International Journal of Parallel Programming, Vol. 23, No. 3, 1995

Minimizing Redundant Dependencies and Interprocessor Synchronizations

Heng-Yi Chao[1] and Mary P. Harper[1]

Received May 31, 1994

Run-time synchronization overhead is a crucial factor in limiting speedup for parallel computers. In this paper, we present a new two-phase algorithm for removing redundant dependencies and minimizing interprocessor synchronizations when scheduling an acyclic task graph onto a multiprocessor system. The first phase removes redundant dependencies before scheduling; the second phase eliminates interprocessor synchronizations after scheduling. In a simulation using randomly generated task graphs, on the average, 98.28% of the dependencies are eliminated in the first phase, and 65.86% of the remaining dependencies are eliminated during the second phase, for a total of 99.41% removed. The approach has also been applied to some benchmark task graphs. The two-phase algorithm, which has $O(n^3)$ time complexity and $O(n^2)$ space complexity, utilizes a new algorithm which computes the transitive closure and reduction at the same time, storing results in a single matrix.

KEY WORDS: Multiprocessor scheduling; redundant dependency; synchronization; transitive closure; transitive reduction.

1. INTRODUCTION

Run-time synchronization overhead is a crucial factor in limiting effective speedup when a system of tasks with dependencies among the tasks is executed on a MIMD machine with identical processing elements. In this paper, we present a new two-phase algorithm for removing redundant dependencies and minimizing interprocessor synchronizations when scheduling an acyclic task graph onto a multiprocessor system. The first

[1] School of Electrical Engineering, 1285 Electrical Engineering Building, Purdue University, West Lafayette, Indiana 47907-1285. (E-mail: hengyi@ecn.purdue.edu, harper@ecn.purdue.edu).

0885-7458/95/0600-0245$07.50/0

phase removes redundant dependencies before scheduling; while the second phase, given a scheduler, minimizes interprocessor synchronizations after scheduling.

When dependent tasks are scheduled on different processors, (interprocessor) *synchronizations* are needed to ensure the correct execution of tasks. To achieve interprocessor communication, some multiprocessor systems use shared memory,[1,2] others use message passing.[3–5] To pass values from a task on one processor to a task on another processor introduces contention for shared resources (e.g., the shared memory, common buses). For each pair of dependent tasks scheduled on different processors, one synchronization is required. Removing synchronizations potentially avoids multiple accesses to the shared resources. Obviously, to achieve greater speedup, synchronizations should be removed whenever they are not necessary.

A task system can be characterized by a directed acyclic graph (DAG), G (called a *task graph*), where vertices represent the tasks and arcs represent the dependencies (vertex and task are used interchangeably in this paper).[6–8] Three types of data dependencies can be identified in programs: flow dependency, antidependency, and output dependency.[9] Each required dependency represents a synchronization between tasks. If there is a dependency from task i to task j, then task j cannot start execution until task i has completed execution (and the required data is transferred). It is assumed that the task graph is acyclic; each strongly connected component can be detected and represented by a single vertex.

The problem of minimizing synchronizations can be formalized as shown in Definition 1.

Definition 1. *Minimal Synchronization Problem (MSP)*

- *Given*: (G, m, S), where $G = (V, A)$ is a task graph,
  $V = \{1, \ldots, n\}$ is the set of vertices,
  $A = \{(i, j)\}$ is the set of dependency arcs,
  m is the number of processors,
  S is a scheduler.
- *Objective*: a minimal set of interprocessor synchronizations.
- *Constraint*: if there is an arc (i, j) in G, then task j cannot start execution until task i has completed execution.

The required synchronizations will be a subset of A, the set of dependency arcs. The key to eliminating a redundant dependency is that an arc (i, j) can be eliminated if it is *transitive*, i.e., there exists a path from i to j going through an intermediate vertex k which is not i or j in the task graph.

1.1. Related Work

The multiprocessor scheduling problem has been studied extensively. A major difficulty in multiprocessor scheduling comes from the conflict between load balancing and communication overhead. Earlier work[7,10–14] has focused on minimizing the schedule length. Price and Salama[3] and Magirou and Milis[15] considered minimizing both the schedule length and the communication overhead. It is always possible to remove redundant dependencies without sacrificing the schedule length.

Redundant dependencies can be eliminated both before and after scheduling; however, previous researchers have only utilized one reduction or the other. Li and Abu-Sufah,[9] Midkiff and Padua,[16] and Krothapalli and Sadayappan[8] remove redundant dependencies only before scheduling; whereas, Shaffer[17] removes redundant dependencies only after scheduling.

Li and Abu-Sufah[9] and Midkiff and Padua[16] studied the problem of removing redundant dependencies in simple loops, where a statement is considered as a task. Krothapalli and Sadayappan[8] further considered removing redundant dependencies in multi-dimensional loops, where the innermost loop body is considered as a task.

When a task system is scheduled on a multiprocessor, additional *precedence* relations are introduced by the scheduler between the tasks scheduled on the same processor (the corresponding arcs are called *precedence* arcs). Shaffer[17] developed an algorithm that used the precedence relations to eliminate interprocessor synchronizations after a task system is scheduled. Given dependency and precedence relations between tasks, an adjacency matrix is formed and its transitive closure is computed by using Warshall's algorithm. A dependency arc (i, j) can be deleted if j can be reached through a successor $k \neq j$ of i, requiring $O(n)$ operations for each dependency arc.

1.2. Our Approach and Contributions

Our approach is to remove redundant dependencies in a directed acyclic task graph both before and after scheduling. Since each of the dependencies must be considered by the scheduler, removing redundant dependencies can often result in a more efficient scheduler. For example, consider the task system in Fig. 1. When task 1 is scheduled, the scheduler might check tasks 2, 3, and 4; however, it is not necessary to check tasks 3 and 4 because task 2 (which must precede tasks 3 and 4) has not been scheduled yet. If the arcs (1, 3) and (1, 4) are removed, the scheduler can immediately detect that only task 2 needs to be considered during the next cycle of scheduling.

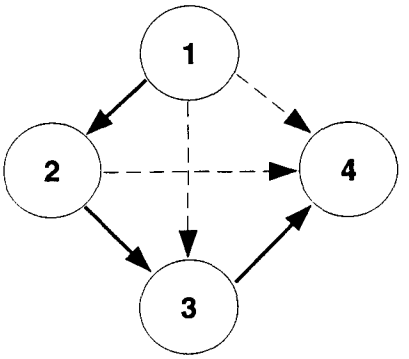

Fig. 1. A complete DAG with four vertices.

Our work is distinct from previous work in several respects:

- We consider eliminating dependencies both before and after scheduling. In a simulation using randomly generated task graphs (see Section 4.1), on the average, 98.28% of the dependencies are eliminated before scheduling. This can have a dramatic impact on the efficiency of the scheduler. And on the average, 99.41% of the dependencies are finally eliminated.
- Our model is valid for any acyclic task graph, which is more general than scheduling nested loops.[8,9,16]
- We develop a new algorithm for jointly computing the transitive closure and reduction of a DAG. The determination of dependency elimination is reduced to $O(1)$ for each arc, as opposed to $O(n)$ in Shaffer's algorithm.[17] The technique is useful for all problems with precedence constraints (e.g., the pipeline scheduling problems,[18,19] superscalar pipeline scheduling problems,[20] microcode compaction problems,[21] and channel routing problems[22]).

The problem under consideration is closely related to the transitive closure and transitive reduction problems.[23–28] In Section 2, we develop a new algorithm which computes the transitive closure and transitive reduction of a DAG simultaneously. In Section 3, we introduce a new algorithm for solving the minimal synchronization problem. The experimental results are shown in Section 4, and conclusions are drawn in Section 5.

2. TRANSITIVE CLOSURE AND TRANSITIVE REDUCTION

The *transitive closure*, denoted by G^+, of a directed graph (or *digraph*) G is obtained by adding an arc (i, j) if there is a path from i to j

in G.[24,25,27,28] The transitive closure of a graph indicates the *reachability* of all pairs of vertices in a graph. Two graphs G and G' are said to be *transitively equivalent* if they have the same transitive closure. Such graphs have the interesting property that a *schedule* satisfying the precedence relations in one graph satisfies the precedence relations in the other. The *transitive reduction*, denoted by G^-, of G is obtained by removing as many arcs as possible, such that the resulting graph is transitively equivalent to G.

The transitive reduction of a graph gives the most concise representation of the set of precedence constraints that must be satisfied by a feasible solution. This is especially useful for exhaustive search algorithms[29,30] since many unnecessary arcs can be removed before scheduling. Note that the reduction can be substantial. Consider a *complete* DAG with n vertices (e.g., Fig. 1 is a complete DAG with $n=4$); there are $n(n-1)/2$ arcs, but there are only $n-1$ arcs in the reduced graph.

For any digraph, its transitive closure can be uniquely determined.[23,24,27] Finding the transitive reduction of an arbitrary digraph is NP-complete[26,28]; however, if the digraph is acyclic, then its transitive reduction can be uniquely determined in polynomial time.[23,27]

Warshall's transitive closure algorithm[24,25] finds the transitive closure of a matrix that represents a binary relation in $O(n^3)$ time with n being the number of vertices. Gries *et al.*[27] proposed a transitive reduction algorithm for an initially closed DAG by inverting Warshall's transitive closure algorithm. Note that the transitive closure must be computed before computing the transitive reduction, which is needed to eliminate redundant dependencies. We will classify arcs in a DAG into transitive arcs and direct arcs as in the following definition:

Definition 2. *Transitive Arc and Direct Arc.* An arc (i, j) in G is called a *transitive* arc if there exists $k \neq i, j$ such that there is a path from i to k and there is a path from k to j in G. Otherwise the arc is a *direct* arc.

The transitive closure is formed by adding transitive arcs, while the transitive reduction is formed by removing transitive arcs. Note that all direct arcs must appear in the original graph, but not all arcs in the original graph are necessarily direct arcs.

2.1. An Algorithm for Jointly Computing the Transitive Closure and Transitive Reduction for a DAG

In this section, we present an efficient algorithm which is able to compute the transitive closure and transitive reduction for a DAG

simultaneously. Warshall's algorithm for transitive closure is shown on the left in Fig. 2.

The original graph G is represented by an adjacency matrix A such that $a_{ij} = 1$ if and only if there is an arc (i, j) in G. The transitive closure is represented by a matrix C such that $c_{ij} = 1$ if and only if there is a path from i to j in G. We have augmented the algorithm with two additional statements (underlined) to develop our transitive closure/transitive reduction algorithm, without changing the results of the algorithm.

Theorem 1. When the Augmented Warshall's algorithm terminates, C is the matrix representing the transitive closure of A. If G^+ is a graph that has an arc (i, j) whenever $c_{ij} = 1$, then G^+ is the transitive closure of G.

Proof. See Baase.[24] □

Lemma 1. At any time in the Augmented Warshall's algorithm, $c_{ij} = 1 \Leftrightarrow r_{ij} > 0$.

Proof. It follows because $r_{ij} = c_{ij}$ initially, and r_{ij} is set to 2 whenever c_{ij} is set to 1. □

Our algorithm, which jointly computes the transitive closure and transitive reduction for a DAG (call it TC/TR), is shown on the right in Fig. 2. The time complexity of TC/TR is $O(n^3)$. Note that in step 5 of TC/TR, r_{ij} is assigned a positive number which is not equal to 1 to indicate that (i, j) is a transitive arc, not a direct arc.

Theorem 2. When the TC/TR algorithm terminates, R represents the transitive closure of G. If G^+ is a graph that has an arc (i, j) whenever $r_{ij} > 0$, then G^+ is the transitive closure of G.

```
Algorithm Augmented Warshall(A)
 1. C := A, R := A
 2. for k = 1 to n do
 3. for i = 1 to n do
 4. for j = 1 to n do
 5.     if c_ik = 1 and c_kj = 1 then
 6.         c_ij := 1, r_ij := 2
 7.     endif
```

```
Algorithm TC/TR(R)
 1. for k = 1 to n do
 2. for i = 1 to n do
 3. for j = 1 to n do
 4.     if r_ik > 0 and r_kj > 0 then
 5.         r_ij := 2
 6.     endif
```

Fig. 2. The augmented Warshall's algorithm and the TC/TR algorithm. TC/TR computes the transitive closure and transitive reduction for an adjacency matrix R. Note that R is passed by reference.

Proof. By Lemma 1, line 5 in the Augmented Warshall's algorithm can be replaced by:

$$\text{if } r_{ik} > 0 \text{ and } r_{kj} > 0 \text{ then}$$

C is the matrix representing the transitive closure of A when the Augmented Warshall's algorithm terminates by Theorem 1. By Lemma 1, $c_{ij} = 1 \Leftrightarrow r_{ij} > 0$. Hence, R represents the transitive closure of G if $r_{ij} > 0$, denoting the presence of a path in G. The C matrix is no longer necessary in TC/TR. □

Theorem 3. When the TC/TR algorithm terminates, R represents the transitive reduction of the DAG G. If G^- is a graph that has an arc (i, j) whenever $r_{ij} = 1$, then G^- is the transitive reduction of G.

Proof. When TC/TR terminates, $r_{ij} = 2$ if and only if there exists k, $r_{ik} > 0$ and $r_{kj} > 0$, that is, there is a path from i to k and there is a path from k to j in G. Hence (i, j) is a transitive arc in G^+ and should be removed to form the transitive reduction. On the other hand, if $r_{ij} = 1$ when TC/TR terminates, then (i, j) is an arc in G, and there does not exist any path from i to j going through an intermediate vertex k. Hence, (i, j) is a direct arc in G and should be retained in the transitive reduction. Consequently, if G^- is a graph that has an arc (i, j) whenever $r_{ij} = 1$, then G^- is the transitive reduction of G. □

3. MINIMIZING INTERPROCESSOR SYNCHRONIZATIONS

In this section, we present a new heuristic algorithm for the minimal synchronization problem. An efficient two-phase algorithm, Min-Sync, that finds the minimal set of interprocessor synchronizations required for a static schedule is shown in Fig. 3.

Algorithm `Min-Sync(G,m,S)`

1. **for** each arc (i,j) in G, set $r_{ij} := 1$.
2. **call** `TC/TR(R)` (phase-1).
3. **call** S to find a schedule by considering (i,j) as an arc if $r_{ij} = 1$ (assume task i is assigned to processor p_i at time t_i).
4. sort tasks in nondecreasing order into a list using p_i as the primary key and t_i as the secondary key.
5. **for** each pair of consecutive elements i,j in the list, set $r_{ij} := 2$ **if** $p_i = p_j$.
6. **call** `TC/TR(R)` (phase-2).
7. **for** each arc (i,j) in G, delete it if $r_{ij} \neq 1$.

Fig. 3. A two-phase algorithm for the minimal synchronization problem.

In the first phase of Min-Sync, redundant dependencies are eliminated before scheduling; while in the second phase, unnecessary interprocessor synchronizations are removed by using the precedence relations introduced by the schedule. When the algorithm Min-Sync terminates:

- if $r_{ij} = 2$, then (i, j) is either a precedence arc or a transitive arc, and is not a required dependency.
- if $r_{ij} = 0$, then there is no arc from i to j.
- if $r_{ij} = 1$, then (i, j) is a dependency arc, and there is no other path from i to j in G. This arc must be retained and denotes a required synchronization.

The time complexity of Min-Sync is $O(n^3)$ and the space complexity is $O(n^2)$ where n is the number of vertices. Note that in Min-Sync, the addition of precedence arcs (Steps 4, 5) is performed in $O(n \log n)$ time and the deletion of each redundant dependency arc (Step 7) is determined in $O(1)$ time; while the time complexity is $O(n^2)$ and $O(n)$ respectively in Shaffer's algorithm.[17]

Note that the scheduler S in Min-Sync can be arbitrary. In our simulations, we have chosen the Highest-Level-First (HLF) algorithm[12,13,31] (see Fig. 4). The *level* of a vertex i in G is defined as:

$$l_i := \begin{cases} 1 & \text{if } i \text{ does not have successors in } G \\ \max\{l_j : j \text{ is a child of } i\} + 1 & \text{otherwise} \end{cases} \quad (1)$$

We say that a task is *ready* at time t if all of its parents are scheduled at a time earlier than t.

```
Algorithm HLF(G,m)
 1. compute the level l_i for each task i
 2. t := 0
 3. let Q be the set of unscheduled ready tasks at t
 4. if Q is empty, then return
 5. k := 0
 6. while k < m and Q is not empty
 7.     i = the task in Q with highest level (ties are broken arbitrarily)
 8.     schedule i at time t on processor k and remove i from Q
 9.     k := k + 1
10. end
11. t := t + 1, goto step 3
```

Fig. 4. A Highest-Level-First scheduling algorithm, where m is the number of processors and l_i is computed by Eq. (1).

The HLF algorithm has a running time which is linear in the number of arcs in the graph. Hence it is desirable for Min-Sync to eliminate as many redundant arcs as possible prior to scheduling. Furthermore, other scheduling algorithms could require an even greater running time. Hence, we believe that phase one reduction is a useful computation step in Min-Sync, especially since the overall time complexity is unchanged.

An example of the reduction process for scheduling a task graph on three processors ($m = 3$) by using the Min-Sync algorithm is shown in Fig. 5. The original graph has 13 arcs as shown in (a). In (b), two transitive arcs (3, 9) and (3, 10) are removed. The schedule obtained by using the HLF algorithm is shown in (c). In the first cycle of the HLF algorithm, $Q = \{1, 2, 3, 4, 6\}$, $l_2 = l_3 = 3$, $l_1 = l_4 = l_6 = 2$. The scheduler first chose 2 and 3, and then chose 6 arbitrarily. Hence, $\{2, 3, 6\}$ are scheduled in the first cycle. After the schedule is imposed on the graph, the precedence arcs (the dotted arcs) from the schedule are added to the graph. These arcs are not synchronizations. Note that three arcs (1, 8), (3, 5), (2, 7) in (b) can be removed by imposing the schedule in (c). Finally, the TC/TR algorithm is applied again to G, and arcs (2, 8), (6, 10), (7, 10) are removed in (d). Only five interprocessor synchronizations are required.

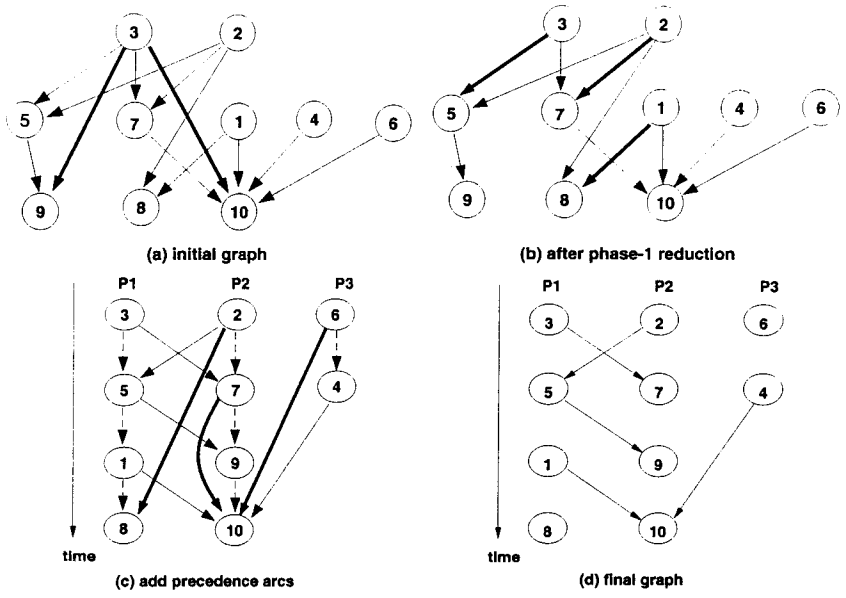

Fig. 5. The reduction process of a task graph by using the Min-Sync algorithm: (a) original graph, (b) after removing transitive arcs in phase-1, (c) after considering the precedence constraints (the dotted arcs), (d) final graph. The schedule in (c) is obtained by using the HLF algorithm. Note that tasks 2, 3, and 6 may not be finished at the same time.

4. EXPERIMENTAL RESULTS AND ANALYSIS

To test the effectiveness of our approach, we have performed a set of simulations on randomly generated DAGs and the task graphs described in Price's paper[3] using the Min-Sync algorithm and the HLF algorithm for scheduling.

Let $|G|$ be the number of arcs in G, $|G'|$ be the minimal number of synchronizations required in the final graph, and $|G^+|$ be the number of arcs in G^+. For a graph G, define the *reduction rate*, $R(G)$, and *maximal reduction rate*, $R^*(G)$, as follows:

$$R(G) := (|G| - |G'|)/|G|, \qquad R^*(G) := (|G^+| - |G'|)/|G^+|$$

Note that $R(G) \leqslant R^*(G)$ because $|G| \leqslant |G^+|$. For a complete DAG, G, with n vertices, $R(G) = R^*(G) \geqslant 1 - 2/n$ since there are $n(n-1)/2$ arcs in G and the number of arcs in G' is less than or equal to $n-1$ (there are $n-1$ arcs in G^-).

For any graph G generated, the reduction rate indicates the gain obtained by this reduction process. There are many graphs that are transitively equivalent to G. The maximal reduction rate indicates the largest possible gain that can be obtained by this reduction process on these graphs.

4.1. Simulation on Randomly Generated DAGs

We have performed simulations on randomly generated DAGs. A random DAG generator is shown in Fig. 6, where n is the number of vertices, p is the arc occurrence probability, random( ) is a random number generator that generates random numbers in (0, 1), and A is a matrix representing the DAG with $a_{ij} = 1$ if and only if there is an arc in the graph from i to j. Note that the graph represented by A is acyclic since a_{ij} can be set to 1 only if $i < j$. In Step 5, vertices are renumbered and A is modified accordingly to create more randomness.

In our simulation, n ranges from 50 to 500 with an increment of 50, p ranges from 0.1 to 1 with an increment of 0.1, and m is 2, 4, or 8. For

```
Algorithm DAG-Generator(A,n,p)
1. a_ij := 0, ∀i,j
2. for i = 1 to n - 1 do
3. for j = i + 1 to n do
4.     if random() ≤ p then a_ij := 1
5. randomly renumber the vertices and modify A accordingly
```

Fig. 6. A random DAG generator.

each (n, p)-pair, ten random instances (DAGs) are generated (1000 DAGs in total). Hence, each data point in the figures represents the average of the results of ten random instances. For each instance, the program is run for $m = 2$, 4, and 8. The average number of arcs in the initial graph is shown in Fig. 7 (100 data points). The average number of arcs after phase-1 reduction is shown in Fig. 8 (100 data points). The number of arcs in the final graph is different when different numbers of processors are used. The average number of arcs after phase-2 reduction for $m = 2, 4, 8$ is shown in Fig. 9 (100 data points each). The minimum (Min), average (Mean), and maximum (Max) over the data points in each of the figures are listed in Table I (where $|G|$ is depicted in Fig. 7, $|G^-|$ in Fig. 8, and $|G'|$ in Fig. 9). On the average, 98.28% (25920/26374) of the initial arcs are eliminated in phase-1; and 65.86% of the remaining dependencies are eliminated during phase-2, for a total of 99.41% (26219/26374) of the initial arcs removed (or 0.59% of the initial arcs are required synchronizations). The average reduction rate and maximal reduction rate for $m = 2, 4, 8$ are shown in Fig. 10. Note that the performance could have been different if a different scheduler had been chosen.

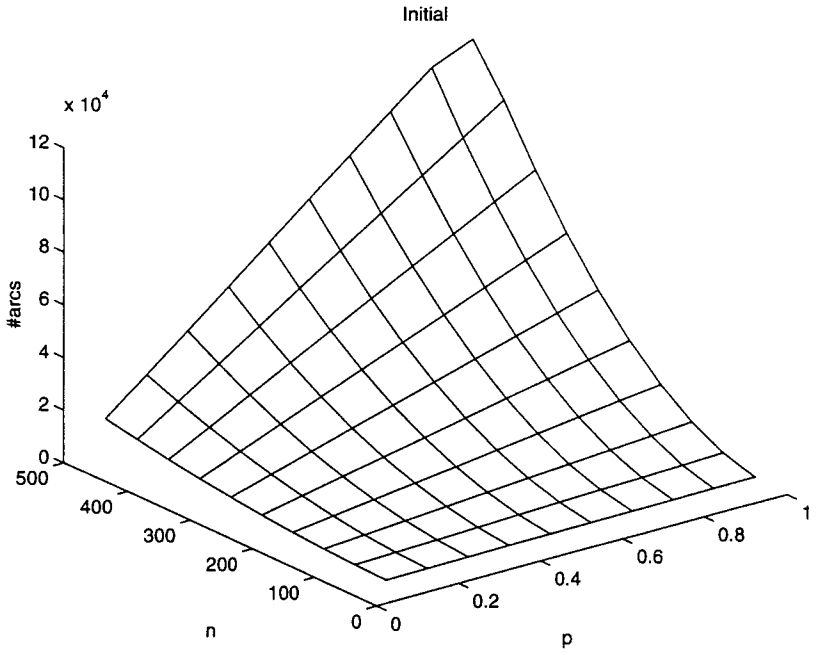

Fig. 7. Average number of initial arcs.

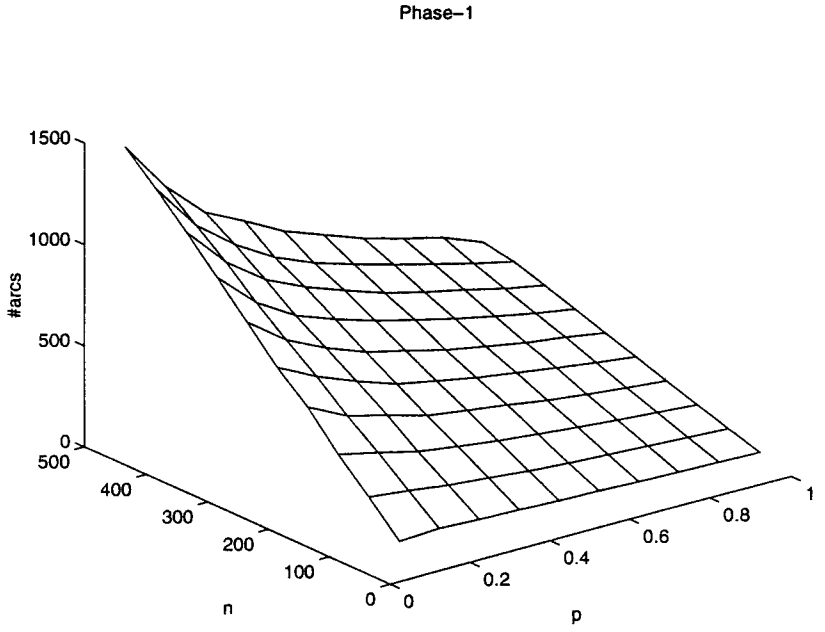

Fig. 8. Average number of direct arcs after phase-1 reduction.

From the simulation results (Figs. 7–10), some observations can be drawn:

- The number of arcs increases as n increases in the initial graph, after phase-1 reduction, and in the final graphs.
- The number of arcs in the initial graph increases as p increases (see Fig. 7). However, the number of arcs after phase-1 reduction decreases slowly as p increases (see Fig. 8).

Table I. The Minimum, Mean, and Maximum Number of Arcs in G, G^+, G^-, and G', Where $|G^-|$ is the Number of Arcs After Phase-1 Reduction, and $|G'|$ is the Final Number of Synchronizations Required

	Min	Mean	Max	Comment
$\|G\|$	121.75	26374	124750	initial graph G
$\|G^+\|$	420.2	46728	124750	transitive closure of G
$\|G^-\|$	49	454	1430	transitive reduction of G
$\|G'\|$	0	155	780	final graph

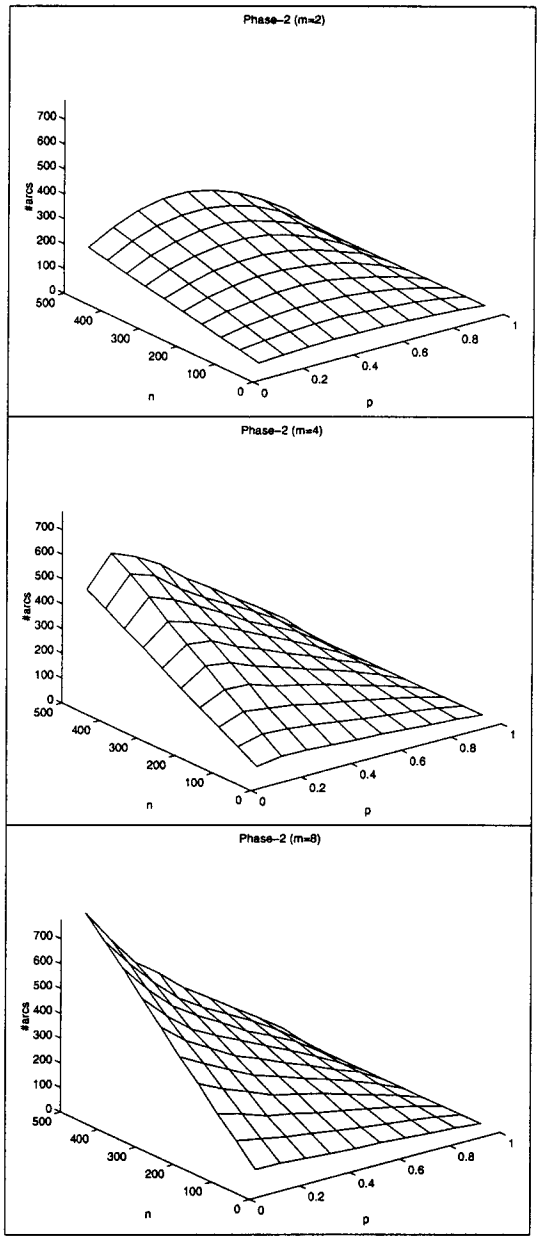

Fig. 9. Average number of direct arcs after phase-2 reduction for $m = 2$, 4, and 8.

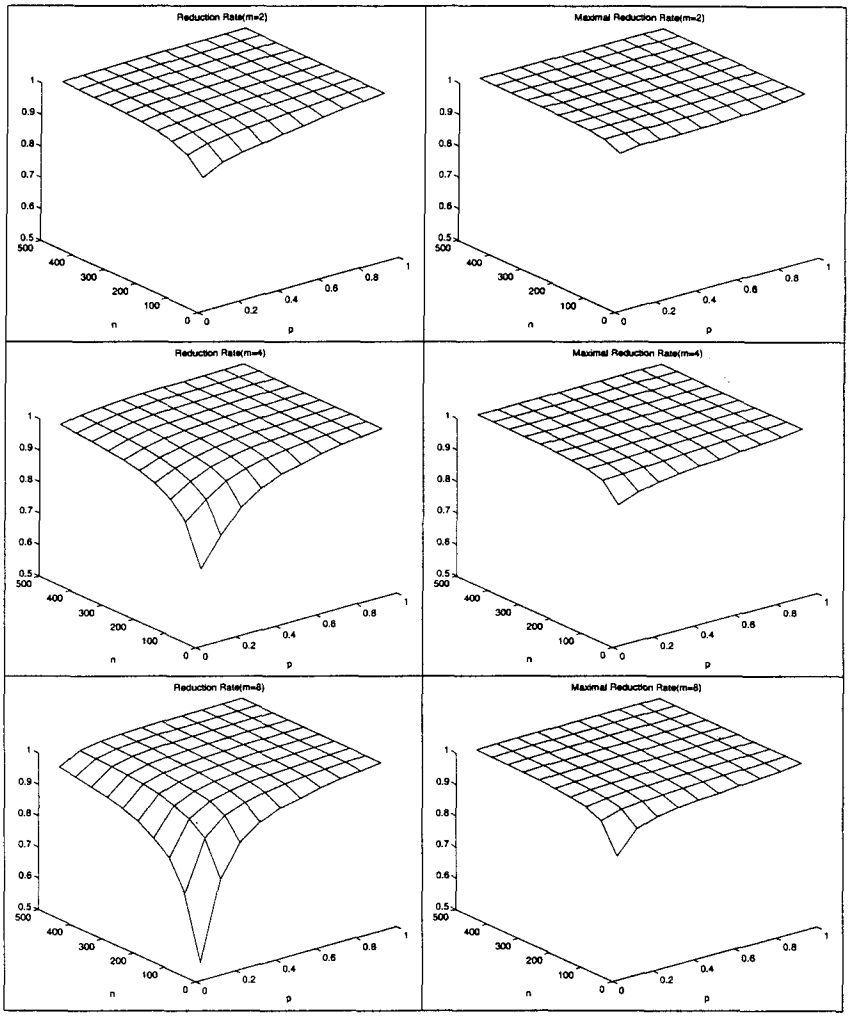

Fig. 10. Average reduction rate and maximal reduction rate for $m = 2$, 4, and 8.

— For small p, the graphs are sparse. When there are fewer arcs in the original graph, it is more likely that these arcs are direct arcs (and must be retained). Hence, most of these are required synchronizations in the final graph.

— For large p, the graphs are dense. When there are more arcs in the original graph, it is more likely that these arcs are transitive arcs (and can be eliminated). Hence, fewer arcs remain in the final graph.

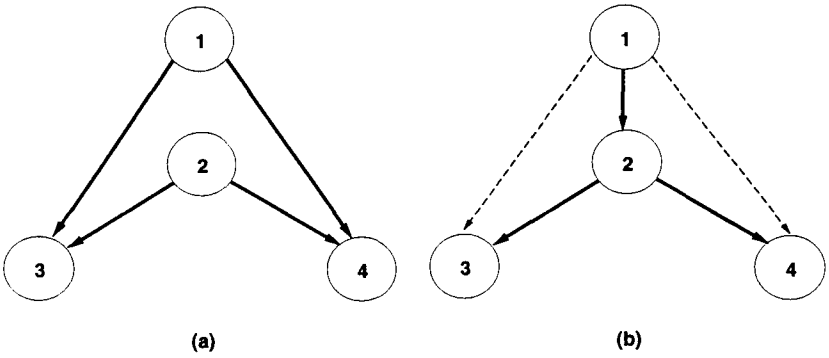

Fig. 11. A simple task graph, where (b) is obtained by adding an arc (1, 2) to (a).

The result may not be intuitive at first glance. Adding more arcs to a DAG does not necessarily increase the number of direct arcs. Consider the task graphs in Fig. 11, where (b) is obtained by adding an arc (1, 2) to (a). Note that there are four direct arcs in (a), but there are only three direct arcs in (b). Adding arc (1, 2) causes arcs (1, 3) and (1, 4), which are direct arcs in (a), to become transitive arcs (they are solid arcs in (a), but dotted arcs in (b)).

- The number of arcs in the final graph increases as m increases. When there are more processors available, a greedy scheduler (e.g., HLF) tends to schedule tasks among processors to minimize the total execution time. Hence, more dependencies are interprocessor dependencies, and are required synchronizations.
- The reduction rates are high (close to 1) except for small n and small p. These graphs are sparse graphs and less reduction can be achieved, especially for large m.

4.2. Simulation on Benchmark Task Graphs

Price and Salama[3] tested the performance of three heuristic scheduling algorithms and the simulated annealing approach on a variety of task graphs representing real problems. Cases 1, 2, 3, and 6 use a task graph in which tasks are fully dependent (i.e., the vertices form a chain); cases 4, 5, 13, and 14 use a task graph in which tasks are fully independent (i.e., no arcs between vertices); case 9 uses a task tree; cases 10 and 11 use a task graph for the parallel Cholesky factorization with 25 tasks; and case 15 is an air defense case study with 20 tasks. We have applied the Min-Sync algorithm to all task graphs except for the extreme cases where the tasks are fully dependent and fully independent. The simulation results

Table II. The Simulation Results of Applying the Min-Sync Algorithm to the Test Cases[a,b]

Case	n	m	S	$\|G\|$	$\|G^+\|$	$\|G^-\|$	$\|G_p\|$	$\|G'\|$	$R(G)$	$R^*(G)$
7, 8	16	2	9	**17**	80	17	12	**7**	0.588	0.913
	16	4	6	**17**	80	17	13	**13**	0.235	0.837
	16	8	6	**17**	80	17	13	**13**	0.235	0.837
9	16	2	10	**15**	49	15	10	**2**	0.867	0.959
	16	4	6	**15**	49	15	11	**10**	0.333	0.796
	16	8	5	**15**	49	15	12	**12**	0.200	0.755
10, 11	25	2	13	**30**	98	30	20	**8**	0.733	0.918
	25	4	9	**30**	98	30	25	**18**	0.400	0.816
	25	8	7	**30**	98	30	25	**24**	0.200	0.755
12	40	2	21	**57**	96	57	52	**8**	0.860	0.917
	40	4	12	**57**	96	57	54	**20**	0.649	0.792
	40	8	7	**57**	96	57	53	**30**	0.474	0.688
15	20	2	12	**47**	133	27	21	**8**	0.830	0.940
	20	4	10	**47**	133	27	19	**13**	0.723	0.902
	20	8	9	**47**	133	27	21	**16**	0.660	0.880

[a] Price and Salama,[3] where column 1 is their test case number.

[b] n, number of tasks; m, number of processors; S, schedule length; $|G|$, number of arcs in the original graph; $|G^+|$, number of arcs in the transitive closure of G; $|G^-|$, number of direct arcs after phase-1 reduction; $|G_p|$, number of direct arcs after adding precedence arcs; $|G'|$, number of direct arcs after phase-2 reduction; $R(G)$, reduction rate; $R^*(G)$, maximum reduction rate.

are shown in Table II. The two columns of special interest (i.e., $|G|$ and $|G'|$) are boldfaced for comparison. The communication costs in Price's paper[3] are computed by using the numbers in column 5 (i.e., $|G|$); however, their communication costs would have been reduced significantly by using the numbers in column 9 ($|G'|$) instead. Note that the schedule length in Table II (i.e., S) equals the optimal length specified in Price's paper.[3] Hence, we have reduced the communication costs without sacrificing the schedule length, and the program is guaranteed to run correctly.

5. CONCLUSIONS

In this paper, we present an efficient algorithm Min-Sync to find a minimal set of interprocessor synchronizations for a task system to be executed on a multiprocessor system. The algorithm uses a new algorithm TC/TR which jointly computes the transitive closure and transitive reduction for a DAG in a single matrix. This matrix allows us to distinguish

direct arcs from transitive arcs and should be useful for a variety of graph-based problems. For any scheduling algorithm, the Min-Sync algorithm can be used to eliminate redundant synchronizations without sacrificing the schedule length. For many optimization problems, the precedence constraints (or dependencies) are represented as DAGs. For example, the data dependency graphs in superscalar pipeline scheduling problems, microcode compaction problems, multiprocessor scheduling problems, and the vertical constraint graphs in channel routing problems. All these problems should also benefit from the TC/TR algorithm.

ACKNOWLEDGMENT

The authors wish to thank Dr. George B. Adams for helpful discussion.

REFERENCES

1. D. Kuck, E. Davidson, D. Lawrie, and A. Sameh, Parallel Supercomputing Today and the Cedar Approach, *Science* **231**:967–974 (1986).
2. A. Gottlieb, R. Grishman, C. P. Kruskal, K. P. McAuliffe, L. Rudolph, and M. Snir, The NYU Ultracomputer—Designing an MIMD Shared-memory and Parallel Machine, *IEEE Trans. on Computers* **C-32**:175–189 (1983).
3. C. C. Price and M. A. Salama, Scheduling of Precedence-Constrained Tasks on Multiprocessors, *The Computer Journal* **33**(3):219–229 (1990).
4. E. Arnould, F. Bitz, E. Cooper, H. T. Kung, R. D. Sansom, and P. Steenkiste, The Design of Nectar: A Network Backplane for Heterogeneous Multicomputers, *Proc. of the Third Int'l Conf. on Architectural Support for Progr. Languages and Oper. Syst.*, pp. 205–216 (1988).
5. W. C. Athas and C. L. Sietz, Multicomputers: Message-passing Concurrent Computers, *IEEE Computer Magazine* **46**:9–24 (1988).
6. F. W. Burton, G. P. McKeown, and V. J. Rayward-Smith, Applications of UET Scheduling Theory to the Implementation of Declarative Languages, *The Computer Journal* **33**(4):330–336 (1990).
7. H. Kasahara and S. Narita, Practical Multiprocessor Scheduling Algorithms for Efficient Parallel Processing, *IEEE Trans. on Computers* **C-33**(11):1023–1029 (1984).
8. V. P. Krothapalli and P. Sadayappan, Removal of Redundant Dependences in DOACROSS Loops with Constant Dependences, *Proc. of the Third SIGPLAN Symp. on Principles and Practice of Parallel Programming* **26**:51–60 (1991).
9. Z. Li and W. Abu-Sufah, On Reducing Data Synchronization in Multiprocessed Loops, *IEEE Trans. on Computers* **C-36**(1):105–109 (1987).
10. J. Blazewicz, M. Drabowski, and J. Weglarz, Scheduling Multiprocessor Tasks to Minimize Schedule Length, *IEEE Trans. on Computers* **C-35**(5):389–393 (1986).
11. E. G. Coffman and R. L. Graham, Optimal Scheduling for Two-Processor Systems, *Acta. Informatica* **1**:200–213 (1972).
12. H. N. Gabow, An Almost-Linear Algorithm for Two-Processor Scheduling, *Journal of the ACM* **29**(3):207–227 (1982).

13. T. C. Hu, Parallel Sequencing and Assembly Line Problems, *Operations Research* **9**:841–848 (1961).
14. H. F. Li, Scheduling Trees in Parallel/Pipelined Processing Environments, *IEEE Trans. on Computers* **C-26**(11):1101–1112 (1977).
15. V. F. Magirou and J. Z. Milis, An Algorithm for the Multiprocessor Assignment Problems, *Oper. Res. Lett.* **8**(6):351–356 (1989).
16. S. P. Midkiff and D. A. Padua, Compiler Algorithms for Synchronizations, *IEEE Trans. on Computers* **C-36**(12):1485–1495 (1987).
17. P. L. Shaffer, Minimization of Interprocessor Synchronization in Multiprocessors with Shared and Private Memory, *Int'l Conf. on Parallel Processing* **3**:138–142 (1989).
18. D. Bernstein, An Improved Approximation Algorithm for Scheduling Pipelined Machines, *Int'l Conf. on Parallel Processing*, pp. 430–433 (1988).
19. T. R. Gross, Code Optimization Techniques for Pipelined Architectures, *COMPCON '83, Spring*, pp. 278–285 (1983).
20. H. Y. Chao and M. P. Harper, Scheduling a Superscalar Pipelined Processor Without Hardware Interlocks, Technical Report TR-EE 94-29, Purdue University (1994).
21. Y. H. Shiau and C. P. Chung, Adoptability and Effectiveness of Microcode Compaction Algorithms in Superscalar Processing, *Parallel Computing* **18**(5):497–510 (1992).
22. T. Yoshimura and E. S. Kuh, Efficient Algorithms for Channel Routing, *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* **CAD-1**:25–35 (1982).
23. A. V. Aho, J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, San Francisco, California (1976).
24. S. Baase, *Computer Algorithms*, Addison-Wesley Publishing Company, San Diego, California (1988).
25. T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*, McGraw-Hill Book Company, New York (1990).
26. M. R. Garey and D. S. Johnson, *Computers and Intractability*, W. H. Freeman and Company, San Francisco, California (1979).
27. D. Gries, A. J. Martin, Jan L. A. van de Snepscheut, and J. T. Udding, An Algorithm for Transitive Reduction of an Acyclic Graph, *Science of Computer Programming* **12**:151–155 (1989).
28. C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall Inc., San Francisco, California (1979).
29. K. K. Lee and H. W. Leong, An Improved Lower Bound for Channel Routing Problems, *IEEE Int'l Symp. on Circuits and Systems* **4**:11–14 (1991).
30. J. S. Wang and R. C. T. Lee, An Efficient Channel Routing Algorithm to Yield an Optimal Solution, *IEEE Trans. on Computers* **39**(7):957–962 (1990).
31. T. L. Adam, K. M. Chandy, and J. R. Dickson, A Comparison of List Schedules for Parallel Processing Systems, *Comm. ACM* **17**(12):685–690 (1974).

Printed in Belgium
Verantwoordelijke uitgever:
Hubert Van Maele
Altenastraat 20 - B-8310 St.-Kruis