Journal of Universal Computer Science, vol. 5, no. 10 (2000), 994-1014
submitted: 1/4/00, accepted: 17/7/00, appeared: 28/10/00 © Springer Pub. Co.

On The Thread Scheduling Problem

Wing-Ning Li
Department of Computer Science and Computer Engineering
University of Arkansas
email:wingning@uafsysb.uark.edu

Jing-Fu Jenq
Department of Computer Science
Montclair State University
jenq@pegasus.montclair.edu

Abstract: This paper considers the thread scheduling problem. The thread scheduling problem abstracts the problem of minimizing memory latency, using a directed data dependency graph generated form a compiler, to improve run time efficiency. Two thread scheduling problems are formulated and shown to be strongly NP-complete. New methods and algorithms for analyzing a data dependency graph in order to compute the theoretical best runtime (lower bound of the finishing time) and to estimate the required minimum number of PEs needed to achieve certain finishing time are presented. The new methods and algorithms improve upon some of the existing analysis and transformation techniques.

Key Words: Scheduling, memory latency, multi-threaded architecture, complexity.

1 Introduction

The thread scheduling problem deals with the static or compile time schedule of a data dependency graph on a multi-threaded architecture. The data dependency graph is a directed acyclic graph (dag), where the vertices represent instruction threads and edges correspond to interdependency among threads. A multi-threaded architecture is a multiprocessor system which consists of many processing elements (PEs). Each PE contains a local data cache called L1 cache, which is connected across a single interconnection network. Global memory is used to share data between PEs and is connected to the interconnection network. Due to the transfer of data among the threads (vertices in the dag) memory latencies are introduced. Thus, in order to schedule threads on the available set of PEs in such a way that the maximum finishing time (makespan) is minimized, we must consider both the processing time of the threads as well as memory latency delays.

This paper formally formulates the thread scheduling problem, analyzes the computational complexity of the problem, and proposes techniques and algorithms to address certain aspects of the problem.

The thread scheduling problem abstracts the problem of minimizing memory latency using a directed data dependency graph generated from a compiler. Such a problem was first considered by Thornton and Andrews [17]. In addition to a formal formulation of the problem, NP-completeness results, and new algorithms, this paper also improves upon some of the analysis and transformation techniques introduced in [17].

These results are practical to implement in the optimization stage of the compiler/loader portion of the system software. These results are directly applicable to the development of programs for the optimal generation of instruction threads in reconfigurable, multi-threaded architecture [17]. The approaches of using a directed data dependency graph produced from a compiler for optimization have been used by other researchers for related applications [2, 7, 14, 15].

The remainder of the paper is organized as follows. The next section will introduce terminology used and formulate the thread scheduling problem. Next, the computational complexity results are proved. Analysis and transformation techniques are developed in Section 4. Concluding remarks are given in the last section.

2 Formulation and terminology

This section is divided into the following subsections for a more logical presentation.

2.1 Problem representation

A dag or data dependency graph can be used to represent the dependencies among elementary computational steps (such as threads) in a given program or algorithm. Some compilers automatically generate these graphs in intermediate optimization stages. The reader is referred to [3, 13] for detailed discussions and examples on how programs are represented as graphs and to [1] for examples on how compilers generate these graphs. In what follows, we shall describe the dag representation as an input instance to be scheduled.

Definition 1 *A program thread graph is a dag $G=(V,E,d,cm)$, where*

a) V is the vertex set of G. Each vertex represents a thread;

b) $E \subset V \times V$ is the edge set of G. The edge set forms a partial ordering of V, representing data and control dependencies between threads in V;

c) d is a thread weighting function with domain V and range Z^+. $d(v) = w$ where w is the (estimated) running time for v;

d) cm is a edge weighting function with domain E and range Z^+. $cm(<u,v>) = w$ *where w is the (estimated) communication cost of edge* $<u,v>$ *when threads u and v are processed by different processors.*

For the sake of generality, we state the edge weighting function cm in terms of arbitrary functions. For the architecture introduced in the next subsection, cm is a constant function denoting memory latency for a cache miss.

2.2 Architecture definition

The multi-threaded architecture introduced in [17] is used here. The block diagram of a simplified tightly coupled architecture obtained from [17] is shown in Figure 1. As can be seen from the figure, associated with each processing element, PE, is a local data cache (L1 cache) which is connected across a single interconnection network. Data memory is a shared memory storing global data. Accessing data from a L1 cache is assumed to introduce no communication overhead. Getting data from L2 cache incurs an overhead of a cache miss from L1 cache, and any delays due to the contention across the interconnection network. Accesses of data in the data memory incur a miss from both L1 and L2 cache, as well as any delay due to network contention.

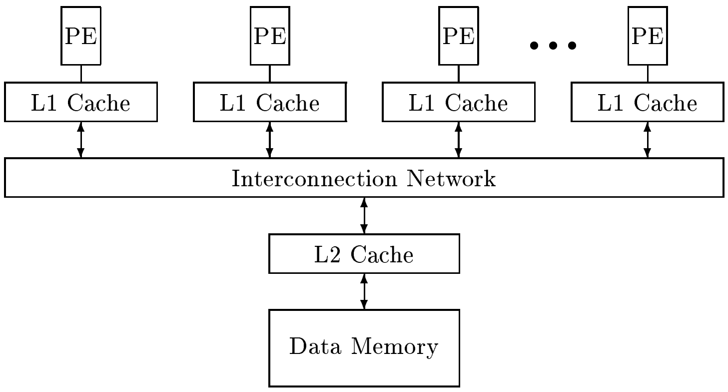


Figure 1: Architecture block diagram.

2.3 A simple example

Before formally state the thread scheduling problem—statically scheduling a program thread graph on a multi-threaded architecture with n PEs, we shall consider a simple example, which is used to illustrate the rationale behind our formulation.

Let us consider the program thread graph shown in Figure 2(a). In the example, we have three threads a, b, and c. The running time of these threads are 6, 5, and 4 respectively. The running time of the thread indicates how long it takes a PE to process it and includes the possible memory latency for executing it.

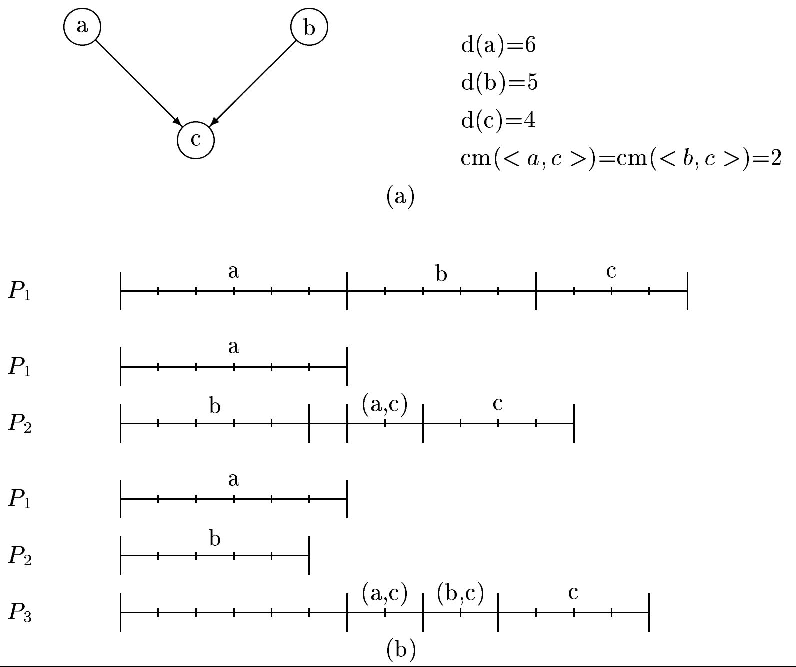


Figure 2: An example program thread graph and its schedule.

The edges from a to c and from b to c indicate that c must wait for the completion of a and b before it can be processed by any PE, due to data and control dependencies. They also represent the need for communication between a,c and between b,c.

If a and c are processed by the same PE, the data generated by a and requested by c are in the local L1 cache of the PE. Hence, no communication overhead is introduced. On the other hand, if a and c are processed by two different PEs, then the only way to communicate the data is through the global data memory. Hence, as described in the previous subsection, an overhead of cache miss as well as any delay due to network contention is introduced. In this example, the PE that process c must wait for another two time units, after the completion of a, so that it has the data to begin the execution of c, since $cm(< a, c >) = 2$. Here we assume once the initial transfer has completed, no more overhead will be introduced, since the possible additional transfer of data from data memory is accounted for in the running time of the thread. To end the subsection, we depict three schedules of the program thread graph of Figure 2(a) using Gantt Chart like diagrams in Figure 2(b).

2.4 Thread scheduling problem

Definition 2 *Let PRED(v) be the set of immediate predecessors of v. PRED(v) = $\{u \mid < u, v > \in E\}$. Let SUCC(u) be the set of immediate successors of u. SUCC(u) = $\{v \mid < u, v > \in E\}$.*

In our notation, when the output of a function $\sigma(x)$ is an ordered pair such as $< y_1, y_2 >$, then $\sigma_1(x)$ denotes the first element and $\sigma_2(x)$ the second element respectively, i.e., $\sigma_1(x) = y_1$ and $\sigma_2(x) = y_2$.

Definition 3 *Given dag $G = (V, E, d, cm)$ as a program thread graph and a multi-threaded architecture with m PEs, a valid schedule is a total function $\sigma : V \longrightarrow N \times \{1, 2, \ldots, m\}$ which satisfies the following two conditions:*

1. *For all $t \in N$, $\mid \{v_i \mid v_i \in V$ and $\sigma_1(v_i) \leq t < \sigma_1(v_i) + ex(v_i) + d(v_i)\} \mid \leq m$;*
2. $\sigma_1(v_i) \geq \max_{u \in PRED(v_i)}\{\sigma_1(u) + ex(u) + d(u)\}$

where $ex(v)$ denotes the extra processing time due to communication overhead and $ex(v) = \sum_{u \in PRED(v), \sigma_2(u) \neq \sigma_2(v)} cm(u, v)$

In plain language, for each thread v_i in V, the function σ assigns to it a starting time $(\sigma_1(v_i))$ for execution and a PE $(\sigma_2(v_i))$. Condition (1) states that no more than m threads are ever executed simultaneously, condition (2) ensures that the precedence constraints as well as communication overheads are respected. In condition(2), $\max_{u \in PRED(v_i)}\{\sigma_1(u) + d(u)\}$ alone ensures the precedence constraints are respected and adding $ex(u)$ to $\sigma_1(u) + d(u)$, both the precedence constraints and the communication overheads are respected. The finishing time ω for a valid schedule σ is given by $\omega = max\{\sigma_1(v) + ex(v) + d(v) \mid v \in V\}$.

Now, the thread scheduling problem can be defined precisely as follows.

Thread Scheduling Problem

Input: A dag $G = (V, E, d, cm)$ and a positive integer m, where G represents a program thread graph and m represents the number of PEs in the multi-threaded architecture.

Output: A valid schedule σ of the input such that the finishing time of σ is minimum among all possible valid schedules of the input.

Basic Thread Scheduling Problem

Input: Same as the thread scheduling problem except that cm is restricted to a constant function.

Output: Same as the thread scheduling problem.

We define both problems as optimization problems. For the decision versions of the problems, we are given an additional integer K and asked whether there exists a valid schedule σ whose finishing time is no more that K.

Definition 4 *Let* $\underline{PRED_\sigma(v)} = \{u \mid u \in PRED(v) \text{ and } \sigma_2(u) \neq \sigma_2(v)\}$, *where* σ *is a valid schedule.*

For the basic thread scheduling problem, $ex(v)$ in definition 3 can be simplified to $cm \times \mid PRED_\sigma(v) \mid$. The examples given in section 6 of [17] can be treated as instances of the basic thread scheduling problem.

2.5 Task execution model comparison

At this time it is appropriate to make a few observations about the similarity and difference between the proposed model and another well-known task execution model in the literature, known as the compile time macro-dataflow model, that has been considered by Wu and Gajski [19], Sarkar [13], Yang and Gerasoulis [20], and others.

The formulation of the input graph in both models is basically identical. The execution models, however, are rather different. Let us analyze some of the crucial differences resulted from different architecture assumptions.

The problem formulation given in the previous section is based on a shared memory multi-threaded architecture. Under such a architecture, a task must obtain data from its predecessor tasks and the communication cost of getting the data from a task's predecessors is accumulative. Note that the extra processing time due to communication overhead in Definition 3, $ex(v)$, of a given schedule σ is given by the following summation:

$$ex(v) = \sum_{u \in PRED(v), \sigma_2(u) \neq \sigma_2(v)} cm(u, v)$$

This accumulative property is illustrated in the last schedule of Figure 2(b).

The compile time macro-dataflow model is based on a message-passing architecture. In this model, a task receives all input from its predecessor tasks before starting execution, executes to completion without preemption, and immediately *sends* the output to all successor tasks in parallel. Under the message-passing architecture, a task instead of fetching data from its predecessors, sends the data needed by its successors in parallel. In this case, the communication cost is dominant in the sense that, assuming all predecessor tasks finish at the same time, the largest communication cost of a task's predecessor running on a different PE determines the wait time of the task. More precisely, to model this dominant property, the two conditions of Definition 3 should be changed to:

1. For all $t \in N$, $\mid \{v_i \mid v_i \in V \textit{ and } \sigma_1(v_i) \leq t < \sigma_1(v_i) + d(v_i)\} \mid \leq m$;
2. $\sigma_1(v_i) \geq \max_{u \in PRED(v_i)}\{\sigma_1(u) + d(u) + cm(u, v_i)\}$

Using the compile time macro-dataflow model, the last schedule of Figure 2 (b) can actually be shortened by two time units. The reader is referred to [19, 13, 16, 12, 20] for results related to the compile time macro-dataflow model.

3 Complexity results

In this section, we shall first establish the basic thread scheduling problem is NP-complete in the strong sense. We shall use the following NP-hard problem [4] to prove that.

Partition into Triangles

Input: A graph $G = (V, E)$, with $\mid V \mid = 3q$ for a positive integer q.

Output: 'yes' iff there is a partition of V into q disjoint sets V_1, V_2, $\ldots$, V_q of three vertices each such that, for each $V_i = \{V_{i[1]}, V_{i[2]}, V_{i[3]}\}$, the three edges $(V_{i[1]}, V_{i[2]})$, $(V_{i[1]}, V_{i[3]})$, and $(V_{i[2]}, V_{i[3]})$ all belong to E.

We may assume that $\mid E \mid \geq 3q$, otherwise the answer to the partition into triangles problem is always negative. Given an instance of the partition into triangle problem, we use the following construction to build a corresponding instance of the decision version of the basic thread scheduling problem, in polynomial time, such that the instance of the partition into triangles problem has a 'yes' answer iff the corresponding instance of the basic thread scheduling problem also has a 'yes' answer.

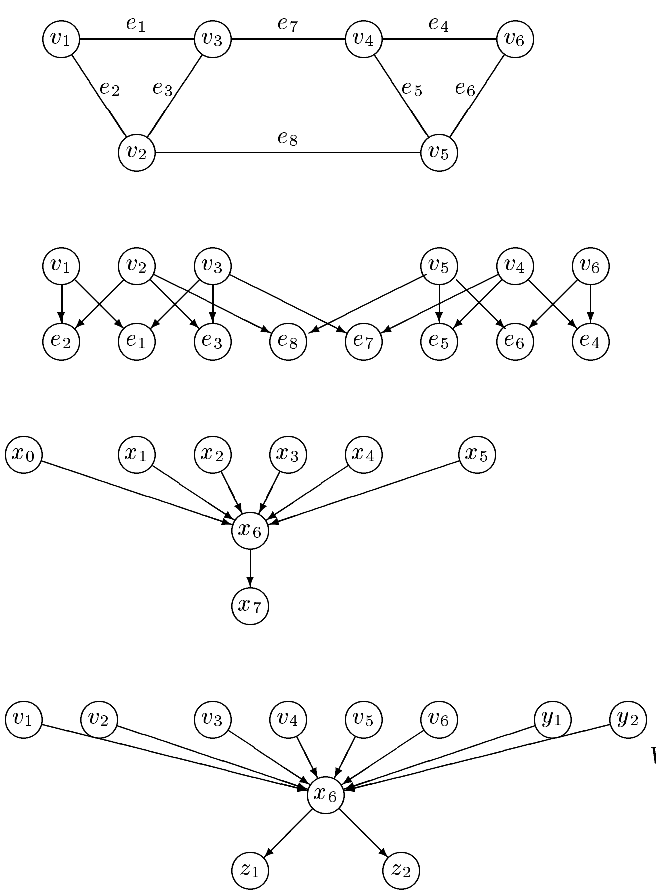
We shall describe the construction in stages so the arguments used in the proof can be more easily followed.

<u>Construction 1:</u>

Let $G = (V, E)$, with $| V |= 3q$ for a positive integer q, be an instance of the partition into triangles problem. The corresponding instance of the basic thread scheduling problem has $m =| E | -2q + 1$ many PEs, $K = 6 + 3 | E |$ as the desired finishing time for the valid schedule, and a dag $G' = (V', E', d, cm)$. The details of G' are as follows.

1. Threads in each group.
   (a) vertex-group (v-threads) V_V. For each vertex $v_i \in V$ there is a thread $v_i \in V_V$, where $d(v_i) = 2$.
   (b) edge-group (e-threads) V_E. For each edge $e_i \in E$, there is a thread $e_i \in V_E$, where $d(e_i) =| E |$.
   (c) x-group (x-threads) V_X. There are eight threads in V_X. They are $x_i, 0 \leq x_i \leq 7$, where $d(x_i) = 1$, $x_i, 0 \leq x_i \leq 5$, $d(x_6) = 2$, and $d(x_7) = 2 | E | -2$.
   (d) y-group (y-threads) V_Y. There are $| E | -3q$ threads y_i, $1 \leq y_i \leq| E | -3q$, in V_Y, where $d(y_i) = 6$.
   (e) z-group (z-threads) V_Z. There are $| E | -3q$ threads z_i, $1 \leq z_i \leq| E | -3q$, in V_Z, where $d(z_i) = 2 | E | -3$.
2. $V' = V_V \cup V_E \cup V_X \cup V_Y \cup V_Z$.
3. Edges connecting threads within each group.
   (a) Threads in V_X are connected by the following edges E_X=$\{< x_i, x_6 >| 0 \leq x_i \leq 5\} \cup \{< x_6, x_7 >\}$.
   (b) Threads in V_V are not connected within. Neither are threads in V_E, V_Y, V_Z.
4. Edges connecting threads between groups.
   (a) From V_V to V_X. $E_{VX} = \{< v_i, x_6 >| v_i \in V_V\}$.
   (b) From V_V to V_E. $E_{VE} = \{< v_i, e_j >| v_i \in V_V, e_j \in V_E, e_j$ is incident to v_i in G$\}$.
   (c) From V_Y to V_X. $E_{YX} = \{< y_i, x_6 >| y_i \in V_Y\}$.
   (d) From V_X to V_Z. $E_{XZ} = \{< x_6, z_i >| z_i \in V_Z\}$.
5. $E' = E_X \cup E_{VX} \cup E_{VE} \cup E_{TX} \cup E_{XZ}$. For each edge $e \in E'$, $cm(e) = 1$ or simply $cm = 1$.

It is clear that for each instance of the partition into triangles problem, Construction 1 delivers an instance of the basic thread scheduling problem. Note that $V' = V_V \cup V_E \cup V_X \cup V_Y \cup V_Z$ and $\mid V' \mid = 3 \mid E \mid - \mid V \mid +8$. Note also that $E' = E_X \cup E_{VX} \cup E_{VE} \cup E_{TX} \cup E_{XZ}$ and $\mid E' \mid = 4 \mid E \mid - \mid V \mid +7$. Note that the numbers used are also bounded by $2 \mid E \mid$. Hence, Construction 1 can be carried out in polynomial time. Figure 3 depicts an example of Construction 1. Figure 3(a) shows an instance of the partition into triangles problem. The overlay of Figures 3(b), 3(c) and 3(d) shows the dag built using Construction 1.



(a)

V_V, V_E, E_{VE}
(b)

(V_X, E_X)
(c)

$V_Y, V_Z, E_{VX}, E_{YX}, E_{XZ}$
(d)

Figure 3: An example of construction 1.

Lemma 1 *If the partition into triangle instance has a 'yes' answer, then the corresponding basic thread scheduling instance built using construction 1 has a 'yes' answer.*

Proof: We need to show there is a valid schedule σ whose finishing time is no more that K $(6 + 3 \mid E \mid)$. σ assigns each of the q partition to the first q PEs. It also assigns the three edges associated with each partition to the same PE that processes the partition. It then assigns threads in V_X to the next PE. For the remaining $\mid E \mid -3q$ PEs, σ assigns y_i, z_i, $1 \leq i \leq \mid E \mid -3q$, and one of the remaining $\mid E \mid -3q$ e-threads to each of them. Now we need to specify the starting time of each thread. Let $V_i = \{V_{i[1]}, V_{i[2]}, V_{i[3]}\}$ be a partition, and $\{e_{i[1]}, e_{i[2]}, e_{i[3]}\}$ be the three edges of E that belong to the partition. We have $\sigma_1(V_{i[1]}) = 0$, $\sigma_1(V_{i[2]}) = 2$, $\sigma_1(V_{i[3]}) = 4$, $\sigma_1(e_{i[1]}) = 6$, $\sigma_1(e_{i[2]}) =\mid E \mid +6$, $\sigma_1(e_{i[3]}) = 2 \mid E \mid +6$. Note that $e_{i[3]}$ will complete its processing by K. For each threads in V_X, $\sigma_1(x_i) = i$, $0 \leq i \leq 6$. Note that the finishing time of x_6 is $8+ \mid E \mid$, where $\mid E \mid$ is the sum of the communication overheads due to E_{VX} $(3q)$ and E_{YX} $(\mid E \mid -3q)$, i.e., $ex(x_6) =\mid E \mid$. $\sigma_1(x_7) =\mid E \mid +8$. Note that x_7 will complete it processing by K also. For each thread $y_i \in V_Y$, $\sigma_1(y_i) = 0$. For each thread $e_i \in V_E$ that does not belong to any partition, $\sigma_1(e_i) = 6$. Note that the finishing time of e_i is $6 + 2+ \mid E \mid$, where 2 is the overhead due to its two predecessors which are processed by other PE or PEs. For each thread $z_i \in V_Z$, $\sigma_1(z_i) = 8+ \mid E \mid$. Note that z_i will complete its processing by K as well.

It is straight forward to verify that σ satisfies conditions (1) and (2) in the definition of a valid schedule and thus is a valid schedule. Since the finishing time of σ is K, the corresponding basic thread scheduling instance has a 'yes' answer. The lemma is proved. $\square$

To prove that Construction 1 is indeed a reduction from partition into triangles to basic thread scheduling, we must show the converse of Lemma 1. The next set of lemmas and observations is developed for that purpose.

All y-threads are predecessors of x_6. Since $d(y_i) = 6$ for $y_i \in V_Y$, any valid schedule σ must satisfy $\sigma_1(x_6) \geq 6$.

Observation 1 *Let σ be any valid schedule. $\sigma_1(x_6) \geq 6$.*

Suppose a valid schedule assigns starting time 6 to x_6. Between time slots 0 to 5, the actions of the PE that processes x_6 can be classified into three groups: doing nothing (idle), always busy, and partially busy. Let us consider the finishing time of x_6 for the three categories above.

For doing nothing case, the finishing time of x_6 is at least $14+ \mid E \mid$. Note that the overheads from V_X is 6, from V_V is $3q$, and from V_Y is $\mid E \mid -3q$.

The always busy case results in the following sub-cases. Each of which is considered next.

1. The PE processes one y-thread. In this case the finishing time of x_6 is at least $6 + (6 + 3q+ \mid E \mid -3q - 1) + 2 = 13+ \mid E \mid$.
2. The PE processes three v-threads. In this case, the finishing time of x_6 is at least $6 + (6 + 3q - 3+ \mid E \mid -3q) + 2 = 11+ \mid E \mid$.
3. The PE processes six x-threads. In this case, the finishing time of x_6 is at least $6 + (3q+ \mid E \mid -3q) + 2 = 8+ \mid E \mid$.
4. The PE processes a combination of v-threads and x-threads. The combination would be 1 v-thread and 4 x-threads or 2 v-threads and 2 x-threads. The respective finishing times are at least $6 + (2 + 3q - 1+ \mid E \mid -3q) + 2 = 9+ \mid E \mid$ and $6 + (4 + 3q - 2+ \mid E \mid) + 2 = 10+ \mid E \mid$.

For the partially busy case and doing nothing case, the finishing time of x_6 cannot be earlier that $8+ \mid E \mid$, which is the best possible finishing time for the always busy case.

Hence, when a valid schedule σ assigns starting 6 to x_6, the earliest finishing time possible for x_6 is $8+ \mid E \mid$. Can a valid schedule reduce the finishing time of x_6 to be earlier that $8+ \mid E \mid$ by starting x_6 later than 6? As far as minimizing the finishing time of x_6 is concerned, the only advantage of delaying the starting time of x_6 is to have more predecessors of x_6 processed by the same PE that processes x_6 so that the communication overheads are reduced. Since each thread's processing time is no less that the cost of communication ($cm = 1$) for our instance, the answer to the above question is no. We summarize the preceding discussion in the next observation.

Observation 2 *Under any valid schedule, the earliest possible finishing time of x_6 is $8+ \mid E \mid$, when x_i, $0 \leq i \leq 6$ are processed by the same PE.*

¿From Construction 1, $d(x_7) = 2 \mid E \mid -2$. Any valid schedule σ that completes x_7 by K must satisfy $\sigma_1(x_7) \leq 8+ \mid E \mid$. Since x_6, the only predecessor of x_7, has $8+ \mid E \mid$ as its earliest finishing time (Observation 2), σ must satisfy $\sigma_1(x_7) = 8+ \mid E \mid$ and $\sigma_2(x_6) = \sigma_2(x_7)$.

Lemma 2 *Any valid schedule σ having finishing time no more that K must assign x-threads to one PE. Furthermore, $\sigma_1(x_6) = 6$ and $\sigma_1(x_7) = 8+ \mid E \mid$.*

Proof: The lemma follows from observation 2 and the preceding discussion. □

Henceforth let σ denote any valid schedule having finishing time K. From observation 2, we know at least one PE is busy from time slots 0 to 5 in σ. Let us consider the scheduling of the remaining $\mid E \mid -2q$ PEs under σ.

Lemma 3 *In σ between time slots 0 to 5, each of the $\mid E \mid -3q$ PEs processes one y-thread and each of the remaining q PEs processes three v-threads.*

Proof: From lemma 2, we have $\sigma_1(x_6) = 6$. This implies that v-threads and y-threads, which are predecessors of x_6, must be completed (before 6) in time slots 0 to 5 in σ. Since $d(y_i) = 6$ for each y-thread y_i and $\mid V_Y \mid = \mid E \mid -3q$, $\mid E \mid -3q$ many PEs must each processes a y_i from 0 to 5 in σ. The remaining q PEs must each processes three v-threads in σ because $\mid V_V \mid = 3q$, $d(v_i) = 2$ for $v_i \in V_V$, and v-threads are not connected to one another. □

Let us call the PE that processes v-thread, x-thread, and y-thread under σ, respectively, v-PE, x-PE, and y-PE.

¿From lemmas 2 and 3, under σ every PE is busy before 6. So the earliest starting time of a e-thread is 6 in σ.

Observation 3 $\sigma_1(e_i) \geq 6$ *for* $e_i \in V_E$.

Besides e-threads, other threads that have not been considered are z-threads. From construction 1, each z-thread has x_6 as its only predecessor. From lemma 2, we know that under σ the PE that processes a z-thread has to be different from the PE that processes x_6. Since $d(z_i) = 2 \mid E \mid -3$, the starting time of z_i in σ must be no later than $K - (2 \mid E \mid -3) - 1 = 8 + \mid E \mid$. From observation 2, we have the following observation.

Observation 4 $\sigma_1(z_i) = 8 + \mid E \mid$ *for* $z_i \in V_Z$ *and no PE can process more than one z-thread in* σ.

The next observation is based on observation 3 and $d(e_i) = \mid E \mid$ in construction 1.

Observation 5 *Each v-PE can process at most 3 e-threads in* σ.

Suppose a v-PE, P, processes three e-threads in σ. Then one e-thread must start at 6, the next at $6 + \mid E \mid$ and $6 + 2 \mid E \mid$ respectively, otherwise the finishing time requirement of σ cannot be met. In order to complete the thread that start at 6 and before $6 + \mid E \mid$, two of its only predecessor v-threads have to be processed by P. Similarly, to complete the next two threads before $6 + 2 \mid E \mid$ and $6 + 3 \mid E \mid$ respectively, their predecessor v-threads have also to be processed by P. From the correspondence given in Construction 1, the threads processed by P correspond to a triangle in G.

Observation 6 *In* σ, *whenever a v-PE processes three e-threads, the threads processed by the v-PE correspond to a triangle in* G.

Observation 7 *In* σ *each y-PE can process at most one arbitrary e-thread along with an arbitrary z-thread.*

Observation 7 is based on observations 3 and 4 and the fact that each e-thread has two predecessors. Although the two predecessors are processed by v-PEs, under σ y-PE can start the e-thread at 6 and finish it by $6+ \mid E \mid +2 = 8+ \mid E \mid$.

In σ if the only other threads that are processed by a y-PE are e-threads, then the first e-thread can be completed no earlier than $8+ \mid E \mid$ and the next no earlier than $10+ \mid E \mid$. At this point, on other e-thread can be processed without violating the finishing time K.

Observation 8 *In σ each y-PE can process at most two e-threads.*

Lemma 4 *In σ, no z-thread is processed by an v-PE.*

Proof: Suppose L many z-threads are processed by v-PEs in σ, where $L \geq 1$. From observation 4, we must have L v-PEs each processes one z-thread. Let us consider the maximum number of e-thread that can be processed here.

The L v-PEs that process z-thread can process at most L e-threads because of observation 4 and $d(e_i) =\mid E \mid$. From observation 5, the remaining v-PEs can process at most $3(q - L)$ e-threads. Since the remaining $\mid E \mid -3q - L$ z-threads are processed by y-PEs, from observation 4, $\mid E \mid -3q - L$ y-PEs must be used for this task. From observation 7, these y-PEs can process at most $\mid E \mid -3q - L$ e-threads. Based on observation 8, the remaining L y-PEs can process at most $2L$ e-threads. So the total number of e-threads that can possibly be processed is $L + 3(q - L)+ \mid E \mid -3q - L + 2L = \mid E \mid -L$. Since $L \geq 1$, in σ the maximum number of e-threads that can processed is less than $\mid E \mid$. This contradicts to σ being a valid schedule having finishing time K. Hence, the lemma is proved. □

Lemma 5 *If the basic thread scheduling instance built using construction 1 has a 'yes' answer, then the corresponding partition into triangle instance has a 'yes' answer.*

Proof: The 'yes' answer for the basic thread scheduling instance implies that there exists σ. Based on lemmas 2,3 and 4, σ must assign one PE for all x-threads, q PEs as v-PE, and the remaining $\mid E \mid -3q$ PEs as y-PEs that process all y-threads and z-threads. Based on observations 5 and 7, in order to process all e-threads, each v-PE must process 3 e-threads and each y-PE 1 e-thread. From observation 6, the three v-threads and the three e-threads processed by each v-PE corresponds to a triangle in G. Since we have q such v-PEs, the corresponding partition into triangle instance has a 'yes' answer. □

Theorem 1 *The basic thread scheduling problem is NP-complete in the strong sense even when $cm = 1$.*

Proof: It follows from construction 1, lemmas 1 and 5, and the fact that basic thread scheduling problem is in the NP class. □

¿From the formulation of the thread scheduling problems in Section 2, instances of the basic thread scheduling problem are instances of the thread scheduling problem. This leads to the next theorem.

Theorem 2 *The thread scheduling problem is NP-complete in the strong sense even when* $cm(e) = 1$ *for every edge.*

By using the above idea, replacing each thread of size K by a chain of K unit threads, and a slightly more complicated construction and analysis, the next theorem can be established [9].

Theorem 3 *The basic thread scheduling problem is NP-complete in the strong sense even when* $cm = 1$ *and each thread is a unit thread (having a unit processing time).*

The complexity results of related scheduling problems has been considered by other researchers [12, 16, 18]. The complexity results presented cannot be implied by the previous results mentioned above.

4 Analysis and transformation techniques

In this section, we propose and discuss several techniques which are used to develop several heuristic algorithms [10] that solve the basic thread scheduling problem. These techniques not only deal with issues of heuristic algorithm design and development, they also address some of the questions and concerns raised by other researchers and improve some of the techniques in [17].

4.1 Critical path analysis

Critical path analysis is used to predict ideal execution time (assuming no communication overhead between threads) [17]. The ideal execution time is used to evaluate the theoretical speedup of various scheduling techniques and algorithms. A stack based critical path analysis algorithm is given in [17]. Besides using an extra space (worst case $O(n)$, where n is the number of nodes in the dag), based on our careful analysis of the algorithm, the algorithm has $O(\mid E \mid^2)$ as its worst case time complexity instead of $O(\mid E \mid)$. In what follows, we give a simple algorithm with worst case time complexity of $O(n + \mid E \mid)$ to solve the same problem. Based on the topological ordering of the vertices [6], the algorithm eliminates the need of the stack. In [17], the term level is used to refer to a thread's earliest completion time. So each thread v has a level, level(v). We shall use the same term for ease of cross-referencing.

The following recurrence formula is used to compute level(v) of each vertex v in the dag.

$$
\begin{aligned}
level(v) &= d(v), \text{ if } PRED(v) = \emptyset \\
&= \max_{u \in PRED(v)} \{level(u)\} + d(v) \text{ if } PRED(v) \neq \emptyset
\end{aligned} \tag{1}
$$

Note that due to communication ($cm > 0$) we cannot assume a single source vertex (a vertex is called a source vertex if its indegree is zero) by the technique of adding a new dummy vertex. Since each path from source vertices to v must go through a predecessor of v, the longest path from source vertices to v must consists of two parts; the first part is a longest path from source vertices to u, where u is a predecessor of v, and the second part is $d(v)$. The recurrence formula captures the above observation and correctly computes level(v) for each v. Note that the maximum level of the sink vertices (vertices with outdegree zero) is the length of the critical path.

The stack based algorithm computes level(v) by incrementally updating its value. The value may be updated many times. Each time the value is updated, the vertex is pushed onto the stack and the level value of each node in the entire subgraph form v to the sink vertices is recomputed. The above observations lead to a more accurate analysis of the algorithm and can be used to construct examples for which the stack based algorithm runs in $O(\mid E \mid^2)$ time, not $O(\mid E \mid)$. This changes the worst case time complexity of the stack based algorithm from $O(\mid E \mid)$, as reported in [17], to $O(\mid E \mid^2)$.

To compute level(v), we need to know level(u) for $u \in PRED(v)$. To ensure the level value of each predecessor is available, the computation of a vertex's level should be carried out in topological order[1].

Let $v1, v2, \ldots, vn$ be a topological order of vertices in G. The algorithm is stated in pseudo code below. The algorithm is based on ideas used in PERT network [8].

```
Algorithm 1: for computing level(v) and critical path
   for (i=1; i <= n; i++) {
     if vi does not have any immediate predecessor
        level(vi) = d(vi);
     else {
        level(vi) = 0;
        for each immediate predecessor u of vi do
          if level(vi) < level(u)
             level(vi) = level(u);
        level(vi) = level(vi) + d(vi); } }
```

[1] We do not need to first compute the topological order. With appropriate data structures it can be computed on the fly.

4.2 Critical network

A dag G, in general, may have many critical paths. A thread is called a critical thread if it belongs to a critical path. The critical network of G is the maximum subgraph of G such that every source to sink path in the subgraph is a critical path in G. An example dag and its critical network are shown in Figure 4 (a) and (b), where the letter inside the circle denotes the vertex and the number denotes the processing time. The two are separated by a colon. An application of critical network for better ideal execution prediction is given in the next subsection. We shall now illustrate the method for critical thread identification as well as critical network identification.

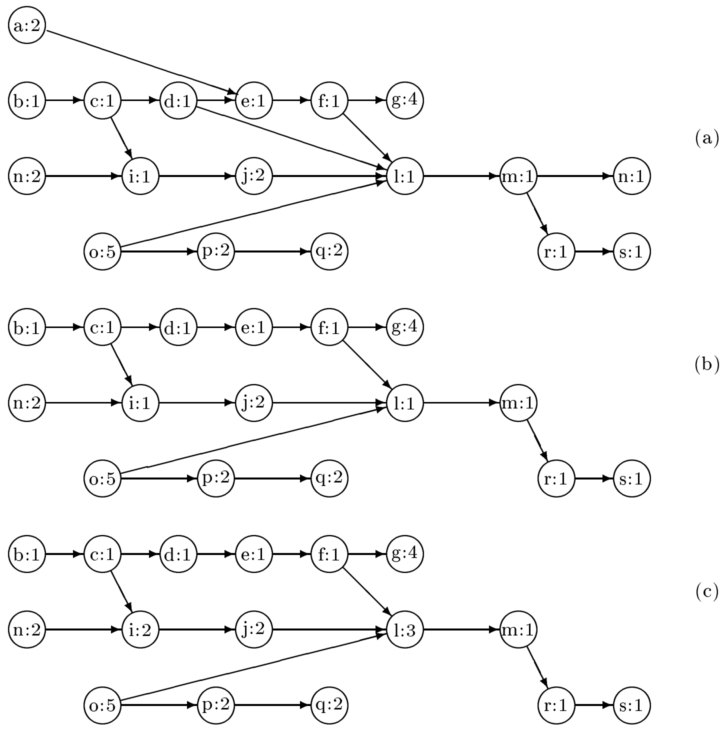


Figure 4: An example of critical network and its transformation.

Let rlevel(u) denote the latest starting time of thread u such that the theoretical runtime is not affected. The following recurrence formula computes rlevel(u), where length stands for critical path length.

$$
\begin{aligned}
rlevel(u) &= length - d(u), \text{ if } SUCC(u) = \emptyset \\
&= \min_{v \in SUCC(u)} \{rlevel(v)\} - d(u) \text{ if } SUCC(u) \neq \emptyset
\end{aligned} \tag{2}
$$

To use the above formula, we must first compute the length of the critical path. The algorithm presented in the previous section can be used to compute that. Notice the symmetry between formula (1) and (2). By consider each thread in reverse topological order, we can compute the rlevel(u) for each u in order $O(n+e)$ time. The algorithm is very similar to Algorithm 1 and is omitted.

Definition 5 $C(v) = level(v) - rlevel(v) - d(v)$. *$C(v)$ is called the criticality of thread v.*

If $C(v) = 0$, then v is a critical thread. It is straight forward to see that $C(v) = 0$ iff v belongs to a critical path. Notice that this is obviously true if v is a sink vertex. For other vertices, we use induction to establish $C(v) = 0$ iff v belongs to a critical path.

Once we know level(v) and rlevel(v), we can compute $C(v)$ and determine the critical thread. The critical network is constructed by including all critical threads and those edges of the dag that belong to critical paths. An edge $< u, v >$ belongs to a critical path iff $level(v) - rlevel(u) - d(v) - d(u) = 0$.

Definition 6 *Given a dag G, an edge $\alpha =< u, v >$ is <u>redundant</u> if the graph contains a directed path from u to v which does not include α. G is <u>irreducible</u> iff there is no redundant edge in G.*

Lemma 6 *The critical network of a program thread graph is irreducible.*

Proof: If the critical network, N, is not irreducible, then there is a redundant edge $\alpha =< u, v >$ in N. Let $p = u, v_{i1}, \ldots, v_{ik}, v$, $k \geq 1$, be the directed path from u to v which does not include α. Since $d(v_{ij}) > 0$, $1 \leq j \leq k$, we cannot have $level(v) - rlevel(u) - d(v) - d(u) = 0$. Hence, α cannot belong to N. It is a contradiction. □

Let $p = v_1, v_2, \ldots, v_k$ be a path of a irreducible dag. Each v_i, $2 \leq i \leq k$, satisfies the property that it has exactly one immediate predecessor in p. This property of the critical network will be used later.

4.3 Critical network transformation

Lower bounds of the execution time of the dag can be used to evaluate the theoretical speedup of various scheduling techniques and can be used to evaluate heuristic scheduling algorithms. In previous approaches [17], the ideal execution time of a critical path is used as the lower bound. Since the critical network is a subgraph of the dag, the 'ideal' execution time (lower bound of the execution time) of the critical network is also a lower bound of the execution time of the dag. Using critical network and the following transformation, we show that a bound which is tighter than that given by the critical path method can be obtained.

Just as in [17], we shall assume maximum achievable parallelism, i.e., sufficient available PEs. The transformation assumes $d(v) \geq cm$ for all critical threads v, i.e., the execution time of each critical thread is greater than the communication time between threads. This is a valid assumption for any reasonable thread granularity and well designed multi-threaded architectures.

Transformation 1: Let $N = (V_N, E_N, t_N, cm_N)$ be a critical network. Let ind(v) be the indegree of $v \in V_N$. We transform N to another dag $A = (V_A, E_A, t_A, cm_A)$, where $V_N = V_A$, $E_N = E_A$, $t_A(v) = (ind(v) - 1)cm_N + t_N(v)$ for each v, and $cm_A = cm_N$. Figure 4(b) and Figure 4(c) depict N and A respectively using this transformation.

Lemma 7 *The ideal execution time of a critical path in A is a lower bound of the completion time of N.*

Proof: Let $p = v_1, v_2, \ldots, v_k$ be a critical path in A. Note that p is also a critical path in N. Let us consider the execution of p along with other threads of N. Since p is a path of N, the earliest completion time of p is no more than the earliest completion time of N. Since $d(v) \geq cm$, for $v \in N$, to finish the execution of p in the earliest possible time is to assign p to one PE and process the threads of p one after another. Let $f(v_i)$ be the best possible finishing time of v_i and $l_A(v_i)$ be the level(v_i) in A.

Note that $f(v_1) \geq d_N(v_1)$ and $l_A(v_1) = d_A(v_1) = d_N(v_1)$. Hence, $f(v_1) \geq l_A(v_1)$. Since p is a path of a irreducible dag N, each v_i, $2 \leq i \leq k$, has exactly one predecessor in p. Other predecessors of v_i are processed by other PEs. Thus, $f(v_2) \geq f(v_1) + (ind(v_2) - 1)cm_N + d_N(v_2)$. From transformation 1, $d_A(v_2) = (ind(v_2) - 1)cm_N + d_N(v_2)$. Since p is a critical path in A, $l_A(v_2) = l_A(v_1) + d_A(v_2)$. Hence, $f(v_2) \geq l_A(v_2)$. By repeatedly using the same argument, we have $f(v_i) \geq l_A(v_i)$, $1 \leq i \leq k$. Note that $l_A(v_k)$ is the ideal execution time of a critical path of A, and $f(v_k) \geq l_A(v_k)$ implies $l_A(v_k)$ is a lower bound of the completion time of N. $\square$

It follows from the above lemma and the discussion that precedes it that we can compute a lower bound of the execution time of the dag, by first constructing the critical network N, then transforming N to A using transformation 1, and finally computing the critical path of A[2].

[2] The steps can be combined in the implementation.

Let us consider the example of Figure 4. The lower bound given by the critical path method is 9 and the lower bound given by the critical network transformation method is 12. The critical network transformation method always provides a tighter lower bound than that given by the critical path method.

4.4 Thread packing

The thread packing technique aims at reducing the number of PEs needed to complete the set of thread sequences obtained from the program thread graph. Some thread sequence formation algorithms such as the one proposed in [17] assume that maximum parallelism can be achieved. Under this assumption, each PE is responsible for the execution of one thread sequence. Once the thread sequences are formed and assigned to each PE, the execution time interval of each thread sequence can be specified and the completion time of all tread sequences can be determined. In thread packing, we are interested in achieving the same finishing time as the current schedule but with the least possible number of PEs.

The problem of thread packing can be modeled as an interval packing problem, where each thread sequence s is represented by an interval $[s(b), s(e)]$. $s(b)$ denotes the beginning time slot of the processing and $s(e)$ denotes the ending time slot. Two thread sequence may be packed together if their corresponding intervals do not overlap. Several thread sequences may be packed together if there corresponding intervals are pairwise nonoverlopping. A simple packing example is shown in Figure 5. Since the packing process does not modify any of the intervals, the finishing time is not affected. As illustrated in the example, the packing process can reduce the number of PEs needed. Using an algorithm similar to the left-edge algorithm used in [5, 11], the maximum PE reduction can be achieved.

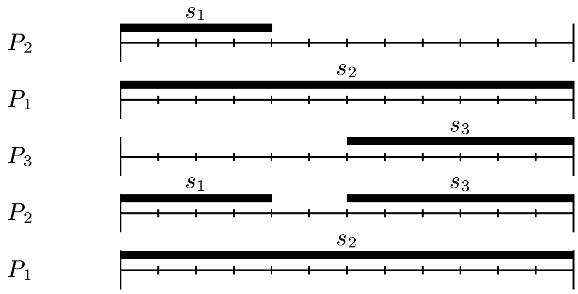


Figure 5: An example of thread packing.

5 Conclusion

The thread scheduling problem is considered in this paper. The thread scheduling problem abstracts the problem of minimizing memory latency, using a directed data dependency graph generated from a compiler, to reduce the finishing time. Two thread scheduling problems are formulated and shown to be NP-complete in the strong sense. New methods and algorithms for analyzing a data dependency graph in order to compute the theoretical best runtime (lower bound of the finishing time) and to estimate the required minimum number of PEs needed to achieve certain finishing time are presented. The new methods and algorithms improve upon some of the analysis and transformation techniques introduced in [17]

These methods and algorithms are efficient and thus together other techniques [10, 17] can provide a practical optimization phase in the compiler for the defined architecture.

References

[1] *IF1 An Intermediate Form for Applicative Languages*, reference manual version 1.0 edition, Univertisy of California-Davis 1985.

[2] M.C. Chang and F. Lai. Efficient exploitation of instruction-level parallelism for superscalar processors by the conjugate register file scheme. *IEEE Transaction on Computers*, 45(3):278–93, 1994.

[3] J. T. Feo. An analysis of the computational and parallel complexity of the livermore loops. *Parallel Computing*, pages 163–185, July 1988.

[4] M.R. Garey and D.S. Johnson. *Computers and Intractability.* Freeman, San Francisco, CA, 1979.

[5] A. Hashimoto and J. Stevens. Wire routing by optimizing channel assignment within large apertures. In *Proceedings of 8th Design Automation Conference*, pages 155–169, 1971.
[6] E. Horowitz and S. Sahni. *Fundamentals of Data Structures in PASCAL*. Computer Science Press, New York, NY, 1994.
[7] D. Kuck et al. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM Symposium on Principles of Programming Languages*, pages 207–218, January 1981.
[8] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart, and Winston, 1976.
[9] W. N. Li. Manuscript in preparation.
[10] W. N. Li and J. F. Jenq. Manuscript in preparation.
[11] W. N. Li and Sartaj Sahni. Pull up transistor folding. *IEEE Transactions on Computer-Aided Design*, 9(5):512–521, May 1990.
[12] C.H. Papadimitriou and M. Yannakakis. Towards an architecture-independent analysis of parallel algorithms. *SIAM J. Comput.*, 19(2):322–328, April 1990.
[13] V. Sarkar. *Partitioning and Scheduling Parallel programs for execution on Multiprocessors*. MIT Press, Cambridge, MA, 1989.
[14] V. Sarkar and J. Hennessy. Compile-time partitioning and scheduling of parallel programs. In *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, pages 17–26, July 1986.
[15] B. Simons, V. Sarkar, Breternitz Jr. M., and M. Lai. An optional asynchronous scheduling algorithm for software cache consistency. In *Proceedings of the Hawaii International Conference on Systems Sciences*, pages 502–511, 1994.
[16] V.J. Rayward Smith. Uet scheduling with unit interprocessor communication delays. *Discrete Applied Mathematics*, 18:55–71, 1987.
[17] M.A. Thornton and D.L. Andrews. Graph analysis and transformation techniques for runtime minimization in multi-threaded architectures. In *Proceedings of the Hawaii International Conference on Systems Sciences*, pages 566–575, 1997.
[18] J. Ullman. Np-complete scheduling problems. *J. Comput. System Sci.*, 10:384–393, 1975.
[19] M. Y. Wu and D. Gajski. Hypertool: A programming aid for message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):330–343, July 1990.
[20] T. Yang and A. Gerasoulis. Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, September 1994.