, and let y be the n^{th} node of C. The hump decomposition algorithm applied to C yields $O(\sqrt{n})$ humps containing nodes occurring before y. Moreover, the hump decomposition applied to p chains comprising n nodes yields $O(\sqrt{np})$ humps.*

Proof. Recall that the costs of nodes are either $+1$ or -1. Hence a hump of height ℓ contains at least ℓ nodes. By the second hump decomposition property, the heights of a series of N-humps must all be different. Thus any sequence of k N-humps comprises $\Omega(k^2)$ nodes. The same property holds for P-humps, and there is at most one L-hump. This proves the first statement.

Suppose there are ℓ_i nodes in the i^{th} chain, where $\ell_1 + \cdots + \ell_p = n$. The total number of humps is $\sum_i O(\sqrt{\ell_i})$, which can be shown to be $O(np)$. □

We now describe the algorithm for merging $p-2$ chains. To initialize, we apply the hump decomposition algorithm to obtain hump representations for each of the chains. This takes $O(n)$

time. The remainder of the algorithm consists of $O(\log p)$ iterations. In each iteration the chains are paired up and each pair is merged into one chain. At the beginning of the i^{th} iteration, we have the hump representation for each of $p/2^{i-1}$ chains. By Observation 1, the total number of humps is $O(\sqrt{np/2^{i-1}})$. Given a pair of chains C' and C'' having respectively n_1 and n_2 humps, we can merge the pair into a single chain and find the humps of this new chain in time $O(n_1+n_2)$. Hence the time for the i^{th} iteration is $O(\sqrt{np/2^{i-1}})$. Therefore the time complexity of all $O(\log p)$ iterations is $\sum_i O(\sqrt{np/2^{i-1}}) = O(\sqrt{np})$, which is $O(n)$.

3.2 Finding an Optimal Schedule of Three Chains with One Arc

Let (V_1, V_2) denote an ordered partition of $V - \{a\}$ that is consistent with G (i.e. no edge of G goes from a node in V_2 to a node in V_1) and such that all nodes of $C(a)$ preceding a are in V_1 and all nodes of $C(a)$ preceded by a are in V_2. We call such an ordered partition *proper*. Note that a proper partition is determined by the last node of $C(b)$ in V_1 and the last node of C^* in V_1. We call the boundaries right after these last nodes *breakpoints*. For the case that all nodes of C^* are in V_2, the breakpoint in C^* is the boundary right before the first node of C^*. Because there are at most n possible breakpoints in $C(b)$ and at most n possible breakpoints in C^*, the number of proper (V_1, V_2)-partitions is $O(n^2)$.

We say a schedule is *consistent* with (V_1, V_2) if it has the form S_1aS_2, where S_1 is an ordering of V_1 and S_2 is an ordering of V_2. Let $h(V_1, V_2)$ be the minimum height of a schedule consistent with (V_1, V_2). Clearly the minimum height of a schedule for G^* in which b precedes a is given by

$$\mu(G^*) = \min_{(V_1,V_2)} h(V_1, V_2), \tag{1}$$

where the minimum is taken over all proper partitions (V_1, V_2) of $V - \{a\}$ such that $b \in V_1$.

To find $h(V_1, V_2)$, let $G^*(V_1)$ and $G^*(V_2)$ be the subgraphs of G^* induced by V_1 and V_2, respectively. Each consists of three parallel chains. We apply the hump-merging algorithm to find optimal schedules S_1^* and S_2^* of these graphs. Then $S_1^* a S_2^*$ is a schedule of G^* consistent with (V_1, V_2), and its height is

$$\max\{\mu(G^*(V_1)), \quad \mu(G^*(V_2)) + \sum_{v \in V_1 \cup \{a\}} s(v)\}. \tag{2}$$

Any consistent schedule of lower height would induce better schedules for $G_1^*(V_1)$ and $G^*(V_2)$, a contradiction. Thus $S_1^* a S_2^*$ is optimal, and $h(V_1, V_2)$ is given by (2).

To compute $\mu(G^*)$ according to (1), it is sufficient to consider each of the $O(n^2)$ proper partitions, and compute (2) for each. In order to achieve linear time, however, we must restrict the set of breakpoints considered. The following observation is a step in that direction.

Observation 2 *We need only consider breakpoints in $C(b)$ and C^* that are hump boundaries.*

Proof. Let S be an optimal schedule for G^*. Let us construct humps for C^* and for the subchain of $C(b)$ which is composed of nodes after b in $C(b)$. By Observation 1, each has $O(\sqrt{n})$ humps. Let T be obtained from S by clustering these humps. By Lemma 3, T is also an optimal schedule. By definition of clustering, each of these humps is consecutive in T, so none is separated by a. Thus in the schedule T, the element a must occur between two of these humps. It follows that in the equation (1), we may restrict the proper partitions to those where the breakpoints in $C(b)$ and C^* lie between these humps. □

By careful examination of these hump boundaries, we can further restrict the search for an optimal schedule. Namely, for any breakpoint in $C(b)$, we show below how to quickly determine just two breakpoints in C^* such that there is an optimal schedule using one of these two. We exploit this property in a procedure to compute the value of (1). Let $C_1(a)$ be the subchain of $C(a)$ up to but not including a, and let $C_2(b)$ be the subchain beginning after a. Let $C_1(b)$ be the subchain of $C(b)$ up to and including b, and let $C_2(b)$ be the remainder of $C(b)$. In the procedure below we assume that hump decompositions are given for C^*, $C_1(a)$, $C_2(a)$, $C_1(b)$, and $C_2(b)$. All these hump decompositions can be found in $O(n)$ time in a preprocessing step.

For the sake of our all-pairs algorithm, we present a procedure MINSCHED that takes an additional parameter. Given a node y of $C(b)$, MINSCHED computes the value of (1) where the minimum is taken over all proper partitions (V_1, V_2) where $b \in V_1$ and $y \in V_2$. Assuming the hump decompositions described above are provided, the time required is $O(\sqrt{n\ell})$, where ℓ is the number of nodes in $C(b)$ between b and x. When y is the terminal pseudonode, the value computed is the minimum height of a schedule for G^* in which b precedes a. In this case the time required is $O(\sqrt{n|C(b)|})$.

The procedure MINSCHED is as follows. Consider the humps of $C_2(b)$. By Observation 1, there are $O(\sqrt{\ell})$ hump boundaries occurring before y. We will try each of these hump boundaries as a breakpoint for $C(b)$. For each, we determine in $O(\sqrt{|C^*|})$ time two candidate breakpoints in C^*, each a hump boundary. Combining a breakpoint for $C(b)$ with a breakpoint for C^* yields a proper partition (V_1, V_2). It follows by hump boundary property that from the given hump decompositions we can obtain the humps of $C(b) \cap V_1$ and $C(b) \cap V_2$ in $O(\sqrt{|C(b)|})$ time (the hump boundaries are a subset of those appearing in the given hump decompositions). We similarly obtain the humps of $C^* \cap V_1$ and $C^* \cap V_2$. Using these humps we compute $\mu(G^*(V_1))$ and $\mu(G^*(V_2))$ in $O(\sqrt{n})$ time, and thus determine the value of (2). The total time required by the procedure is thus $O(\sqrt{\ell n})$.

We commence the proof that for any breakpoint in $C(b)$, we can determine two breakpoints in C^* that are sufficient to examine. First, we know from Observation 2 that we need only consider boundaries between humps of C^*. Every such boundary is either next to an N-hump (i.e. between two N-humps or just before the first N-hump or just after the last) or next to a P-hump, or both. We show that by scanning all the boundaries in the first category, one can be determined which is at least as good a breakpoint as all the others in that category, and similarly for the second category.

Lemma 7. *For any fixed breakpoint in $C(b)$, a breakpoint next to an N-hump can be found in $O(\sqrt{n})$ time that is just as good as all the other such breakpoints.*

The analogous lemma holds for the breakpoints next to P-humps, and is proved symmetrically. The proof of Lemma 7 relies on the following observation, which follows directly from Lemma 5.

Observation 3 *Let G' be an acyclic graph with node costs. Let S_1, S_2, A, and B be subsequences of nodes in G', where $s(B) \leq 0$. Suppose $S = S_1ABS_2$ and $T = S_1BAS_2$. If either (1) $h(A) \geq h(B)$, or (2) $s(A) \geq 0$, then $h(S) \geq h(T)$.*

Let $N_1, N_2, \ldots, N_q$ be the series of N-humps in C^*, which is in standard form. We must select one best breakpoint from among the $q+1$ occurring between these humps. For $i = 0, 1, \ldots, q$, let (W_i, W_i') be the proper partition with $N_1, \ldots, N_i$ in W_i and $N_{i+1}, \ldots, N_q$ in W_i'. We define $T_i = S^*(W_i, W_i')$. Note that $N_1, \ldots, N_q$ are in the same order in $T_0, \ldots, T_q$. Without loss of generality, we may assume that all humps other than $N_1, \ldots, N_q$ are also in the same order in $T_0, \ldots, T_q$ as well. Suppose $T_0 = H_1H_2 \cdots H_\ell a H_{\ell+1} \cdots H_m$, where each H_j, $1 \leq j \leq m$, is a hump belonging to $C_1(a)$, $C_2(a)$, $C_1(b)$, $C_2(b)$, or C^*. By definition of T_0, every hump N_i occurs after a in T_0. For $j = 0, 1, \ldots, \ell$, let A_j denote the series of humps in T_0 from H_j to H_ℓ, i.e. $A_j = H_jH_{j+1} \cdots H_\ell$.

For $k = 0, 1, \ldots, q$, let B'_k denote the portion of T_0 starting at a, and ending just before the occurrence of N_k. Let B_k be B'_k but with all humps N_i omitted. Thus, if N_i is the j_i^{th} hump of T_0 for $i = 1, \ldots, q$, then

$$B_k = aH_{\ell+1} \cdots H_{j_1-1} H_{j_1+1} \cdots H_{j_2-1} H_{j_2+1} \cdots H_{j_k-1}.$$

Note that B_k is a segment of each of $T_{k-1}, T_k, \ldots, T_q$. The following four claims help us relate the height of T_k to the heights of T_{k-1} and T_{k+1}.

Claim 1 *If there exists $j \leq \ell$ such that $h(H_j) \geq h(N_k)$, then $h(T_{k-1}) \geq h(T_k)$.*

Proof. Since $h(A_j B_k) \geq h(H_j) \geq h(N_k)$, by the first condition of Observation 3

$$h(A_j B_k N_k) \geq h(N_k A_j B_k)$$

. Because $h(H_j) \geq h(N_k) > h(N_{k-1}) > \cdots > h(N_1)$, all of $N_1, \ldots, N_{k-1}$ must be scheduled before H_j in T_{k-1}. It follows that $A_j B_k N_k$ is a segment of T_{k-1}. Note that $N_k A_j B_k$ is a segment of T'_k where the (V_1, V_2)-partition of T'_k is the same as that of T_k. Hence $h(T_{k-1}) \geq h(T'_k) \geq h(T_k)$. □

Claim 2 *If there exists $j \leq \ell$ such that $s(A_j B_k) \geq 0$, then $h(T_{k-1}) \geq h(T_k)$.*

Proof. Since $s(A_j) + s(B_k) = s(A_j B_k) \geq 0$ and $s(B_k) < 0$, we have $s(A_j) > 0$. This implies that at least one P-hump exists in A_j. Let $H_{j'}$ be the first P-hump in A_j. We know that $s(A_{j'}) \geq s(A_j)$. Hence $s(A_{j'} B_k) \geq 0$. By the second condition of Observation 3, we have $h(A_{j'} B_k N_k) \geq h(N_k A_{j'} B_k)$. Because $H_{j'}$ is a P-hump, all of $N_1, \ldots, N_{k-1}$ must be scheduled before $H_{j'}$ in T_{k-1}. It follows that $A_{j'} B_k N_k$ is a segment of T_{k-1}. Note that $N_k A_{j'} B_k$ is a segment of T'_k where the (V_1, V_2)-partition of T'_k is the same as that of T_k. Hence $h(T_{k-1}) \geq h(T'_k) \geq h(T_k)$. □

Claim 3 *If for every $1 \leq j \leq \ell$, $h(H_j) < h(N_k)$ and $s(A_j B_k) < 0$, then $h(T_{k-1}) \leq h(T_k)$.*

Proof. There is a $1 \leq j' \leq \ell$ such that $A_{j'} B_k N_k$ is a segment of T_{k-1} and $N_k A_{j'} B_k$ is a segment of T_k. Since $s(A_{j'} B_k) < 0$, if we can prove that $h(N_k) > h(A_{j'} B_k)$, then $h(T_{k-1}) \leq h(T_k)$ by the first condition of Observation 3. Note that $h(A_{j'} B_k) = \max\{h(A_{j'}), s(A_{j'}) + h(B_k)\}$. Let us show that $h(N_k)$ is actually greater than both elements of the maximum.

Suppose H_{j^*} is the hump containing the last peak of $A_{j'}$. Namely $h(A_{j'}) = s(H_{j'}) + s(H_{j'+1}) + \cdots + s(H_{j^*-1}) + h(H_{j^*})$. By the third hump decomposition property, H_{j^*} cannot be later than the first P-hump. This implies that none of $H_{j'}, H_{j'+1}, \ldots, H_{j^*-1}$ is a P-hump. Thus $s(H_{j'}) + s(H_{j'+1}) + \cdots + s(H_{j^*-1}) \leq 0$. It follows that $h(N_k) > h(H_{j^*}) \geq h(A_{j'})$.

By the second hump decomposition property and the fact that B_k is composed of N-humps, we have $h(N_k) > \tilde{h}(B_k) = h(B_k) - s(B_k)$. Since $s(A_{j'}) + s(B_k) = s(A_{j'} B_k) < 0$, it follows that $h(N_k) > h(B_k) + s(A_{j'})$. □

Claim 4 *If for every $1 \leq j \leq \ell$, $h(H_j) < h(N_k)$ and $s(A_j B_k) < 0$, then for every $1 \leq j \leq \ell$, we have $h(H_j) < h(N_{k+1})$ and $s(A_j B_{k+1}) < 0$.*

Proof. Since N_{k+1} is a N-hump, $h(H_j) < h(N_k) < h(N_{k+1})$. Note that

$$B_{k+1} = B_k H_{j_k+1} H_{j_k+2} \cdots H_{j_{k+1}-1},$$

where $H_{j_k+1}, H_{j_k+2}, \ldots, H_{j_{k+1}-1}$ are all N-humps. It follows that $s(B_{k+1}) \leq s(B_k)$. Hence $s(A_j B_{k+1}) \leq s(A_j B_k) < 0$. □

Now we can prove Lemma 7. Let $1 \le k \le q$ be the least number such that for every $1 \le j \le \ell$, $h(H_j) < h(N_k)$ and $s(A_j B_k) < 0$. By Claim 4, for every $k < i \le q$, $h(H_j) < h(N_i)$ and $s(A_j B_i) < 0$. By Claim 3 we know

$$h(T_{k-1}) \le h(T_k) \le h(T_{k+1}) \le \cdots \le h(T_q). \tag{3}$$

Note that for every $0 \le i \le k-1$, either $h(H_j) \ge h(N_i)$ or $s(A_j B_i) \ge 0$. By Claim 1 and 2,

$$h(T_0) \ge h(T_1) \ge \cdots \ge h(T_{k-2}) \ge h(T_{k-1}). \tag{4}$$

It follows from (3) and (4) that the boundary between N_{k-1} and N_k is one of the best breakpoints among N-humps of C^*. Note that if such a k does not exist, then the boundary right after N_q is as least as good as all the other boundaries among N-humps.

Let $H_{j'}$ be the highest hump among all H_j's, $1 \le j \le \ell$. In particular, if $h(H_{j'}) < h(N_k)$ then $h(H_j) < h(N_k)$ for all $1 \le j \le \ell$. Let $H_{j''}$ be the hump that $s(H_1 H_2 \cdots H_{j''-1})$ is the smallest. Since $s(H_1 H_2 \cdots H_\ell)$ is fixed, $s(A_{j''})$ is the largest among all A_j's. This implies that if $s(A_{j''} B_k) < 0$ then $s(A_j B_k) < 0$ for all $1 \le j \le \ell$. In order to locate the best breakpoint between among N-humps of C^*, we just iteratively scan the N-humps. For each N-hump N_k in turn, we test if $h(H_{j'}) < h(N_k)$ and $s(A_{j''} B_k) < 0$ are both true. Since only $q+1$ iterations are required, we need only $O(q)$ time. The lemma follows by using Observation 1 to show that $q = O(\sqrt{n})$.

4 The Algorithm for All Pairs

Recall that G is composed of p chains of n nodes. In this section we shall show how to decide the $\prec$ relation 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 $O(n^3)$ time. We present a faster algorithm, one that runs in time $O(n^{1.5}p)$. The algorithm is outlined in Figure 5. We elaborate it in the following subsections.

Input:	$C_1, C_2, \ldots, C_p$.
Output:	For every C_i and C_j, for every node v in C_i, output the arc vw where w is the first node in C_j such that $v \prec w$.
Step 1	Construct humps for every possible pair of subchains.
Step 2	**For** every $1 \le i \le p$ **do**
Step 3	Build a merge tree T_i^* from all chains except C_i.
Step 4	**For** every $1 \le j \le p$ **do**
Step 5	From T_i^* compute $C_{i,j}^*$, the chain merged from all chains except C_i and C_j.
Step 6	**For** every node v in C_i **do**
Step 7	Find the first node w^* in C_j such that $v \prec w^*$.
Step 8	Output the arc vw^*.

Fig. 5. The $O(n^{1.5}p)$-time algorithm of computing the $\prec$ relation among every pair of nodes.

Step 1: Let ℓ_i be the number of nodes in C_i. For every chain C_i, we construct humps for every possible pair of subchains C_i' and C_i'' where $C_i = C_i' C_i''$. There are two passes, one for C_i' and the other for C_i''. The first pass is as follows. To initialize, let C_i' be the hump composed of the first node of C_i. We then execute a loop of $\ell_i - 1$ iterations. In the k^{th} iteration, we add the $(k+1)^{st}$ node of C_i to C_i', and then reconstruct humps for C_i'. The second pass is very similar to the first one. Initially let C_i'' be the hump composed of the last node of C_i. In the k^{th} iteration, we add the $(k+1)^{st}$ last node of C_i to C_i'', and then reconstruct humps for C_i''.

Since the lengths of C_i' and C_i'' are always less than or equal to ℓ_i, the number of humps constructed in every iteration is always bounded by $O(\sqrt{\ell_i})$. Whenever a new node is added to C_i', it takes only $O(\sqrt{\ell_i})$ time to reconstruct humps for the lengthened subchain. It follows that the time complexity of Step 1 is $\sum_{1 \leq i \leq p} O(\ell_i \sqrt{\ell_i}) = O(n^{1.5})$.

Step 3: In this step the algorithm constructs a rooted binary tree of chains to help implement step 5. The leaves of the tree are all the chains except the i^{th} (in hump representation). Every nonleaf vertex is (the hump representation of) the chain resulting from merging the vertex's children. Thus the root is the chain resulting from merging all the original p chains except the i^{th}.

By the same analysis as in Subsection 3.1, building a merge tree takes $O(\sqrt{np})$ time. Note that the humps of every chain in the merge tree are constructed at the same time. The hump representation uses $O(1)$ space for every constructed hump. Since there are $O(\sqrt{np})$ humps in this merge tree, the space used for the tree is only linear.

Step 5: For a given i and j, we must compute C_{ij}^*. It is easy to see that there is a set of $O(\log p)$ vertices of the binary tree constructed in step 2 such that the leaves below these vertices comprise all chains except the i^{th} and the j^{th}. We compute the amortized complexity of Step 5. Fix a value of i. In step 3 the algorithm constructed a merge tree whose leaves were all the chains except the i^{th}. We use this merge tree in executing step 5, namely computing C_{ij}^*, for $j = 1, \ldots, p$. We show that the total time required is $O(p\sqrt{n})$. Hence the amortized complexity of step 5 is $O(\sqrt{n})$.

For each value of j, we recompute the vertices of the merge tree whose chains include the j^{th} chain, working from the leaf to the root. Once the root has been recomputed, we obtain the chain merged from all chains but the i^{th} and the j^{th}, i.e. C_{ij}^*. Thus the root gets recomputed a total of p times, each of its children gets recomputed $p/2$ times, and so on. In general, each vertex at level k gets recomputed $p/2^k$ times.

Now we evaluate the sum of times required for recomputing the vertices at level k. The total time required is proportional to the number of humps in chains belonging to vertices at level $k+1$. The number of chains at level $k+1$ is 2^{k+1}, so the number of humps is $O(\sqrt{n2^{k+1}})$. This, then, is the time required for recomputing all the vertices at level k, once each. Since each vertex at level k gets recomputed $p/2^k$ times, the total time invested in recomputing vertices at level k is $O((p/2^k)\sqrt{n2^{k+1}})$, which is $O(p\sqrt{n/2^k})$. Finally, we sum over all levels k, obtaining $O(p\sqrt{n})$, as required.

Steps 6–8: Our goal is to determine for each v in C_i the first node w^* in C_j such that $v \prec w$. For the recursion, we address a more complicated goal, in which we are given an interval in which to seek w^*. Namely, given a node v in C_i, and given two nodes x and y of C_j such that w^* is guaranteed to be between them, find w^*.

First we give a recursive procedure BINSEARCH(v, x, y) based on binary search. Let w be the node of C_j located midway between x and y. We use the procedure MINSCHED of Section 3 to determine the value of (1) where the minimum is taken over all proper partitions (V_1, V_2) where $w \in V_1$ and $y \in V_2$. If the value is zero, we conclude that there is a valid schedule in which w precedes v, and hence $v \not\prec w$. It follows that w^* is located between w and y. We therefore recurse on this interval.

If the value is positive, we claim that there is no valid schedule in which w precedes v, i.e. no schedule in which w precedes v whose maximum cumulative cost is zero. For suppose there were such a schedule S. By our assumption that w^* lies between x and y, we have $v \prec y$, so there is no valid schedule in which y precedes v. In particular, v precedes y in S. Let V_1 be the set of nodes preceding v in S, and let V_2 be the set of nodes following v. Then (V_1, V_2) is a proper partition where $w \in V_1$ and $y \in V_2$, and we have that $h(V_1, V_2)$ is no more than the cost of S, which is zero. This contradiction proves our claim. Hence $v \prec w$, and we can recurse on the interval between x and w.

The time required by MINSCHED is $O(\sqrt{n\ell})$, where ℓ is the number of nodes in the interval from x to y. The length of that interval is halved in each recursion, so the time required by BINSEARCH is a geometric series and hence is also $O(\sqrt{n\ell})$.

If Step 6 were just a simple FOR-loop, then the time complexity of Step 6–8 would be $O(\ell_i\sqrt{n\ell_j})$. By using divide-and-conquer for Step 6–8, however, we achieve time $O(\sqrt{n\ell_i\ell_j})$ as follows. For the recursion, we are given an interval $[p, q]$ of C_i and an interval $[x, y]$ of C_j. For every v in the first interval, we need to find the corresponding w^*. We are given that for every such v, the corresponding w^* lies in the interval $[x, y]$ of C_j. The procedure is as follows. Let v be the node of C_j midway between p and q, and call BINSEARCH(v, x, y) to find the corresponding w^*. It then follows that for every v' between p and v, the corresponding w'^* lies between x and w^*, and for every v' between v and q, the corresponding w'^* lies between w^* and y. We therefore recurse with these intervals. Let ℓ_i be the number of nodes in the interval $[p, q]$, and let ℓ_j be the number of nodes in the interval $[x, y]$. The recurrence for the time required by the procedure is

$$\begin{aligned} T(1, \ell_j) &\le c\sqrt{n\ell_j} \\ T(\ell_i, \ell_j) &\le c\sqrt{n\ell_j} + \max_{0\le\ell\le\ell_j}\{T(\ell_i/2, \ell) + T(\ell_i/2, \ell_j - \ell)\}. \end{aligned}$$

It can be proved that therefore the time complexity is $O(\sqrt{n\ell_i\ell_j})$.
Overall Complexity: The complexity of the algorithm is dominated by the time for steps 6–8, which is $\sum_{i,j} O(\sqrt{n\ell_i\ell_j})$. This can be shown to be $O(n^{1.5}p)$, proving Theorem 2.

References

[1] ———, "Scheduling with Application to Register Allocation and Deadlock Problems," University of Waterloo, PhD Thesis, 1976.

[2] Abdel-Wahab, H. M. & Kameda, T., "Scheduling to Minimize Maximum Cumulative Cost Subject to Series-parallel Precedence Constraints," *Operations Research* 26 (1978), 141–158.

[3] ———, "On Strictly Optimal Schedules for the Cumulative Cost-Optimal Scheduling Problem," *Computing* 24 (1980), 61–86.

[4] Emrath, P. A., Ghosh, S. & Padua, D. A., "Event Synchronization Analysis for Debugging Parallel Programs," *Supercomputing '89* (November 1989), 580–588.

[5] Garey, M. R. & Johnson, D. S., *Computers and Intractability—A Guide to the Theory of NP-Completeness*, W. H. Freeman and Company, 1979.

[6] Helmbold, D. P. & McDowell, C. E., "A Class of Synchronization Operations that Permit Efficient Race Detection," *University of California at Santa Cruz Technical Report* (January 1993).

[7] Helmbold, D. P., McDowell, C. E. & Wang, J-Z., "Analyzing Traces with Anonymous Synchronization," *International Conference on Parallel Processing* (August 1990), II70–II77.

[8] Netzer, R. H. B. & Ghosh, S., “Efficient Race Condition Detection for Shared-Memory Programs with Post/Wait Synchronization," *International Conference on Parallel Processing* (August 1992), II242–II246.

[9] Netzer, R. H. B. & Miller, B. P., “On the Complexity of Event Ordering for Shared-Memory Parallel Program Executions," *International Conference on Parallel Processing* (August 1990), II93–II97.

Appendix: Omitted Proofs

This part does not appear in the WADS paper.

Lemma 3 *Let G' be an acyclic graph with costs on nodes and H be a hump in 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 $h(T) \leq h(S)$.*

Proof. The original lemma in [2] restricts G' to be a chain graph. We can prove as follows that the same properties hold even without the restriction. Suppose $H = v_1 \cdots v_p \cdots v_d$, where v_p is a useful peak of H. Suppose $w_1 w_2 \cdots w_\ell$ is the segment of S such that $w_{j_i} = v_i$ for all $1 \leq i \leq d$ and $1 = j_1 < j_2 < \cdots < j_d = \ell$. The only difference between S and T is that the segment of $W = w_1 w_2 \cdots w_\ell$ in S becomes

$$W' = w_{j_1+1} \cdots w_{j_2-1} w_{j_2+1} \cdots w_{j_p-1} H w_{j_p+1} \cdots w_{j_{p+1}-1} w_{j_{p+1}+1} \cdots w_{j_d-1}$$

in T. Since H is a chain in G', there is no precedence relation imposed by G' between w_i and w_j if $w_i \in H$ and $w_j \notin H$; otherwise S cannot be be a schedule of G'. Hence T is also a schedule of G'.

We denote the heights of w_i in S and T by $h_S(w_i)$ and $h_T(w_i)$, respectively. Note that $h_S(v_p) = h_T(v_p)$ since the set of nodes preceding v_p does not change. Let us show that for every $1 \leq \alpha \leq \ell$ there exists an $1 \leq \beta \leq \ell$ such that $h_T(w_\alpha) \leq h_S(w_\beta)$.

- If $\alpha = j_i$ for some $1 \leq i \leq d$, then $h_T(w_\alpha) = h_T(v_i) \leq h_T(v_p) = h_S(v_p) = h_S(w_{j_p})$.
- If $j_i < \alpha < j_{i+1}$ for some $p \leq i < d$, then $h_T(w_\alpha) = s(v_{i+1}) + s(v_{i+2}) + \cdots + s(v_d) + h_S(w_\alpha)$. Since H is a hump and $i \geq p$, $s(v_{i+1}) + s(v_{i+2}) + \cdots + s(v_d) = h_S(v_d) - h_S(v_i) \leq 0$. Thus $h_T(w_\alpha) \leq h_S(w_\alpha)$.
- If $j_i < \alpha < j_{i+1}$ for some $1 \leq i < p$, then $h_T(w_\alpha) = -s(v_1) - s(v_2) - \cdots - s(v_i) + h_S(w_\alpha)$. Since H is a hump and $i < p$, $-s(v_1) - s(v_2) - \cdots - s(v_i) = h_S(v_0) - h_S(v_i) \leq 0$, where v_0 is the node precedes v_1 in S. Thus $h_T(w_\alpha) \leq h_S(w_\alpha)$.

It follows that the height of $h(W') \leq h(W)$. Since the nodes other than w_i's preserve their heights in S and T, this lemma is proved. □

Lemma 4 *Let G' be an acyclic graph with node costs. Let A, B, S_1, S_2, and S_3 be subsequences of nodes in G'. Suppose $S = S_1 A S_2 B S_3$, $T_1 = S_1 B A S_2 S_3$, and $T_2 = S_1 S_2 B A S_3$. If the series of BA is in standard form then either $h(S) \geq h(T_1)$ or $h(S) \geq h(T_2)$.*

Proof. The original version of this lemma in [2] restricts A and B to be humps, and restricts S, T_1, and T_2 to be schedules of G'. We can prove as follows that the same property holds even if without these restrictions. Since the heights of nodes in S_1 and S_3 are not changed in S, T_1, and T_2, it suffices to ensure that

$$h(AS_2B) \geq h(BAS_2) = \max\{h(B), s(B) + h(A), s(BA) + h(S_2)\} \quad \text{or}$$
$$h(AS_2B) \geq h(S_2BA) = \max\{h(S_2), s(S_2) + h(B), s(S_2B) + h(A)\},$$

where $h(AS_2B) = \max\{h(A), s(A) + h(S_2), s(AS_2) + h(B)\}$.

- If $s(A) < 0$ and $s(B) < 0$ then $h(A) > s(B) + h(A)$ and $s(A) + h(S_2) > s(BA) + h(S_2)$. Since the series of BA is in standard form, $h(A) \geq h(B)$. Thus $h(AS_2B) \geq h(BAS_2)$.

- If $s(A) > 0$ and $s(B) > 0$ then $s(A) + h(S_2) \geq h(S_2)$ and $s(AS_2) + h(B) \geq s(S_2) + h(B)$. Since the series of BA is in standard form, $\tilde{h}(B) \geq \tilde{h}(A)$. It follows that $h(B) - s(S) \geq h(A) - s(A)$ and thus $s(A) + h(B) \geq s(B) + h(A)$. Now that $s(AS_2) + h(B) \geq s(S_2B) + h(A)$, we have $h(AS_2B) \geq h(S_2BA)$.
- Suppose $s(A) \geq 0$ and $s(B) \leq 0$.
  - If $s(S_2) \geq 0$ then $h(A) \geq s(B) + h(A)$, $s(A) + h(S_2) \geq s(BA) + h(S_2)$, and $s(AS_2) + h(B) \geq h(B)$. It follows that $h(AS_2B) \geq h(BAS_2)$.
  - If $s(S_2) < 0$ then $s(AS_2) + h(B) \geq s(S_2) + h(B)$, $h(A) > s(S_2B) + h(A)$, and $s(A) + h(S_2) > h(S_2)$. It follows that $h(AS_2B) \geq h(S_2BA)$. □

In section 4 we have shown that the recurrence of the time complexity for steps 6–8 is

$$\begin{aligned} T(1, \ell_j) &\leq c\sqrt{n\ell_j} \\ T(\ell_i, \ell_j) &\leq c\sqrt{n\ell_j} + \max_{0 \leq \ell \leq \ell_j} \{T(\ell_i/2, \ell) + T(\ell_i/2, \ell_j - \ell)\}. \end{aligned}$$

It can be shown that $T(\ell_i, \ell_j) = O(\sqrt{n\ell_i\ell_j})$ by proving

$$T(\ell_i, \ell_j) \leq c\sqrt{n\ell_i\ell_j} + c\sqrt{n\ell_i\ell_j/2} + c\sqrt{n\ell_i\ell_j/4} + \cdots + c\sqrt{n\ell_j}.$$

Let us prove this inequality by induction on ℓ_i. If for all $1 \leq \ell_i \leq k - 1$ this inequality holds, then

$$\begin{aligned} T(k, \ell_j) &\leq c\sqrt{n\ell_j} + \max_{0 \leq \ell \leq \ell_j} \{T(k/2, \ell) + T(k/2, \ell_j - \ell)\} \\ &\leq c\sqrt{n\ell_j} + \max_{0 \leq \ell \leq \ell_j} \Big\{ c\sqrt{nk\ell/2} + c\sqrt{nk\ell/4} + \cdots + c\sqrt{n\ell} + \\ &\qquad c\sqrt{nk(\ell_j - \ell)/2} + c\sqrt{nk(\ell_j - \ell)/4} + \cdots + c\sqrt{n(\ell_j - \ell)} \Big\} \\ &\leq c\sqrt{n\ell_j} + c\sqrt{nk\ell_j} + c\sqrt{nk\ell_j/2} + c\sqrt{nk\ell_j/4} + \cdots + c\sqrt{2n\ell_j}. \end{aligned}$$

Since the the inequality holds for $\ell_i = k$, it holds for every integer ℓ_i. Therefore $T(\ell_i, \ell_j) = O(\sqrt{n\ell_i\ell_j})$.

By the above analysis, the time complexity of this algorithm is

$$\begin{aligned} & O(n^{1.5}) + \sum_{1 \leq i \leq p} \left(O(\sqrt{np}) + \sum_{1 \leq j \leq p} \left(O(\sqrt{n}) + O(\sqrt{n\ell_i\ell_j}) \right) \right) \\ &= O(n^{1.5}) + O(p\sqrt{np}) + \sum_{1 \leq i \leq p} \sum_{1 \leq j \leq p} O(\sqrt{n\ell_i\ell_j}) \\ &= O(n^{1.5}) + O(p\sqrt{np}) + O(\sqrt{n}) \sum_{1 \leq i \leq p} O(\sqrt{\ell_i}) \sum_{1 \leq j \leq p} O(\sqrt{\ell_j}) \\ &= O(n^{1.5}) + O(p\sqrt{np}) + O(\sqrt{n})(O(\sqrt{np}))^2 \\ &= O(n^{1.5}p). \end{aligned}$$

The space used by this algorithm is $O(n)$. Therefore Theorem 2 is proved.