# NP-COMPLETENESS OF LINEARLY-CONNECTED MULTIPROCESSOR SCHEDULING

by

Gilbert Verghese
and
Charles R. Dyer

Computer Sciences Technical Report #709
August 1987

# NP-Completeness of Linearly-Connected Multiprocessor Scheduling

Gilbert Verghese
Charles R. Dyer

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

**Abstract**

This paper analyzes the computational complexity of scheduling for linearly-connected multiprocessor architectures. A program to be scheduled is specified by a directed acyclic graph (DAG), where each vertex of the DAG represents a unit time operation to be performed by one of a fixed number of processors, and each arc of the DAG represents a precedence constraint. Each vertex must be assigned a start time and a processor such that all operations are completed and the constraints imposed by the DAG and the architecture are observed. A scheduling algorithm is a decision procedure to determine whether a given number of time units is sufficient for completion of the operations specified by a given DAG. We first show how precedence-constrained scheduling and linearly-connected multiprocessor scheduling can be expressed as subgraph isomorphism problems. We then specify another type of linearly-connected multiprocessor scheduling problem in terms of subgraph isomorphism and prove that it is NP-complete. Finally, we use similar techniques to prove that scheduling for the Aspex Inc. PIPE machine is NP-complete. The NP-completeness results support the development of approximation algorithms and heuristic methods for linearly-connected multiprocessor scheduling.

---

The support of the National Science Foundation under Grant No. DCR-8520870 and the National Aeronautics and Space Administration under Grant No. NAGW-975 is gratefully acknowledged. We also thank Udi Manber and Chuck Stewart for their helpful comments.

## 1. Introduction

Recently a number of multiprocessor architectures have been designed in which the processors are linearly-connected. Parallel processing applications using this kind of restricted-communication architecture have included computer vision [3] and arithmetic expression evaluation [4]. Despite the simplicity and regularity of this organization, however, efficient parallel programming techniques are extremely difficult to determine and analyze. In particular, we have been developing low-level vision algorithms for the Aspex Inc. PIPE machine [6,8].

In this paper we formally analyze the complexity of scheduling operations on linearly-connected multiprocessors such as PIPE. A set of $n$ processors is *linearly-connected* if there is a directed connection from processor $i$ to processor $i$-1 and from processor $j$ to processor $j$+1, for $2 \leq i \leq n$ and $1 \leq j \leq n$-1. Hence there is no global memory, but a processor can directly access data from its own local store or receive data from either of its two nearest neighbors. In the same time unit it may also perform an operation on these data and distribute the result to its two neighbors and its own local store.

The complexity of scheduling operations on a related model of computation is known. That model assumes data can be relayed directly from any processor to any other through the use of shared memory. Consequently, processors act as if they were connected in a complete directed graph. The scheduling of operations using this model of computation is called Precedence-Constrained Scheduling.

The Precedence-Constrained Scheduling problem (PCS) consists of a DAG, G, whose vertices represent unit-time tasks, $n$ processors, and an overall deadline, $d$. An *$n$-processor schedule* for G is a partitioning of all the tasks into a sequence of sets of up to $n$ elements each, such that the precedence constraints imposed by the graph are obeyed. The length of the schedule is the length of this sequence. Given an instance, <G, $n$, $d$>, PCS asks whether there is an $n$-processor schedule for G that meets the deadline $d$. Finding an optimal $n$-processor schedule has

the same time and space complexity as PCS (up to a polynomial). PCS is known to be NP-complete [2,7]. Note that the number of processors, $n$, is a variable. The major scheduling question still open is whether the problem remains NP-complete for a fixed number ($n \geq 3$) of processors [1]. A polynomial algorithm exists for finding a 2-processor schedule for a given DAG. A polynomial algorithm also exists when the precedence graph is a forest and $n$ is arbitrary. In both cases a greedy approach known as *Highest Level First* is used [1].

The remainder of this paper is organized as follows: Section 2 gives a formal definition of the problem of linearly-connected multiprocessor scheduling (LCS). In Section 3 we prove that PCS and LCS can be reformulated in terms of the subgraph isomorphism problem. Section 4 defines another linearly-connected scheduling problem in terms of subgraph isomorphism and proves that it is NP-complete. Finally, in Section 5 we prove that a specific restriction of PIPE scheduling is NP-complete. This establishes the difficulty of the general PIPE scheduling problem.

## 2. Linearly-Connected Multiprocessor Scheduling

An input program is specified by a DAG and is interpreted as a dataflow graph in which each directed arc $(a, b)$ represents data produced by vertex $a$ and received by vertex $b$. Vertices represent tasks to be assigned to processors such that there is a connection (not necessarily direct) between processors assigned to adjacent vertices. In the PCS model of computation all pairs of processors are adjacent. In the LCS model of computation two processors can communicate by relaying data from processor to processor. The data are delayed one time unit for each interim processor since only one datum can be distributed by a processor per time unit. This also implies that a given processor may not perform more than one task per time unit. A processor may store data in local memory and delay the associated task in order to first perform another task or relay

data. The *cost* of the vertex associated with a given task is the time delay from the moment all the inputs of a task are available at a single processor to the moment the task is completed. Hence the cost of a path for a given processor assignment is the total time taken to "traverse" the path.

Let $f$ be a function which assigns to each vertex $a$ a positive cost $f_c(a)$ and a processor number $f_p(a)$. The cost of a path is the sum of the costs of all vertices on the path. The *level* of a vertex $a$ is the cost of the most costly path ending at $a$; $L(a)$ is the set of all vertices at the same level as $a$. The function $f$ is called an *n–processor assignment* for G if (i) $f_p: L(a) \longrightarrow \{1, ..., n\}$ is one-to-one, for each vertex $a$ of G, and (ii) $|f_p(a) - f_p(b)| \leq 1$ for each arc $(a, b)$ of G. That is, (i) no two vertices at the same level have the same processor number, and (ii) arcs of G correspond to direct connections between processors. If all vertex costs are 1, then $f$ is called a *perfect* [4] $n$-processor assignment for G. That is, $f$ is perfect when there is no indirect communication or delay of any value from source to destination.

Processors may be used to relay data from $a$ to $b$; the number of processors required is always less than $n$. However these processors do not correspond to vertices of G. We call H an *r-extension* of G if G is a vertex subset of H and for each arc $(a, b)$ in G there is exactly one path, of length at most $r+1$, from $a$ to $b$ in H. We will also refer to the path from $a$ to $b$ in H as an $r$-extension of the arc $(a, b)$ in G.

**Definition of LCS:** Given an instance, <G, $n$, $d$>, LCS asks whether there is an $n$-processor assignment for an $n$-extension of G under which no path cost exceeds $d$.

Since the processors in the definition of PCS were fully connected, there was no need for an $n$-processor assignment function $f$. This function is required for LCS to satisfy the hardware restriction that only certain connections exist between processors. PCS is concerned therefore with only the second and third conditions that no processor be assigned two tasks at one time and that the deadline be met. The problem of determining the existence of a perfect $n$-processor assignment for G is reported to be NP-complete in [4]. We make use of this property in Section 5.

## 3. PCS and LCS are Equivalent to Restrictions of Subgraph Isomorphism

An instance of the subgraph isomorphism problem [2] consists of two directed graphs, a key graph and a target graph, P. The problem asks whether the target graph contains a subgraph isomorphic to the key graph. The question is asked over a domain of instances containing all pairs of directed graphs. A *restriction* of the original problem results from the same question being asked over a subset of the original problem's domain of instances.

In this section we give a procedure which transforms any instance of PCS to an instance of subgraph isomorphism such that the two problems ask equivalent questions (i.e. have the same answer). From an instance <G, $n$, $d$> of PCS this procedure first constructs the digraph D, whose form is shown in Figure 1 and is defined as

$$D = ( \{1, ..., n\} \times \{1, ..., d\} , \{((x, i) , (y, i+1)) \mid 1 \leq x, y \leq n , \ 1 \leq i \leq d\text{-}1 \} ).$$

Note that only the parameters $n$ and $d$ are used to construct D. Let P be the transitive closure of D. Consider P to be the target graph and G the key graph in a subgraph isomorphism problem. The procedure to construct P from $n$ and $d$ requires time polynomial in these parameters. We now prove that the two problems ask equivalent questions.

**Lemma 3.1:** There is an $n$-processor schedule for G that meets the deadline $d$ if and only if P has a subgraph, H, isomorphic to G.

*Proof*:

If there is such a subgraph H and an isomorphism $\phi$: H —> G, then an $n$-processor schedule for G that meets the deadline $d$ is $( \{\phi((x, i)) \mid (x, i) \in H\} )^{i=1, ..., d}$. This is a partitioning of all the tasks into a sequence of sets of up to $n$ elements each, such that the precedence constraints imposed by the graph G are obeyed.

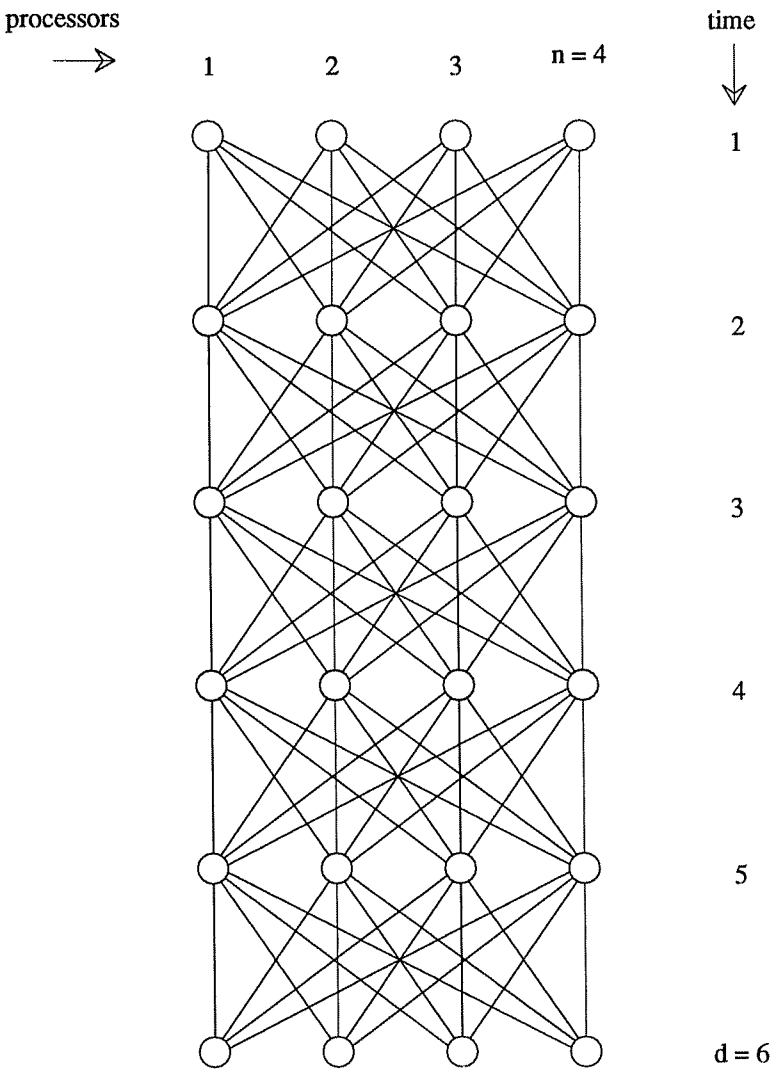



Figure 1. Structure of the digraph D. Arcs are directed forward in time. In this digraph there is an arc from each vertex at one level to each vertex at the next level.

The proof of the converse is also straightforward. If there is an $n$-processor schedule, ($L_1$, $L_2$, ..., $L_r$), for G that meets the deadline $d$ ($r \leq d$), then processors can be assigned to unique vertices in $L_i$ since $|L_i| \leq n$. The mutual disjointness of the domains and ranges of these one-to-one mappings, $m_i : L_i \longrightarrow \{(x,i) \mid 1 \leq x \leq n\}$, $1 \leq i \leq r$, implies the existence of an isomorphism

between G and a subgraph of P. □

The reduction given above proves the following:

**Theorem 3.1:** PCS is equivalent to a restriction of subgraph isomorphism.

We now prove the same results for LCS using a different restriction of subgraph isomorphism. The reduction is more complex, but the proof is similar. Consider the digraph (see Figure 2):

$R = (\ \{1, ..., n\} \times \{1, ..., d\}\ ,\ \{((x, i)\ ,\ (y, i+j)) : 1 \le x, y \le n\ ,\ |x-y| \le 1,\ 1 \le i \le d\text{-}1,\ 1 \le j \le d\text{-}i\}\ ).$

For each vertex, $a$, of R, add a new vertex $a'$ and new arcs $(a, a')$ and $(a', a)$. For each original arc $(a, b)$ of R add a new vertex $v_1$ and *extend* the arc by the path $(a, v_1, b)$. Restore the arc $(a, b)$. Repeat this $n-1$ times with the arcs $(v_1, b)$, then $(v_2, b)$, ..., $(v_{n-1}, b)$ (see Figure 3). A total of $2(n-1)$ new arcs are added, so extending each arc $(a, b)$ of R adds paths of lengths 2, 3, ..., $n+1$, between $a$ and $b$. This new digraph, P, (Figure 3) has

$$2nd + \frac{d(d-1)}{2} \times (3n-2)(d-1)(n-1) \text{ vertices and}$$

$$2nd + \frac{d(d-1)}{2} \times (3n-2)(d-1)(2n-1) \text{ arcs.}$$

That is, it has $O(n^2d^3)$ arcs and vertices. P is now the target graph of the subgraph isomorphism problem.

The key graph, G″, of the subgraph isomorphism problem is obtained from G as follows. Construct G′ from G by extending each arc $(a, b)$ of G by a new path $(a, v_1, v_2, ..., v_{n-1}, b)$. G′ is an $n$-extension of G. Next, for each original vertex $a$ of G, add a new vertex $a'$ not in G′, as well as the new arcs $(a, a')$ and $(a', a)$. The resulting digraph (Figure 4) is G″.

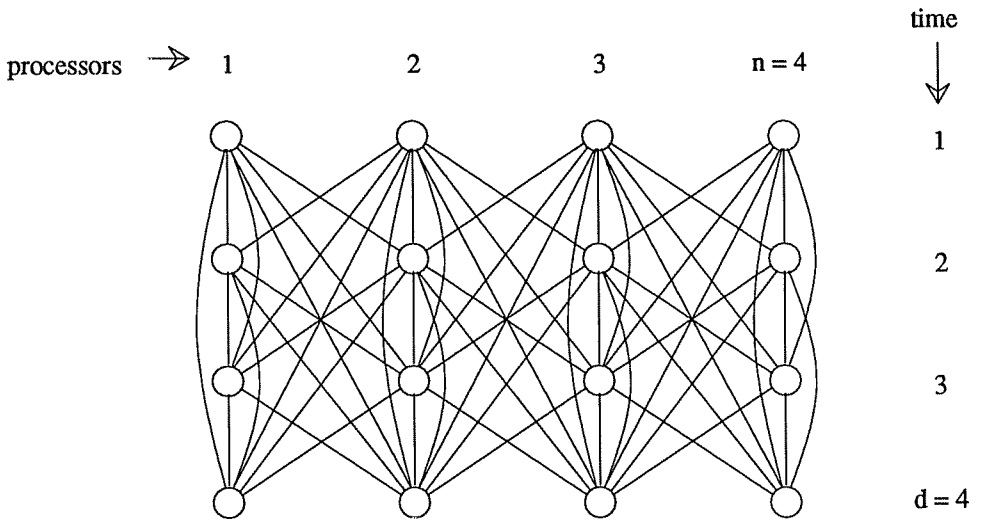



Figure 2. Digraph R used in forming an instance of subgraph isomorphism from an instance <G, 4, 4> of LCS. Arcs are directed forward in time.

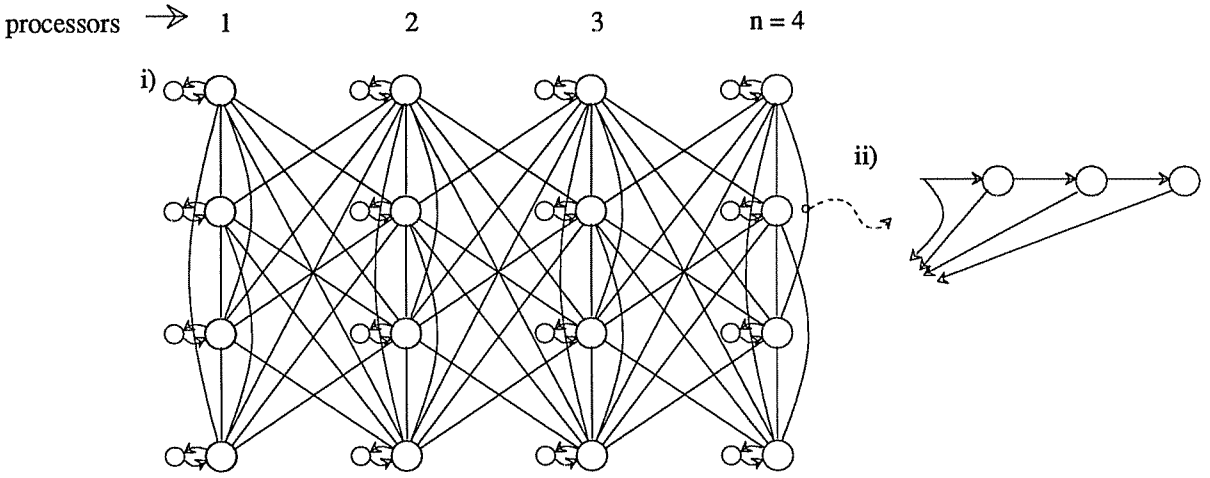



Figure 3. Digraph P formed by extending R in two ways: (i) cycles are introduced, and (ii) all arcs of R are replaced by new arcs and vertices as indicated.

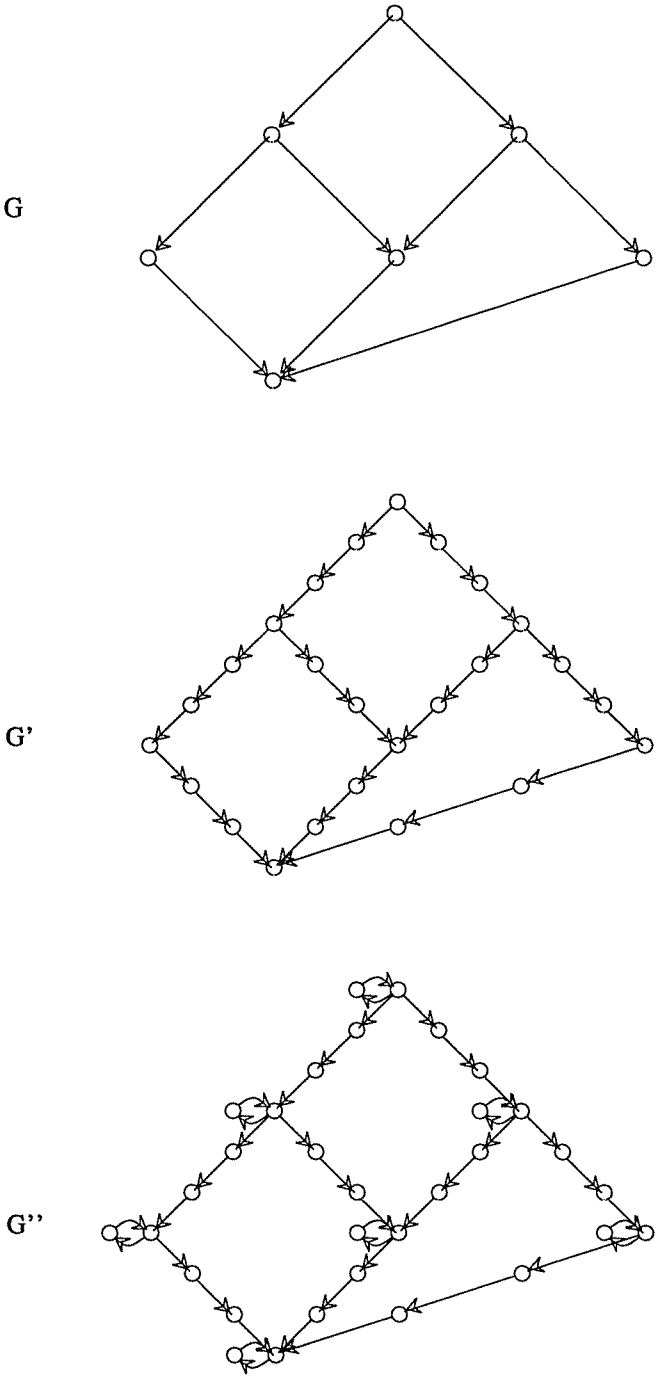



Figure 4. Digraphs G', G'' obtained from the instance <G, 3, d> of LCS.

We now claim that subgraph isomorphism and LCS ask equivalent questions given these instances:

**Lemma 3.2:** P has a subgraph isomorphic to G″ if and only if there is an $n$-processor assignment for an $n$-extension of G under which no path cost exceeds $d$.

*Proof*:

Assume $\phi$ is an isomorphism from G″ to a subgraph of P. For each vertex $a$ in G there is a cycle $(a, a')$ in G″. Each cycle $(a, a')$ in G″ corresponds under $\phi$ to a cycle ((x, i), (x, i)′) in P. That is, $\phi$ maps G into R $\subset$ P. Let H be the subgraph of G′ induced by the inverse image of R under $\phi$. Add edges in H between disconnected vertices which are connected in G′. G′ is an $n$-extension of H. Since G is a vertex subset of H, H is an $n$-extension of G. We will construct an $n$-processor assignment $f$ for H under which no path cost exceeds $d$. For each vertex $b$ in H, let $f_p(b) = y$, where $\phi(b) = (y, j)$, for some j. Let

$$f_c(b) = \min\{\text{ j-i : } (a, b) \text{ is an arc of H, } \phi(a) = (x, i), \text{ and } \phi(b) = (y, j);$$
$$\text{or } b \text{ has no predecessors in H, i = 0, and } \phi(b) = (y, j)\}.$$

This guarantees that if $\phi(a) = (x, i)$ then the level of vertex $a$ under the assignment $f$ is i. Since $\phi$ maps H into R, no two vertices at the same level under this assignment can have the same processor number. $f$ is thus an $n$-processor assignment for H. Since $\phi$ maps H into R and R has critical path length $d$ (i.e. the longest path in R has length $d$), the cost of the most costly path in H under the $n$-processor assignment $f$ cannot exceed $d$.

Now assume that there is an $n$-processor assignment $f$ for an $n$-extension H of G under which no path cost exceeds $d$. We first show that there is an isomorphism from H to a subgraph of R. The isomorphism from G″ to a subgraph of P will then follow. The partition of H given by $\{L(a) \mid a \in H\}$ can be ordered to give $(L_1, L_2, ..., L_r)$, where $r \leq d$ and $L_i$ contains all vertices of H at level $i$ for $1 \leq i \leq r$. Then each vertex $a$ of H can be mapped to the vertex $(f_p(a), i)$ of

R, where $L_i$ contains $a$. Now since R contains all possible arcs $(a, b)$ for which $|f_p(a) - f_p(b)|$ $\leq 1$, H is isomorphic to a subgraph of R. Let $\phi$ be an isomorphism from H to a subgraph of R. G′ is an $n$-extension of H. $\phi$ can be extended by mapping vertices in G′ - H to vertices in P - R. Hence P contains a subgraph isomorphic to G′. Furthermore G is mapped into R under this isomorphism. But for each vertex $a$ of R there is a cycle $(a, a')$, where $a'$ is in P - R. Thus P contains a subgraph isomorphic to G″. □

The reduction given above proves the following:

**Theorem 3.2:** LCS is equivalent to a restriction of subgraph isomorphism.

These reductions are computable in time polynomial in $n$ and $d$. However, using the reductions along with a routine for subgraph isomorphism is likely to result in inefficient algorithms. The instances constructed form a very restricted subset of all instances of subgraph isomorphism. The reductions do, however, give insight into the nature of PCS and LCS.

## 4. L3CS

LCS's model of computation assumes linearly-connected processors. However it restricts each processor to sending a *single* datum to up to three possible destinations: the processor's own local store and the local stores of its two nearest neighbors. Hence when the processor is used to relay data, no other operation can be performed. The model used in this section assumes linearly-connected processing units called stages and allows each stage to send a *different* datum to each of the three destinations. This requires each stage to have, in addition to the processor, a routing network to direct data to specific destinations. In general the routing network may change at each time unit. Thus a processor can send the result of an operation to a given destination while

the stage is also used to relay data to other destinations. L3CS is the problem of scheduling DAGs for this model of computation.

We assume that any datum input to a stage is stored locally. It may subsequently be sent to the processor or to one or both of the neighboring stages. Since a locally-stored value is never sent back to the same stage, up to two destinations are possible for the purpose of relaying data. The "recursive" destination is used only to save the result of an operation performed by the processor. Therefore, a stage must implement the use of data for three purposes: for processing or for communication to two possible destinations. We will define L3CS directly as a restriction of subgraph isomorphism, using as templates the digraphs formed in the previous section for LCS. The templates will be extended to properly model the three possible uses of data by a stage, as well as the communication and delay of data for the scheduling of tasks.

We require two additions to the key and target graphs used in the transformation from LCS to subgraph isomorphism. The first involves replacing processors by stages in the target graph, P, and the second involves differentiating execution (EXEC) vertices from communication (COMM) vertices in the key graph G″. Rather than introducing cycles as shown in Figure 3 for P, three copies of each original vertex of R are combined to form a stage. Hence rather than cycles there are stages with three vertices which have parents and children in common. This reflects the choices in the use of data that we wish to model. Figure 5 illustrates the transformation from R to P in the case of L3CS. Each vertex within a stage of the target graph, P, can be either an EXEC or a COMM vertex. To ensure that at most one of the vertices is an EXEC vertex and at most two are COMM vertices, each vertex within a stage has an arc to a single three-cycle, denoting an EXEC vertex, and to each of two two-cycles, denoting COMM vertices. The key graph will have its vertices connected to either two- or three-cycles.

The definition of the ″ operation for L3CS differs slightly from the definition of ″ for LCS. In Figure 4 cycles are introduced to differentiate between EXEC and COMM vertices. However a

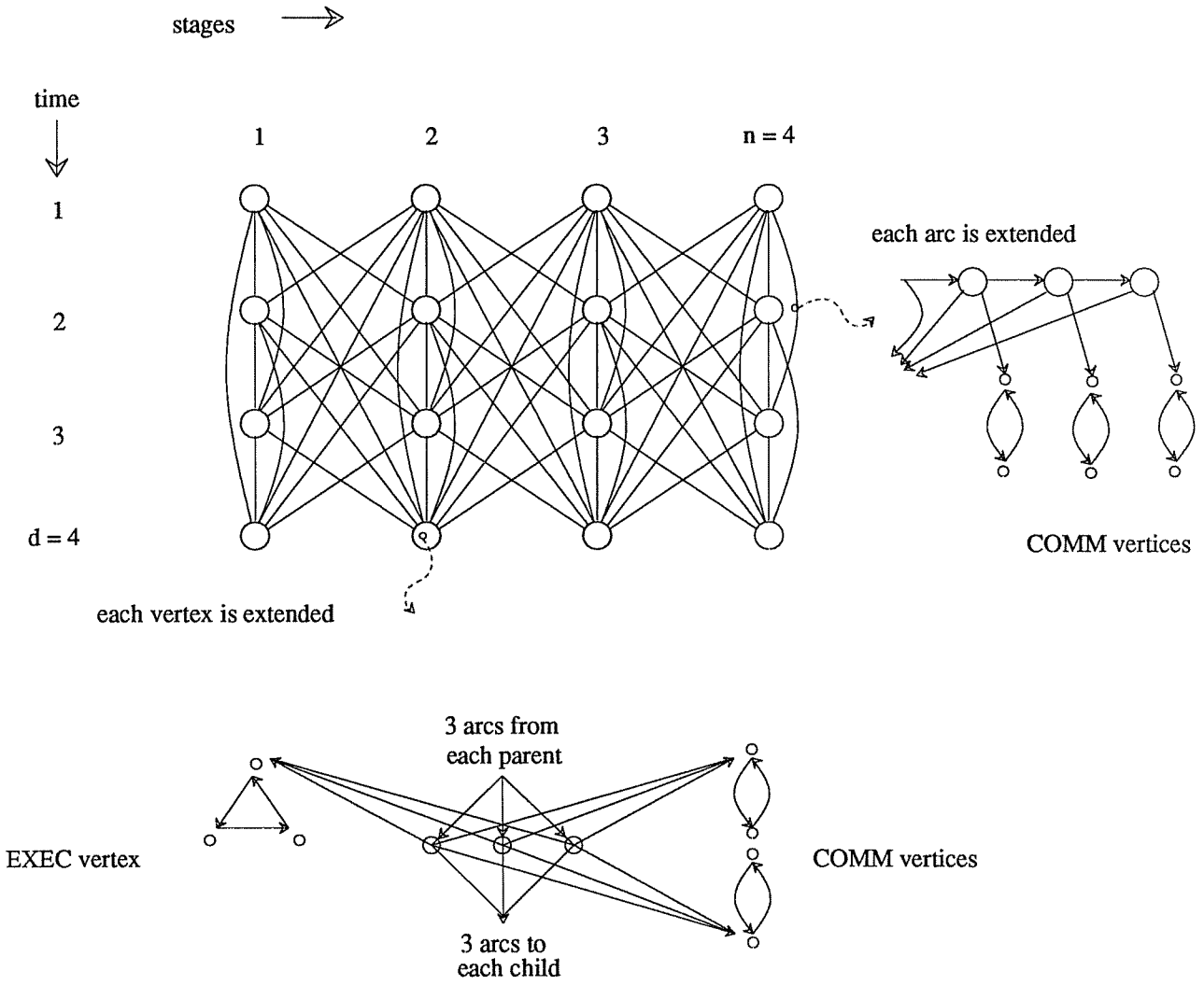



Figure 5. Digraph P, the target graph in an instance of subgraph isomorphism formed from an instance <G, 4, 4> of L3CS.

COMM vertex in the key graph could be mapped to an EXEC vertex in the target graph. This is not necessary for L3CS since there are COMM vertices within each stage of the target graph. For L3CS each EXEC vertex of G″ has an arc to a three-cycle, and each COMM vertex has an arc to a two-cycle. This is shown in Figure 6.

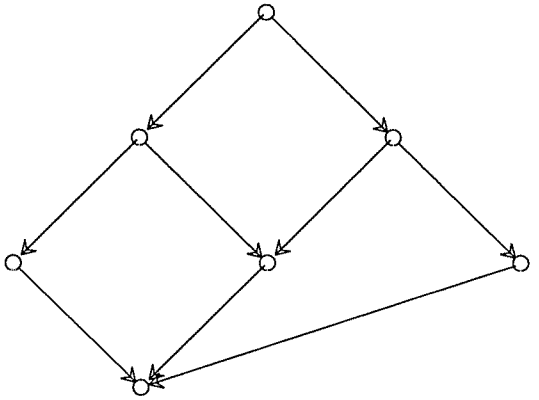
**Definition of L3CS:** Given an instance, <G, $n$, $d$>, L3CS asks whether P has a subgraph isomorphic to G″, where the target graph P and the key graph G″ are defined as above.

### 4.1. L3CS is NP-Complete

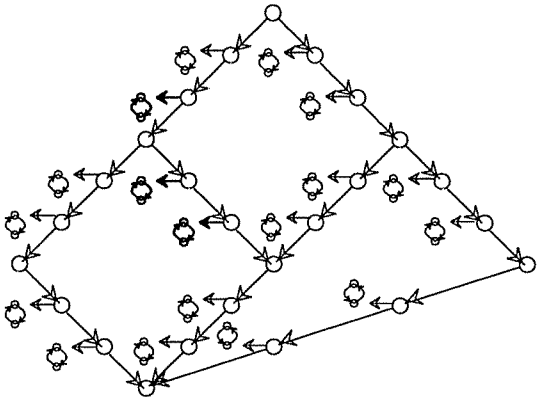
Since L3CS is a restriction of subgraph isomorphism, it is NP-hard. In this section we give a polynomial time many-one reduction to L3CS from PCS, which is known to be NP-complete [2,7]. This reduction establishes the NP-completeness of L3CS.

Recall that given an instance, <G, $n$, $d$>, PCS asks whether there is a partitioning of G's tasks into a sequence of $d$ sets of up to $n$ elements each, such that the precedence constraints imposed by G are obeyed. The partitioning is called an *$n$–processor schedule*. The reduction requires specification of instance and target graphs for L3CS from <G, $n$, $d$>, such that the question asked by PCS is correctly answered by L3CS. The target graph depends only on $n$ and $d$, while the key graph depends on G as well as $n$ and $d$. A target graph, P, similar to that used in the proof of Lemma 3.2, is used here. The number of stages, $n$, and the time limit, $(n+1)d$, determine the target graph, P. The time limit in the proof of Lemma 3.2 was $d$. Here P has $n+1$ times as many arcs and (EXEC) vertices. The proof of Lemma 3.2 used G″ as the key graph. We define the graph I″ in terms of $n$ and $d$. It will have the properties of being (1) isomorphic to exactly two subgraphs of P and (2) dense in P, in the sense of having $nd$ fewer EXEC vertices. In particular $nd$ EXEC vertices of P will not be members of the image of I″ under the isomorphism. The key graph will consist of G″ $\cup$ I″. Hence only $nd$ vertices can be used for scheduling G, as

G



G'



G''

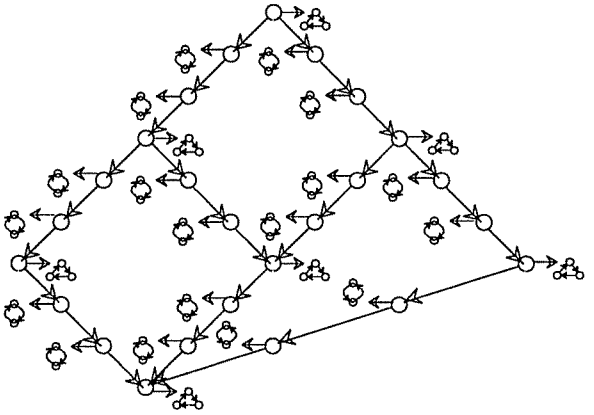


Figure 6. Digraphs G', G'' obtained from the instance <G,3,d> of L3CS.

in PCS. I″ is the result of performing the ″ operation on the DAG I($n$, $d$).

**Definition of I($n$, $d$):**

$$V_k = \{1, ..., n\} \times \{kn+1, ..., n+kn+1\}$$

$$\begin{aligned} E_k = &\{ ((x, i), (x, i+1)) \mid 1 \le x \le n, kn+1 \le i \le n+kn \} \cup \\ &\{ ((x, i), (x+1, i+1)) \mid 1 \le x < i\text{-}kn, kn+1 \le i \le n+kn \} \cup \\ &\{ ((n+1\text{-}x, i), (n\text{-}x, i+1)) \mid 1 \le x < i\text{-}kn, kn+1 \le i \le n+kn \} \cup \\ &\{ ((x, n+kn), (x+2, n+kn+1)) \mid 1 \le x \le n-2 \} \cup \\ &\{ ((n+1\text{-}x, n+kn), (n\text{-}x\text{-}1, n+kn+1)) \mid 1 \le x \le n-2 \} \end{aligned}$$

$$I(n, d) = (\cup \{V_k \mid 0 \le k \le d-1\}, \cup \{E_k \mid 0 \le k \le d-1\}).$$

Hence I($n$, $d$) is formed by connecting $d$ copies of I($n$, 1) end to end as shown in Figure 7. For $k$ = 1, ..., $d$, the levels $kn$ and $kn$+1 of the DAG I($n$, $d$) are connected by arcs from vertices to all neighbors up to distance 2 away. Three examples of I($n$, $d$) are given in Figure 7.

**Lemma 4.1:** There is an $n$-processor schedule for G that meets the deadline $d$ if and only if there is an isomorphism from G″ ∪ I″ to a subgraph of P.

*Proof*:

I″ is *dense* in P in the sense that there are $n^2d$ EXEC vertices in I″ and $n(n+1)d$ EXEC vertices in P. However the EXEC vertices of P at levels $k(n+1)$, for $k$ = 1, ..., $d$, cannot be isomorphically mapped to vertices of I″ due to communication delays. Hence the critical path length of any subgraph of P isomorphic to I″($n$, $d$) is the same as the critical path length of P. There are only two isomorphisms from I″ to a subgraph of P. They are essentially the same by symmetry. The allocation of P's EXEC vertices and arcs to the isomorphic image of I″ is shown in Figure 8. Only the remaining EXEC vertices and paths in P can be used for the image of G″. Since there are sufficiently many arcs remaining in P to represent communication from all stages to all other stages, the existence of an $n$-processor schedule for G meeting the deadline $d$ implies the existence of an isomorphism from G″ ∪ I″ to a subgraph of P.

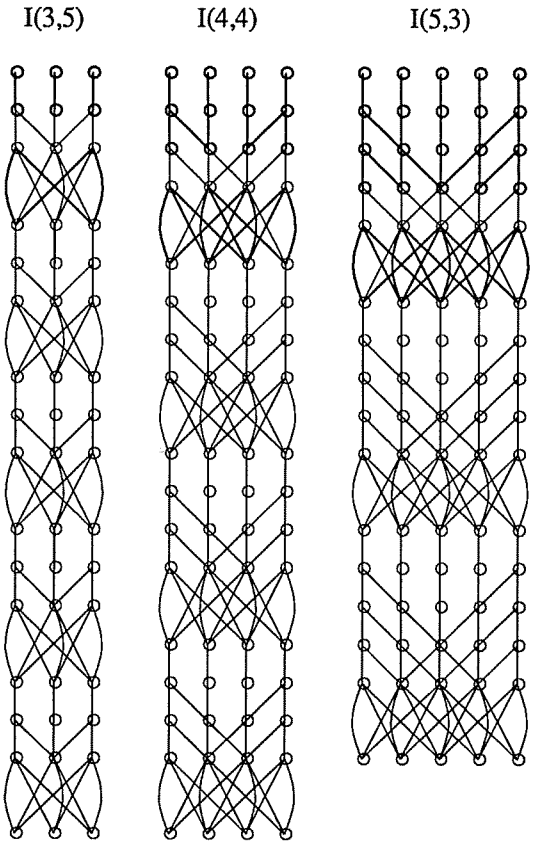



Figure 7. Digraphs I(3,5), I(4,4), and I(5,3). Arcs are directed downward.

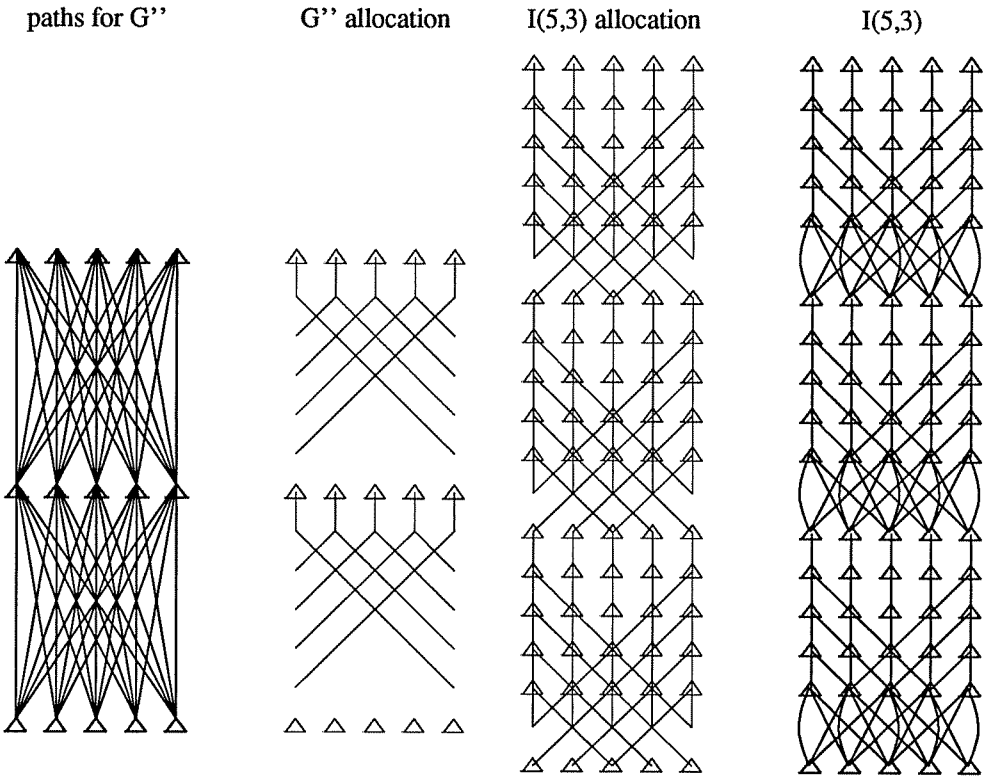



Figure 8. EXEC vertex and arc allocation for a subgraph of P isomorphic to G'' ∪ I(5,3)

Since there is a unique isomorphism from I″ to a subgraph of P (up to symmetry), the existence of an isomorphism from G″ ∪ I″ to a subgraph of P implies that all EXEC vertices of G″ are mapped to at most $d$ distinct levels of P. These levels form the partitioning of G required in an $n$-processor schedule for G. Thus there is an $n$-processor schedule for G that meets the deadline $d$ if and only if there is an isomorphism from G″ ∪ I″ to a subgraph of P. □

Finally, the many-one reduction given in the proof of Lemma 4.1 is computable in time polynomial in $nd$. Hence L3CS is not shown here to be NP-complete in the *strong* sense [2]. However since $n$, the number of processors, is bounded from above in practice, $d$ is the dominating factor. No schedule with deadline $d$ exists for a DAG having critical path length

greater than $d$ or more than $nd$ vertices. Preprocessing can be added to the reduction to detect these cases. Hence in practice the reduction is computable in time polynomial in the number of vertices in the DAG. This supports the development of heuristic methods and approximation algorithms for scheduling DAGs on architectures similar to L3CS [5].

## 5. PIPE Scheduling

LCS's model of computation allows three different inputs to be saved in the local store of a processor at each time unit. In this section we consider a related model of computation which corresponds to a physically existing hardware architecture, Aspex Inc.'s PIPE [3]. It possesses the same interconnection scheme as LCS and L3CS (with several additional buses, which we ignore here). However, a PIPE stage allows only one value to be saved in the local store (image buffers) at each time unit. That is, two binary operations are used to initially combine the three input values into one; this value is then stored in a buffer and can be used subsequently for performing various operations in the stage. Consequently, unlike LCS and L3CS only one of the three input values to a stage can be stored at any time unit. Hence storing the result of a stage's operation requires that the destination stage have no other inputs in the time unit following the operation.

Program DAGs consist of vertices labeled with basic lookup table, arithmetic, and Boolean operations that can be executed by various functional units within a stage of PIPE. Because of the hardware available in each stage, in general several vertices may be assigned to the same stage at one time. This will not complicate the proofs, however. Other details of a PIPE stage are discussed in [3]. The only additional details needed here are that there is one two-variable lookup table operation (TVF-LUT) on each stage and there are two binary arithmetic (ALU) operations which combine the stage's three inputs into one. Given a labeled DAG G, $n$ PIPE stages, and an overall deadline, $d$, PIPE scheduling asks whether there is an assignment function mapping each

labeled vertex of G to a PIPE operation, stage number, and time unit, such that the label corresponds to the operation, every arc in G corresponds to a direct or indirect connection in PIPE, and the overall deadline $d$ is met.

In the next section we prove that a restriction of PIPE scheduling is NP-complete (a sufficient condition for general PIPE scheduling to be NP-complete). This restriction simplifies the presentation because it does not require the detailed specification of a stage. We will treat stages like single processors by defining DAG *components* which map onto single stages. For the purposes of scheduling, components will necessarily map to entire stages. Components can then be treated as single vertices. A component consists of a DAG isomorphic to the DAG $(\{a, b\}, \{(a, b), (a, b)\})$, where $a$ is labeled ALU, $b$ is labeled TVF-LUT, $a$ has two incoming arcs and two outgoing arcs, and $b$ has two incoming arcs and one outgoing arc. Every component then has two incoming arcs and one outgoing arc, as does every vertex in a *reverse* binary tree (precise definition below).

### 5.1. PIPE Scheduling is NP-Complete

We stated in Section 2 that in the LCS model of computation the problem of determining the existence of a perfect $n$-processor assignment for a given DAG G and a given number of processors $n$, is NP-complete. The restriction of this problem to the domain R = {(V, E) | (V, $\{(a, b) \mid (b, a) \in E\}$) is a directed binary tree}, is also known to be NP-complete [4]. The elements of R are called *reverse* binary trees.

In this section we prove the same result for the problem of determining the existence of a perfect $n$-processor assignment for a given DAG on PIPE. To do so we will restrict this problem to the domain of DAGs R′ = { ($\{C_x \mid x \in V$ and $C_x$ is a component$\}$ , $\{(C_a, C_b) \mid (a, b) \in E\}$ ) | (V, E) ∈ R }. Informally, the components form reverse binary trees. For each reverse binary tree

in R there is a "reverse binary tree of components" in R′. We have treated components as vertices for simplicity of specifying connections among components. This also suggests a natural bijection between R and R′. We now make the following claim:

**Lemma 5.1:** In the PIPE model of computation the problem of determining the existence of a perfect $n$-processor assignment for a given DAG $T' \in R'$ and a given number of processors $n$, is NP-complete.

*Proof*:

This problem is equivalent to the NP-complete problem of determining the existence of a perfect $n$-processor assignment for a given DAG $T \in R$ and a given number of processors $n$ in the LCS model of computation. The equivalence of the two problems is given by the bijection between their domains, R and R′. □

To prove that PIPE scheduling is NP-complete we first note that it is NP-hard since we can verify in linear time whether a given assignment function meets a deadline $d$ and is an $n$-processor assignment function for a given DAG G and a given number of processors $n$. We now give a polynomial time many-one reduction from the NP-complete problem in Lemma 5.1 to PIPE scheduling.

Let $<T', n>$ be an instance of the problem in Lemma 5.1. We will construct an instance $<G, n, d>$ of PIPE scheduling. Let $d$ be the critical path length of the DAG T in R which corresponds to T′ in R′. Let $I = \cup \{ C_i \mid i \leq nd - |T| \text{ and } C_i \text{ is a component} \}$. Hence I is a DAG containing $nd - |T|$ components. The ALU operations in these disconnected components produce constant results and require no input. Finally, let $G = I \cup T'$.

**Lemma 5.2:** There is a perfect $n$-processor assignment for T′ in the PIPE model of computation if and only if G can be scheduled on PIPE with $n$ stages and deadline $d$.

*Proof*:

If there is a perfect $n$-processor assignment for T′ on PIPE, then G can be scheduled on PIPE with $n$ stages and deadline $d$ since vertices of $T' \subset G$ can be assigned to stages according to the perfect $n$-processor assignment. All the vertices of $I \subset G$ can be assigned to the remaining stages at any of the $d$ time units since the components of I are all independent.

If G can be scheduled on PIPE with $n$ stages and deadline $d$, then exactly $|T|$ stages are assigned to components of T′ since all other stages are assigned to components of I. Furthermore, no stages can be used for relaying or delaying of values since only one value can be stored in a given stage's buffers per time unit. This value must be the output of the ALU operation corresponding to the ALU vertex of the component which was assigned to the given stage. Since there is no relaying or delaying of values, the assignment of PIPE stages to T′ given by the schedule of G is a perfect $n$-processor assignment for T′. □

Finally, the many-one reduction is computable in time polynomial in $nd$. Thus the comments made in Section 4 concerning L3CS also apply here.

## 6. Concluding Remarks

The NP-completeness proof given for PIPE scheduling is simpler than that given for the NP-completeness of L3CS. The former does not apply to the latter since the former relied on the restricted capabilities of PIPE with respect to its inability to store data for both communication and delay, thus forcing a perfect assignment.

The results of this paper make precise the difficulty of scheduling programs for linearly-connected multiprocessors such as PIPE. Because of this inherent complexity, we have simultaneously been developing heuristic methods for scheduling which, in practice, produce nearly optimal schedules [5].

## References


1. D. Dolev and M. Warmuth, Scheduling flat graphs, *SIAM J. Comput.* **14**, 1985, 638-657.

2. M. R. Garey and D. S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, Freeman, New York, 1979.

3. E. W. Kent, M. O. Shneier, and R. Lumia, PIPE — Pipelined image-processing engine, *J. Parallel and Distributed Computing* **2**, 1985, 50-78.

4. C. E. McDowell and W. F. Appelbe, Processor scheduling for linearly connected parallel processors, *IEEE Trans. Computers* **35**, 1986, 632-638.

5. C. V. Stewart and C. R. Dyer, A scheduling algorithm for the Pipelined Image-Processing Engine, to appear in *J. Parallel and Distributed Computing*.

6. C. V. Stewart and C. R. Dyer, Convolution Algorithms on the Pipelined Image-Processing Engine, Technical Report 643, Computer Science Department, University of Wisconsin — Madison, May 1986.

7. J. D. Ullman, NP-complete scheduling problems, *J. Comput. Systems Sci.* **10**, 1975, 384-393.

8. G. Verghese, S. Mehta, and C. R. Dyer, Image Processing Algorithms on the Pipelined Image-Processing Engine, Technical Report 668, Computer Science Department, University of Wisconsin — Madison, September 1986.