University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

Thread Partitioning and Scheduling Based on Cost Model

Xinan Tang †
Jian Wang †
Kevin B. Theobald †
Guang R. Gao

CAPSL Technical Memo 03
April 15, 1997

A condensed version appears in the Proceedings of the Ninth Annual ACM Symposium on Parallel Algorithms and Architectures, Newport, RI, June 1997
Also published as ACAPS Technical Memo 106, School of Computer Science, McGill University, Montréal, Québec, Canada

† School of Computer Science, McGill University, Montréal, Québec, Canada

University of Delaware • 140 Evans Hall • Newark, Delaware 19716 • USA
http://www.capsl.udel.edu • ftp://ftp.capsl.udel.edu • capsladm@capsl.udel.edu

Abstract

There has been considerable interest in implementing a multithreaded program execution and architecture model on a multiprocessor whose primary processors consist of today's off-the-shelf microprocessors. Unlike some custom-designed multithreaded processor architectures, which can interleave multiple threads concurrently, conventional processors can only execute one thread at a time. This presents a unique and challenging problem to the compiler: partition a program into threads so that it executes both correctly and in minimal time.

We present a new heuristic algorithm based on an interesting extension of the classical list scheduling algorithm. Based on a cost model, our algorithm groups instructions into threads by considering the trade-offs among parallelism, latency tolerance, thread switching costs and sequential execution efficiency. The proposed algorithm has been implemented, and its performance measured through experiments on a variety of architecture parameters and a wide range of program parameters. The results show that the proposed algorithm is robust, effective, and efficient.

Contents

List of Figures

List of Tables

1 Introduction

With the relentless progress of microprocessor technology, CPU speeds have increased dramatically. As this trend continues, interprocessor latencies will become an even greater impediment to getting maximal performance from parallel systems. Multithreaded processors can address this problem, because they can perform useful computations during these interprocessor latencies. In a multithreaded system, a program is partitioned into sequences of instructions called *threads*. Instructions within a thread are executed consistent with their sequential order, and threads are scheduled dynamically. If there is a long-latency operation, such as a fetch from a remote memory, the processor can switch to another thread and do some other work during the interim.

This is a familiar concept in operating systems, where multithreading has long been used to tolerate I/O latencies [32]. Kernel threads have been built into the OS kernel in modern microkernel-based systems such as Amoeba [22], Mach [1], and Chorus [13]. Recently, user-level threads have been extensively used in some applications, such as Internet servers. However, these instances of multithreading are intended to tolerate latencies much larger than simple interprocessor latencies (e.g., disk accesses). Therefore, threads are typically large. Multithreaded *processors,* on the other hand, tend to deal with much finer threads, so these threads are much more numerous. It is too much of a burden for the programmer to partition the program into many threads manually, so effective compiler support is essential for multithreaded programming. Therefore, it is our goal to develop efficient thread-partitioning algorithms for compilers.

Multithreaded processors can be classified according to whether their threads are *preemptive* or *non-preemptive*. A preemptive thread can be suspended in the middle of execution. In machines such as HEP [21] and Tera [4], the processor switches threads regularly, such as every cycle. Other processors, such as Sparcle [3], stay in one thread until a long-latency operation occurs, and then switch to another thread. In either case, processor design can be quite difficult. By contrast, a non-preemptive thread, once started, runs to completion without any other threads intervening. This makes hardware design considerably easier, and amenable to implementation with off-the-shelf microprocessors [18]. However, this means that before a thread can begin execution, all data needed by that thread must be ready and present in the processor, so that the thread is not stalled by any long-latency operations. In particular, all remote operations must be handled using *split-phase transactions* in which one thread initiates the operation, while the operation that uses the returned value is performed in another thread, which is not started until the remote operation finishes.

We are interested in the latter class of processors, so our goal is as follows: Given a parallel system based on non-preemptive multithreaded processors, in which every operation is either "local" or "remote" (the latter of which can be overlapped with other operations), develop an algorithm for partitioning user programs into threads, formulating all remote operations as split-phase transactions and minimizing total execution time.

```
S1:  j = A[i];  /* (1, 10) */
S2:  u = x/2;   /* (6,  0) */
S3:  v = x/3;   /* (6,  0) */
S4:  k = B[i];  /* (1,  7) */
S5:  y = C[j];  /* (1, 10) */
S6:  z = D[k];  /* (1, 10) */
S7:  w = u+v+y; /* (1,  0) */
S8:  w = w+z;   /* (1,  0) */
```

(a) Original Sequence of Statements

```
Thread_1: /* Start at cycle 0 */
      S1: j = A[i];
      S4: k = B[i];
      S2: u = x/2;
          End_Thread;
Thread_2: /* Start at cycle 10 */
      S6: z = D[k] ;
          End_Thread;
Thread_3: /* Start at cycle 12 */
      S5: y = C[j];
      S3: v = x/3;
          End_Thread;
Thread_4: /* Start at cycle 24 */
      S7: w = u+v+y;
      S8: w = w + z;
```

(b) Optimally Threaded Code

Figure 1: An example of thread partition

1.1 An Example of Thread Partitioning

The following example illustrates the thread-partitioning problem which we are exploring in this paper. Figure 1 shows a sequence of statements $S_1 \ldots S_8$. Each statement has two costs, which are listed in the comments: the *local* cost (left number) and the *remote* cost (right). The former is the time required by the CPU to perform the operation, and the latter is the total time spent waiting for the remote operation to finish, including network latencies and synchronization delays. For example, assuming all array accesses are *remote*, a processor executing statement S_1 needs one cycle to issue the read request and then must wait another 10 cycles for the value of `A[i]` to return from another processor. On the other hand, since S_2 is an ALU operation, it takes 6 local cycles to finish and its remote cost is zero.

In a purely sequential processor, the total execution time of this program section would be the sum of all local and remote costs (55 cycles), because the processor must idle during the remote portions of operations. If the processor could continue to execute while remote loads are pending, then S_2 and S_3 could run while the remote load from S_1 is in progress. However, the CPU would still stall upon reaching S_7, as the value fetched by S_5 (required by S_7) would not be available yet.

This is where the benefits of multithreading come in: First, by rapidly switching to another ready thread, the processor can hide remote latencies. In Figure 1(a), after the CPU executes S_6 in thread 2, it can switch to thread 3 to execute S_5 without waiting for S_6 to finish. Second, threads are dynamically scheduled by the availability of data. Should S_5 require more than 10 cycles (due to unforeseen latencies), thread 4 would not start at cycle 24 but would wait until S_5 finishes.

As shown in Figure 1(b), the original program is partitioned into four threads. There are four *remote* statements, S_1, S_4, S_5, and S_6, which perform split-phase transactions. Statements S_5 and S_6 are separated from S_7 and S_8 because the latter depend on data produced by the former. For the same reason, S_1 and S_4 are separated from S_5 and S_6. This means at least three threads are required. Furthermore, it would not be optimal to put S_5 and S_6 in the same thread, because under a non-preemptive thread model, this thread would not be able to run until both `j` and `k` were ready. Putting S_5 and S_6 into separate threads allows S_6 to start while `A[i]` is still being loaded.

Local statements S_2 and S_3 are scheduled so as to fill in the gaps otherwise spent waiting for remote latencies. The *End_Thread* instruction at the end of each thread completes the current thread and allows the processor to start a new thread. Assuming in this example that thread switching costs one cycle, the comments in Figure 1(b) show the starting time of each thread. This is an optimal partition, which takes 26 cycles to run (see Section 4 for details).

The example above shows that optimal thread scheduling (that minimizes total execution time) can be a complex problem. Subject to constraints such as split-phase requirements, thread partitioning implies trade-offs between *parallelism* (small concurrent threads), *switching costs* (merging threads), *sequential efficiency* (large sequential threads), and *latency tolerance* (ordering and distributing local nodes).

1.2 Synopsis

It is a challenge to generate code that can effectively utilize the underlying resources of multithreaded machines. It is even more demanding to generate threaded code automatically from user programs. In this paper, we study the *static* thread partitioning approach for non-preemptive threads. Our focus is on automatically partitioning user programs written in conventional imperative languages, and statically scheduling threads to run with a minimal total execution time.

There is scant research on thread partitioning (a survey is provided in Section 6). If multithreaded machines are to gain wide acceptance, then there is an urgent need to investigate this topic further. The main contribution of this paper is a heuristic algorithm for partitioning programs into threads and for scheduling operations within threads. We investigate a novel approach of combining the partitioning and scheduling steps together based on a thread-execution cost model. In the design of our algorithm, we effectively apply proven heuristics used in *instruction scheduling* and *m-processor scheduling* to thread partitioning. Our algorithm is based on an interesting extension of the classical list scheduling algorithm, and it groups operations into threads by balancing the following goals: parallelism exploitation, latency tolerance, minimization of thread switching costs, and sequential execution efficiency. The experimental results show our algorithm is robust, efficient, and effective for a wide range of architecture and program parameters and thus it provides a general framework for thread partitioning.

In the next section, we build a thread-execution cost model by introducing costs for statements. In Section 3, we describe the thread partitioning problem in detail and formulate it as an optimization problem. We then propose a list-scheduling based heuristic algorithm to solve this problem in Section 4 and report the experimental results in Section 5. Finally, related work is briefly reviewed in Section 6, and future work is discussed in Section 7.

2 Multithreaded Execution Model

Many multithreaded architectures have been built or proposed, including HEP [21], Tera [4], *T [23], StartT-NG [5], TAM [8], RWC-1 [26], and EARTH [17] (most of which have been decribed in a survey paper [9]). Some of these machines use preemptive threads, which, as suggested in Section 1, require thread-scheduling strategies different from non-preemptive processors. Our work is primarily focused on non-preemptive processors and influenced heavily by the EARTH model [18]. However, the following abstract multithreaded execution model and abstract graph-based program representation, on which our algorithm is based, leave out implementation details specific to EARTH. In this way, our results are useful for other non-preemptive computers besides EARTH.

2.1 Processor Execution Model

We assume that the multithreaded system consists of a number of processing elements (each with its own memory), an interconnection network, and support for the following multithreading features:

1. Remote requests are executed and dispatched by a CPU quickly, so that it can move on to other computations while the request is being sent over the network.

2. Computation and communication can proceed simultaneously. When a CPU is executing a thread, communication to this and other threads occurs in parallel without CPU intervention.

3. Threads are dynamically synchronized and scheduled by an independent synchronization mechanism. Because threads are non-preemptive, the synchronization mechanism must guarantee that no thread gets started until all the required data is in local memory.

Even though our model does not *require* hardware support for all three features above, such support would yield the best system performance. Since most multithreaded machines require extra cycles to switch from one thread to another, we use δ to denote this cost (in cycles). As with the other costs of multithreading, δ is likely to be lower in machines with good hardware multithreading support.

In the EARTH model [17, 18], threads are executed by an off-the-shelf processor called the *Execution Unit* (EU), and synchronization and communication tasks are performed by a special co-processor called the *Synchronization Unit* (SU). A remote request is first executed by the

EU, then dispatched to the SU. It is the SU who finishes this split-phase operation, including network communication and thread synchronization.

2.2 Program Modeling

In order to be independent of a particular language, the input program is represented as a hierarchy of Data Dependence Graphs (DDGs), in which each level is a Directed Acyclic Graph (DAG). For each graph $G = (V, E)$, each node in V corresponds to a program statement. A node can be a *simple* node corresponding to an assignment statement or a *compound* node corresponding to a compound statement such as a loop. A compound node may have one or more additional graphs representing the internal components of the compound statement A node is called *remote* if the corresponding statement accesses remote memory; otherwise, it is *local*. Once the program has been divided among the processors, the DDGs for a particular processor can be derived. After the data distribution is known, the compiler can determine whether each node is remote or local.

In the following, we assume a DDG with $|V|$ nodes, $\{v_1, \ldots, v_{|V|}\}$. Each node v_i has a *local cost* c_i^L and a *remote cost* c_i^R, defined as follows:

$$
\begin{aligned}
c_i^L &= \begin{cases} \text{cycles spent in EU} & \text{if } v_i \text{ local} \\ \text{time for EU to dispatch to SU} & \text{if } v_i \text{ remote} \end{cases} \\
c_i^R &= \begin{cases} 0 & \text{if } v_i \text{ local} \\ \text{time from dispatch to finish} & \text{if } v_i \text{ remote} \end{cases}
\end{aligned}
$$

Rather than representing a remote operation as a single entity, we separate it into remote and local components and assign costs accordingly. The advantages of such a separation are 1) split-phase transactions can be modeled, and 2) remote costs are clearly identified.

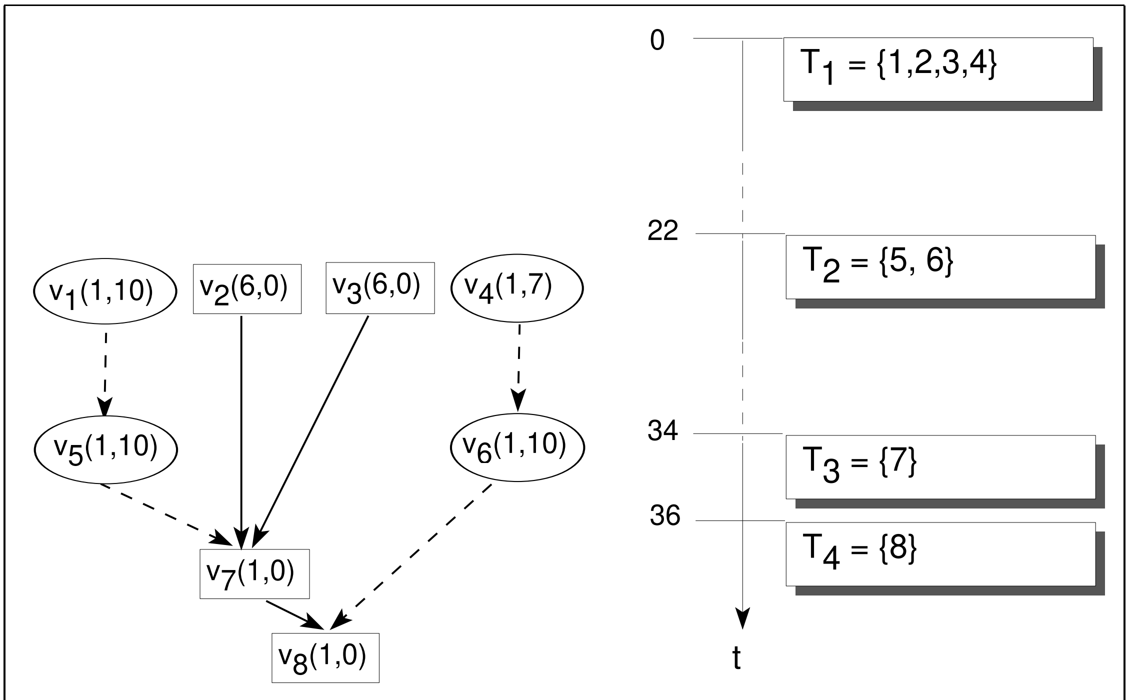
If there is a data dependence from v_i to v_j, then there is an edge (v_i, v_j) in E. All out-going edges from a remote node are called *remote*, from a local node *local*. Edges can be represented as an adjacency matrix:

$$
C_{i,j} = \begin{cases} 1 & \text{if } (v_i, v_j) \in E \\ 0 & \text{otherwise} \end{cases}
$$

A DDG corresponding to the program in Figure 1(a) is shown in Figure 2(a). Each node is labeled with a cost pair (c_i^L, c_i^R). Remote nodes are denoted by ellipses and local nodes by rectangles. Remote edges are shown as dashed lines and local edges as solid lines.

3 Problem Statement

The thread partitioning problem studied in this paper is to divide a DDG into threads so that



(a). A DDG in which nodes are annotated with local and remote costs

(b). A partition with 4 threads. Execution time=37 and $\delta = 1$

Figure 2: A DDG and a Trivial Partition

1. All data dependences are obeyed;
2. All remote edges become split-phase transactions;
3. Total execution time of the program is minimized.

In order to enforce both the data dependences and split-phase requirements, there must be some kind of ordering among threads and nodes. Data dependences impose a partial order among nodes, so combining these nodes into threads will introduce a partial order among the threads. Since a thread is executed sequentially, the nodes within that thread are naturally in a total order, which must reflect the partial order required by intra-thread data dependences. For non-preemptive threads, remote edges are performed as split-phase transactions; thus, two nodes connected by a remote edge must be placed in different threads.

Formally, a *thread* T_k is a non-empty subset of V with a total order $\prec$ defined over the members of T_k. Let $T(v_i)$ be the thread containing v_i. Let $\mathcal{P}$ be a set of threads $T_1 \ldots T_k$ with a partial order $\sqsubset$ defined over the threads. We say that $\mathcal{P}$ is a *correct partition* if

1. $\mathcal{P}$ is a partition of V;
2. $C_{i,j} = 1 \Longrightarrow (T(v_i) = T(v_j) \wedge v_i \prec v_j) \vee (T(v_i) \sqsubset T(v_j))$;

3. $C_{i,j} = 1 \wedge c_i^R > 0 \Longrightarrow T(v_i) \sqsubset T(v_j)$

The second condition ensures data dependences, and the third condition enforces the split-phase requirement. For example, if there is a remote edge from v_i to v_j, then v_i and v_j must be in separate threads, and v_i's thread must execute first. If the edge from v_i to v_j is local, then these nodes can be in the same thread, but v_i must execute before v_j. If they are in different threads, then v_i's thread must execute first.

Figure 2(b) shows a partition of the DDG with four threads. The members of each thread are listed in their execution order (for readability, only the node numbers are listed). Thread T_1, for example, has nodes $v_1 \ldots v_4$ with $v_1 \prec v_2 \prec v_3 \prec v_4$. The data dependences and remote accesses impose the partial order $T_1 \sqsubset T_2 \sqsubset T_3 \sqsubset T_4$ (which just happens to be a total order). If a multithreaded machine were to schedule threads in the order in which their data becomes available, then each thread would start at the time indicated on the time axis in Figure 2(b), assuming a thread switching cost δ of 1 cycle. The total execution time in this example is 37 $(36 + 1)$ cycles.

Producing a correct partition guarantees that the program executes correctly, but does not necessarily minimize execution time, which is one of our goals. For instance, the partition in Figure 2(b) takes 37 cycles to run, but an optimal partition takes only 26 cycles. We will use this example to discuss the difficulties when seeking the optimal solution in thread partitions.

The first difficulty is in deciding the order of execution within a thread. The difficulty in such scheduling is that the *critical path* of the partitioned threads is unknown until all threads are generated and scheduled. For instance, when deciding the position of v_4 within T_1, other threads have not been generated yet. Thus, there is no way to know exactly which of the four nodes will lie on the critical path of the final schedule. Heuristics may help; for example, since v_4 is a remote node, it may be better to issue it earlier. However, such simple heuristics may not always produce the best results, because it is possible that v_2 could lie on the critical path rather than v_4.

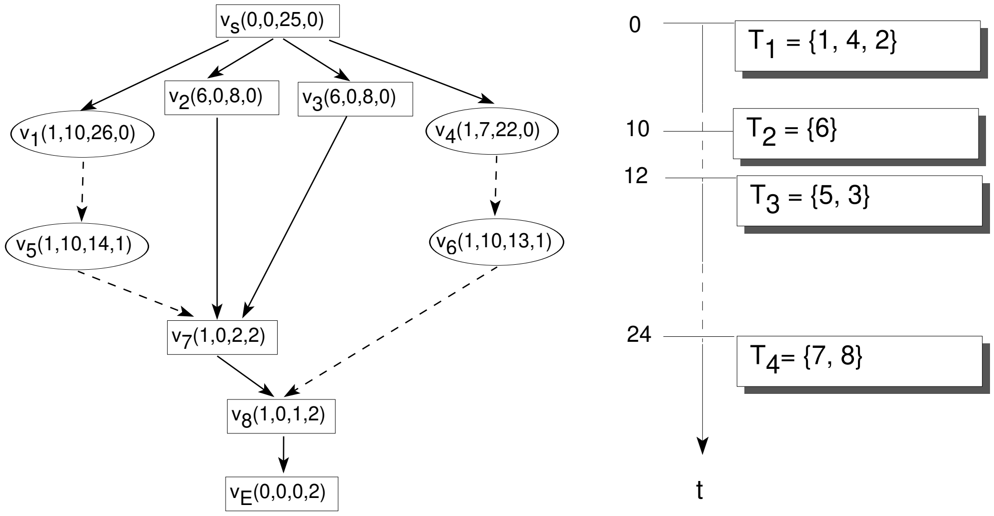
The second difficulty is how to choose between short and long threads. Short threads can run concurrently, thereby exploiting parallelism, but long threads lead to fewer context switches, hence lowering thread-switching costs. When assigning a node to a thread, we can combine the node with others to form a larger thread or create a new thread for this node. As mentioned in Section 1.1, it is better to put two nodes (v_5 and v_6) into separate threads rather than the same thread. On the other hand, two nodes (v_7 and v_8) can be merged into one thread to save thread switching costs. The balance between short and long threads is strongly impacted by the value of δ. For instances, if δ is relatively large then merging nodes into threads is more profitable than creating new threads. On the other hand, if δ is 0 (thread switching is free) then each remote node can belong to a separate thread.

The final challenge is to make the best use of local nodes to cover remote latencies. In Fig. 2(b), assigning two local nodes in T_1 makes their combined local execution time $(6+6 = 12)$ longer than the remote execution time of v_1 (10). In Section 4, we will show that it is profitable to assign v_3 to other threads so that it can be *completely* used to cover other latencies, rather than partially.

In summary, thread-partitioning decisions imply trade-offs between *parallelism*, *synchronization costs*, and *sequential efficiency* [30]. With our cost model, we add another dimension, the placement of local nodes so that they can be fully used to hide latencies.

Traub [33] shows that the thread partitioning problem for lenient languages is NP-complete. To our knowledge, we are the first to prove that thread partitioning for strict languages with minimum execution time as an optimization goal is NP-hard, see appendix for details.

4 List-Scheduling Based Heuristic Algorithm



(a). Each node annotated with $(c_i^L, c_i^R, h_i, l_i^V)$ (b). Optimal partition (time=26)

Figure 3: An Annotated DDG and Its Optimum Partition ($\delta = 1$)

Since the problem we just formulated is NP-hard, feasible algorithms require the use of heuristics. We will first give a brief introduction and intuition about the heuristics used in thread partitioning, followed by a presentation of the list-scheduling based algorithm and an analysis of its complexity. Finally, a tracing example of the algorithm execution is listed to demonstrate how the heuristic rules work cooperatively to derive an optimal partition.

4.1 Overview of Thread Partitioning Heuristics

A heuristic algorithm works by making "educated guesses" when searching the solution space. To generate a partition with minimal execution time, predicting which nodes lie on the critical path of the DDG is key to the success of a heuristic partitioning algorithm. To help determine which nodes are critical, two numbers are computed for each node: the *height* h_i, and the *remote level* l_i^V. To compute these numbers easily, the DDG is augmented with two additional nodes, the *start* and *end* nodes (v_s and v_e). Edges are added between v_s and each node without predecessors, and between each node without successors and v_e. (Figure 3(a) shows the DDG of Figure 2(a) augmented in this way.) The height and remote level are then defined as follows:

$$\left(\begin{array}{lcl} h_e & = & 0 \\ l_s^V & = & 0 \\ h_i & = & c_i^L + \left\{ \begin{array}{ll} \max_{j:C_{i,j}=1} h_j & \text{if } v_i \text{ local} \\ \max_{j:C_{i,j}=1} h_j + c_i^R + \delta & \text{if } v_i \text{ remote} \end{array} \right. \\ l_i^V & = & \max_{j:C_{j,i}=1} \left(l_j^V + \left\{ \begin{array}{ll} 0 & \text{if } v_j \text{ local} \\ 1 & \text{if } v_j \text{ remote} \end{array} \right. \right) \end{array} \right.$$

The height of a node is the length of the longest path from that node to the end, including that node's local and remote costs. If the node is remote, then δ is added to the height, because there will have to be at least one thread switch before executing any of the dependents (by the split-phase requirement). Thus, h_i gives a lower bound on the number of cycles required to finish executing the DDG when v_i starts executing. The *remote level* l_i^V of node v_i is the maximum of the number of *remote* edges along any path from v_s to v_i. Therefore, it gives a lower bound on the number of thread switches that must occur before v_i begins execution. These numbers can be computed using graph traversal in $O(|V| + |E|)$ time. The augmented DDG in Figure 3(a) is annotated with the quadruple $(c_i^L, c_i^R, h_i, l_i^V)$.

Intuitively, the height helps to decide which node may go earlier, and the remote level ensures that split-phase transactions are handled properly. For instance, the partition shown in Figure 2(b) was derived by grouping all nodes with the same distance from v_s (*not* the same as the remote level) into one thread. However, since h_4 is greater than both h_2 and h_3, we may guess that v_4 is more likely to be on the critical path of the final generated threads, so we should put v_4 ahead of v_2 and v_3. The resulting partition is shown in Figure 3(b) (T_1). On the other hand, because l_7^V and l_8^V are the same, it is safe to put v_7 and v_8 in the same thread, saving one switch cost as shown in Figure 3(b) (T_4).

Unfortunately, height and remote level alone are not enough, because they do not show us that reassigning v_3 to another thread, and splitting v_5 and v_6 will produce an optimal partition, as shown in Figure 3(b). We need to introduce the *starting* and *finishing* times of threads, and use additional heuristics to deal with situations such as when the *finishing* time of T_1 is too late or the *starting* time of T_2 in Figure 2(b) is delayed by more than δ when v_5 and v_6 are merged into one thread. These are covered in the next section.

4.2 List Scheduling Based Algorithm

The list scheduling partitioning algorithm takes as input a DDG, annotated with h_i and l_i^V costs as described in the previous section. It works by moving nodes one by one from F, a *free list* of unassigned nodes sorted by height in descending order, to $\mathcal{P}$, which represents the current state of the partition. Initially, $\mathcal{P}$ is empty and F has one node, v_s. A node enters into F after all its predecessors have been placed in F. The following definitions are used:

- The *starting time* s_i^V of node v_i is the time when the node starts execution in the final scheduled partition.
- The *starting time* s_i^T of thread T_i is the time when the thread starts its execution in the final partition.
- The *finishing time* f_i^V of node v_i is the time at which all of the node's operations (local and remote) have finished:

$$f_i^V = s_i^V + c_i^L + c_i^R + \begin{cases} 0 & c_i^R = 0 \\ \delta & c_i^R > 0 \end{cases}$$

- The *finishing time* f_i^T of thread T_i is the time when all *local* operations on the thread have finished: $f_i^T = s_i^T + \sum_{j:v_j \in T_i} c_j^L$
- The *ready time* r_i^V of node v_i is the time at which all required data is ready, which is the maximum finishing time of all predecessor nodes: $r_i^V = \max_{j:C_{j,i}=1} f_j^V$
- The *remote level* l_i^T of thread T_i is the maximum of the remote levels of its constituent nodes: $l_i^T = \max_{j:v_j \in T_i} l_j^V$

The actual starting times of nodes and threads rely on a runtime thread scheduler. We assume a FIFO scheduling policy, in which threads are scheduled to run in the order in which they become ready. Because a thread can't begin running until all data it requires is ready, the "ready time" of thread T_i is the maximum of the ready times of nodes in T_i with remote predecessors. For example, nodes v_5 and v_6 are separated from their predecessors v_1 and v_4; their *ready* times are computed as follows (see Figure 2(b)):

$$\begin{array}{lllll} r_5^V = & f_1^V = & s_1^V + c_1^L + c_1^R + \delta & = & 0 + 1 + 10 + 1 \;\; = 12 \\ r_6^V = & f_4^V = & s_4^V + c_4^L + c_4^R + \delta & = & 1 + 1 + 7 + 1 \;\; = 10 \end{array}$$

Pseudocode for the main thread partition program is shown in Figure 4(a). Each iteration schedules one node. In each pass, $ThreadPartition$ takes the first element v_i off of F (the unassigned node with the greatest height) and computes v_i's ready time r_i^V. Then, procedure $AssignThread$ assigns v_i to a thread, either assigning it to an existing thread in $\mathcal{P}$ or creating a new thread for that node. The search will first use the remote level to guarantee that data dependence constraints are satisfied, and then compare the ready time r_i^V with the starting times of other threads to minimize the total execution time. Once a thread T_k has been assigned, procedure $InsertIntoThread$ determines the position of v_i within T_k. At the end of each iteration, the F list is updated according to the topological sorting order of nodes as shown in Figure 4(b), and iteration continues until F is empty.

All quantities except the remote levels of threads may vary during execution of the algorithm, and thus need to be recomputed dynamically. The starting times of threads in $\mathcal{P}$ are organized in non-decreasing order. The algorithm maintains the relative order of threads already in $\mathcal{P}$,

Input : a DDG and v_i with h_i and l_i^V.
Output: $\mathcal{P}$ sorted by thread starting time.
PROC ThreadPartition()
  $\mathcal{P} := \emptyset$; $F := \{v_s\}$
  WHILE ($F \neq \emptyset$) DO
    v_i := first node in F
    r_i^V := ComputeReadyTime(v_i,$\mathcal{P}$)
    k := AssignThread($\mathcal{P}$,v_i,l_i^V,r_i^V)
    InsertIntoThread(v_i, T_k)
    Update(F,v_i)
END.

(a) Main Program

PROC Update(F, v_i)
  FOR each successor v_j of v_i DO
    IF all predecessors of v_j have been
      in either F or $\mathcal{P}$
    THEN add v_j into F
END;

(b) Update Free List

PROC AssignThread($\mathcal{P}$,v_i,l_i^V,r_i^V)
  IF $\forall l_j^T < l_i^V (T_j \in \mathcal{P})$ THEN
    k:=l_i^V /* Rule 1 is succeeded */
  ELSE
    $k := min(k)$ such that $l_k^T = l_i^V$
    REPEAT
      test v_i, r_i^V against T_k
      IF match with Rule 2-4 THEN
          break;
      ELSE $k := k + 1$
    UNTIL ($k > |\mathcal{P}|$)
  apply Rule 5
END;

(c) Thread Formation

PROC InsertIntoThread(v_i, T_k)
  j = number of last node in T_k
  REPEAT
    IF $C_{j,i} = 1 \vee$ ($\max(h_i, c_i^L + h_j) \geq$
      $\max(h_j, c_j^L + h_i)$)
    THEN
      insert v_i after v_j; RETURN
    ELSE
      go to previous node
  UNTIL v_j = head of list
  insert v_i at head of list
END;

(d) Placement within a thread

Figure 4: Partition Program Based on List Scheduling

although new threads may be inserted in the middle, causing the starting times of later threads to change. The algorithm also guarantees that the remote levels of threads are monotonically increasing.

Procedures $AssignThread$ and $InsertIntoThread$ are listed in Figure 4(c) and (d) and are discussed in the next two subsections.

4.2.1 Thread Formation

Thread formation is a process to decide which thread a node will join. A node can be merged into an existing thread or a new thread can be created for that node. Merging nodes reduces the number of threads generated and thus reduces the total costs due to thread switching. On the other hand, creating a new thread may reduce thread waiting times.

The main loop in procedure $AssignThread$ is listed in Figure 4(c). To assign v_i, procedure $AssignThread$ searches $\mathcal{P}$ sequentially, starting at the first thread whose remote level equals l_i^V, using the heuristic rules given below. They are used to 1) ensure the data dependences and split-phase requirements are met, and 2) balance small concurrent threads and large sequential threads. Rules 1-4 are tested one by one in order. If Rule 1 or 4 is satisfied, a new thread is created. If Rule 2 or 3 is satisfied, then the node is added to the first existing thread that satisfies Rule 5.

Rule 1: If l_i is greater than the remote level of any of the threads in $\mathcal{P}$ then add a new thread $T_{\mathcal{P}+1} = \{v_i\}$ to $\mathcal{P}$.

Rule 2: If the node with the maximum finishing time of v_i's predecessors is a local node, then put v_i in the same thread as that predecessor.

Rule 3: If v_i is ready within δ cycles of the start of T_k (i.e., $|r_i^V - s_k^T| <= \delta$), then merge v_i into thread T_k. It is better to merge it into an existing thread than to create a new thread if the penalty for waiting is less than the thread switching time.

Rule 4: If v_i can go in between T_{k-1} and T_k, then create a new thread there. This is allowed if 1) $r_i^V + c_i^L <= s_k^T - \delta$ and 2) $r_i^V > f_{k-1}^T$. The first condition says v_i can start earlier than T_k by at least δ, and the second condition says the previous thread does not prevent v_i from starting. This rule tries to find some idle interval between T_{k-1} and T_k to exploit the parallelism between the newly created thread and T_k.

Rule 5: If a previous rule decides that v_i should join T_k, this is only allowed if $(f_k^T + L_i < s_{k+1}^T)$. In other words, only add v_i to T_k if this doesn't delay the start of thread T_{k+1}. If this condition fails, search successive threads until the condition is satisfied.

The first two rules guarantee that the data dependence constraints are satisfied, and the next two rules try to balance small concurrent threads with big sequential threads. Finally, Rule 5 addresses the latency tolerance problem, trying to make full use of local nodes to cover remote latencies.

For instance, Rule 3 tries to decide when merging a node into an existing thread is beneficial. Suppose v_6 ($h_6 = 13$) is being scheduled, and v_5 ($h_5 = 14$) has already been scheduled. As calculated in the previous section, $r_5^V = 12$ and $r_6^V = 10$. Since $r_5^V - r_6^V$ is 2, which is greater than δ, Rule 3 would not add it to $T(v_5)$. Instead, Rule 4 would apply, and a new thread would be squeezed in the space before $T(v_5)$, allowing v_6 to start earlier, as shown in Figure 3(b).

Rule 5 prevents threads from becoming too long and delaying the threads that follow. For instance, if the program tries to place v_3 in the same thread as v_1, v_4 and v_2, the finishing time would become greater than the starting time of the next thread:

$$f_1^T = s_1^T + c_1^L + c_4^L + c_2^L + c_3^L = 0 + 1 + 1 + 6 + 6 = 14 > 12$$

This means some local computation is "wasted" in terms of latency tolerance. Rule 5 reduces total execution by forcing v_3 into another thread, as shown in Figure 3(b).

4.2.2 Sequencing within Threads

If a node v_i is assigned to an existing thread, then a place for v_i within that thread must be found. Testing all possible orders for optimality would require exponential time, so our heuristic approach simply inserts v_i at one place in the sequence while preserving the relative order of the nodes already in the sequence. Procedure *InsertIntoThread*, shown in Figure 4(d), examines the existing nodes from the end to the beginning of the list. It stops and inserts v_i *after* an existing node v_j if 1) v_i is data dependent on v_j, or 2) placing v_i in front of v_j would increase the total execution time of the program. If *InsertIntoThread* reaches the head of the list, then it puts v_i at the head of the list.

To estimate whether or not condition 2 above is satisfied, *InsertIntoThread* examines the costs of the two alternate orders $v_i \prec v_j$ and $v_j \prec v_i$. By definition, h_i is a lower bound on the number of cycles from the beginning of v_i to the *end* node. If v_j comes before v_i, then v_j delays the start of v_i by c_j^L cycles, so the time from the beginning of the pair to the *end* node is at least $c_j^L + h_i$ cycles. However, v_j itself could be on the critical path, so the time from the beginning of the pair to the end of the program is bounded by $\max(h_j, c_j^L + h_i)$. Similarly, if v_i comes first, then the maximum time is at least $\max(h_i, c_i^L + h_j)$. *InsertIntoThread* uses the smaller of these two values to decide which should go first.

4.2.3 Complexity of the Algorithm

This algorithm has $O(|V|^2)$ time complexity. In the *main* loop shown in Figure 4(a), each node is selected once. Each pass through the loop calls *AssignThread* and *InsertIntoThread*. As shown in Figure 4(c), *AssignThread* may search all existing threads. In the worst case, when each thread contains only one node, it will take $O(|V|)$ time to finish. For procedure *InsertIntoThread*, the worst case is when only one thread is formed. This case also needs $O(|V|)$ time to find a proper position for a node. Thus, the whole loop spends $O(|V|^2)$ time in the two procedures in the worst case. Furthermore, when calculating the ready time, the node's incoming edges connecting it to its predecessors are searched one by one. Thus, the complexity is $O(d)$ where d is the in-degree of the node. For the whole loop, there are $|E|$ edges to be searched in total, so the total time complexity is $O(|V|^2 + |E|)$. Since $|E|$ is bounded by $|V|^2$, this reduces to $O(|V|^2)$.

4.2.4 Tracing Algorithm Execution

We will illustrate how the heuristic algorithm derives the optimal partition for the DDG shown in Figure 3(a). Table 1 lists the algorithm execution trace. Column F displays the *free list* in each selection cycle, column v_i lists the current searched node, column h_i, r_i^V and l_i^V give the height, the ready time, and the remote level of the currently searched node respectively, column $\mathcal{P}$ shows all existing threads so far after the current node is processed, and the last column indicates which heuristic rules are used in forming a thread.

Initially, the *start* node v_s is chosen and it forms the first thread. After v_s is processed, F contains four nodes: v_1, v_2, v_3, and v_4. Node v_1 is selected since it has the highest height. Using rule two, v_1 is merged into thread one because v_s is a local predecessor of v_1. Similarly, v_4 is chosen next and merged into thread one.

At iteration 3, v_5 is chosen and rule one is used to create a new thread since $l_5^V = 1$, which is the greatest so far. Next, v_6 is selected and rule four is applied to create a new thread to exploit parallelism, because $r_6^V + c_6^l <= T_2 - \delta$. Threads in $\mathcal{P}$ are sorted according to their starting times. The sorting result is shown under the $\mathcal{P}$ column in table 1.

Now, F has two remaining nodes: v_2 and v_3. Node v_2 is chosen arbitrarily. It is decided to enter into thread one by rule two. Then rule five is applied and the checking is successfully passed. Next, v_3 is chosen and planned to enter thread one. However it can not be inserted into thread one because $f_1^T = 14 > 12 = s_2^T$ if it were inserted. For the same reason, v_3 can not be put in thread two. But it can settle in thread three since $f_3^T < \infty = s_4^T$ (the starting time of unformed threads is preset to infinity).

We proceed with scheduling v_7 and thread four is created since h_7 is higher than that of any scheduled node. Finally, v_8 and v_e are merged into the thread four by using rule two. The final partition is shown at iteration 9 in table 1.

Iteration	F	v_i	h_i	r_i^V	l_i^V	$\mathcal{P}$	Rules Used
0	{s}	s	26	0	0	$T_1 = \{s\}$	1
1	{1,4,2,3}	1	26	0	0	$T_1 = \{s, 1\}$	2,5
2	{4,5,2,3}	4	22	0	0	$T_1 = \{s, 1, 4\}$	2,5
3	{5,6,2,3}	5	14	12	1	$T_1 = \{s, 1, 4\}$, $T_2 = \{5\}$	1
4	{6,2,3}	6	13	10	1	$T_1 = \{s, 1, 4\}$, $T_2 = \{6\}$ $T_3 = \{5\}$	4
5	{2,3}	2	8	0	0	$T_1 = \{s, 1, 4, 2\}$, $T_2 = \{6\}$ $T_3 = \{5\}$	2,5
6	{3}	3	8	0	0	$T_1 = \{s, 1, 4, 2\}$, $T_2 = \{6\}$ $T_3 = \{5, 3\}$	2,5
7	{7}	7	2	24	2	$T_1 = \{s, 1, 4, 2\}$, $T_2 = \{6\}$ $T_3 = \{5, 3\}$, $T_4 = \{7\}$	1
8	{8}	8	1	25	2	$T_1 = \{s, 1, 4, 2\}$, $T_2 = \{6\}$ $T_3 = \{5, 3\}$, $T_4 = \{7, 8\}$	2,5
9	{e}	e	0	26	2	$T_1 = \{s, 1, 4, 2\}$, $T_2 = \{6\}$ $T_3 = \{5, 3\}$, $T_4 = \{7, 8, e\}$	2,5

Table 1: Execution Trace of the Heuristic Algorithm

5 Experimental Results

To test the efficacy of heuristic algorithms, a Thread Partition Test Bed (TPTB) was built to experiment with various algorithms for thread partitioning. TPTB consists of three main components: 1) a visual tool to show DDGs graphically, 2) a random graph generator to produce required DDGs, and 3) a partition engine to do thread partitioning. The random graph generator can produce different DDGs according to the architecture and program specifications. The partition engine currently implements three heuristic algorithms for comparison purposes.

Running these algorithms on a large number of randomly-generated graphs whose characteristics resemble actual benchmark programs, we found:

- The new algorithm we proposed produces partitions whose run time is within 7% of the ideal run time (as determined by the critical path in the DDG).
- It compares favorably with the other two heuristic algorithms.
- It is particularly effective for fine-grain parallel programs.
- It benefits from the various heuristics used. Each rule plays an important role in ensuring partition efficiency.
- The experimental results are valid for various architecture and program characteristics.

5.1 Algorithms Compared

The new algorithm we developed uses a complex set of heuristics in an attempt to optimize thread partitioning. To measure how important it is to use such a complicated method, we implemented two simpler strategies.

The first is a simplified version of our new algorithm. In the simplified version, procedure $AssignThread$ considers only three threads: the previous, current, and next. This reduces the time complexity of that procedure from $O(|V|)$ to $O(1)$. Procedure $InsertIntoThread$ searches only the last d nodes of the thread, where d is the maximum out-degree of the DDG. This cuts the complexity from $O(|V|)$ to $O(d)$. Thus, the total complexity of this simplified algorithm is $O(d|V|)$.

The simplifications are based on two assumptions. First, it is highly likely that a node is assigned to one of the three threads surrounding the current thread, which is decided by the remote level of the node. Second, it is not likely that a thread has more than d nodes since there exist at most d parallel nodes from the same parent.

The second algorithm comes from a real compiler [15]. It simplifies procedure $AssignThread$ by assigning all nodes at one remote level to the same thread, reducing its complexity to $O(1)$. Within a thread, Procedure $InsertIntoThread$ schedules remote nodes as early as possible, so long as data dependences are obeyed. Thus, it has the same complexity $O(|V|)$ as before. Even though the total complexity is still $O(|V|^2)$, this algorithm normally runs faster than the new one due to the simplifications introduced.

To see how close to optimal the new algorithm performs, we compare its results against an estimated lower bound for the running time of an optimal partition. Because the actual running time of threads depends on a runtime scheduler which dynamically schedules ready threads, we assume threads are scheduled to run in the order that they become ready (FIFO scheduling). Thus, the execution time of partitioned threads is the starting time of the *end* node, s_e^V. The lower bound of the running time of the *optimal* partition is defined as:

$$T_{optimal} = \max(h_s, \sum_{i \in V} c_i^L) \quad (1)$$

It is the larger of h_s (the height of the *start* node) and the sum of all local costs. Furthermore, $T_{optimal}$ is the greatest lower bound since the total running time cannot be less than either of these. The running time of the *optimal* partition may in fact be longer than $T_{optimal}$, but this is impossible to verify for larger graphs because the problem is NP-hard.

d	Node Distribution(%)				
	Local			Remote	
	F-call	LM	ALU	RM	RF-call
3	10	20	70	80	20

(a) Program Parameters

EARTH/MANNA				Future MTA Machine			
Cycle=20ns		$\delta = 36$		Cycle=2ns		$\delta = 1$	
T_{lm}	T_{rm}	T_{le}	T_{ri}	T_{lm}	T_{rm}	T_{le}	T_{ri}
10	300	3	40	50	500	3	1

(b) Architectural Parameters

Table 2: Architectural and Program Parameters Used

5.2 Experimental Framework

In order to characterize multithreaded machines, we use five parameters: 1) the local-memory access time (T_{lm}), 2) the remote-memory access time (T_{rm}), 3) the thread switching cost (δ), 4) the local execution time of ALU instructions (T_{le}), and 5) the local time to issue remote operations (T_{ri}).

Table 2(b) lists the parameters for two representative multithreaded machines. The first set is for EARTH-MANNA, an emulation of a multithreaded machine running on a real parallel machine [17]. Values for δ (36 cycles) and T_{ri} (40 cycles) are relatively high, because while EARTH-MANNA has hardware support for multithreading, the hardware is not optimized for this function, and so there are large overheads in performing multithreaded operations. The second set assumes a state-of-the-art processor with optimal hardware support, so we assume both thread switching and invoking remote operations are quick (1 cycle). We further assume that the network, while faster than EARTH-MANNA, has not kept pace with the CPU, so the remote latency problem is exacerbated. Finally, exact execution time for ALU instructions is difficult to predicate since it depends on the distribution of ALU instructions. We assume that on average each ALU instruction takes 3 cycles to finish.

In order to characterize a parallel program, we use these parameters: 1) the size (number of nodes) of a DDG ($n = |V|$), 2) its average degree (d), 3) the node types, and 4) the node and cost distributions.

We built a random graph generator to produce graphs according to these parameters. The average degree d is based on static profiling of the EARTH-C benchmarks [15]. Other parameters shown in Table 2(a) are related to node (statement) types. A local node can be a *Local Memory* access, an *ALU* op, or a *Function Call*. A remote node can be either a *Remote Memory* access or a *Remote Function Call*. Function-call and loop nodes are treated the same in this program model since these nodes take more time than memory-access and ALU nodes.

Machine	Remote	Nodes					Ave.
	(%)	10	50	100	150	200	(%)
EARTH	0	100.0	100.0	100.0	100.0	100.0	100.0
	25	99.87	100.0	99.24	99.21	71.00	93.86
	50	100.0	99.97	98.57	99.46	87.73	97.07
	75	100.0	100.0	99.19	100.0	99.74	99.74
	100	100.0	100.0	100.0	99.69	100.0	99.96
Average		99.97	99.99	99.40	99.65	91.58	**98.12**
Future Machine	0	100.0	100.0	100.0	100.0	100.0	100.0
	25	83.48	99.99	96.48	99.95	82.73	92.53
	50	97.09	99.98	100.0	99.95	92.95	97.99
	75	91.42	100.0	99.98	100.0	97.78	97.88
	100	100.0	100.0	100.0	99.99	100.0	100.0
Average		94.40	99.99	99.29	99.98	94.73	**97.68**

Table 3: E measured under $R_{lrf} = 1:4$

We further assume that 70% of local nodes are ALU ops (use registers only), and the rest are Local Memory and Function Call types. For remote nodes, we assume that the majority of them are about to access remote memory, setting RM to 80%.

In the experiment, three parameters are varied: 1) n ranges from 10 to 200 vertices; 2) the percentages of remote nodes are set to 0%, 25%, 50%, 75%, and 100% of total nodes; and 3) The ratioes between the upper bounds of local functions and those of remote functions (R_{lrf}) are set to $1:4$ and $1:2$.

5.3 Absolute Algorithm Efficiency

The *absolute efficiency* of the new algorithm is defined as $E = T_{optimal}/T_{new}$. In this ratio, $T_{optimal}$ is defined in Equation 1 and T_{new} is s_e^V. Table 3 shows E, for two machine models. Each number is the estimated optimal running time divided by the running time of the partition generated by our algorithm. It also represents the speed of the partition generated by the new algorithm as a fraction of the "ideal" speed. A speed of 100% indicates that the new algorithm has found an optimal partition. A speed of less than 100% means the new algorithm did not reach this ideal limit, but it may still have found an optimal partition because the estimate for the running time is only a lower bound.

The changing parameters are the node count and the percentage of remote nodes. Because the average performance is 98.12% for EARTH-MANNA and 97.68% for the future machine, it demonstrates that the new algorithm works well. Furthermore, the table shows that the algorithm works extremely well when the majority of nodes are remote. For example in the EARTH model, E is increased from 71% to 99.74% when n is 200 and the remote node ratio goes up from 25% to 75%. Since a fine-grained program usually employs many remote operations, this result indicates that the new algorithm is highly suitable to fine-grained multithreading

Machine	Remote (%)	Nodes 10	50	100	150	200	Ave.
EARTH	0	100.0	100.0	98.34	100.0	100.0	99.67
	25	69.42	100.0	80.62	76.86	83.08	82.00
	50	89.40	99.97	80.05	90.96	91.07	90.28
	75	93.76	100.0	98.40	99.78	91.43	96.67
	100	100.0	100.0	100.0	99.78	99.80	99.92
Average		90.53	99.99	91.48	93.46	93.08	**93.71**
Future Machine	0	100.0	100.0	100.0	100.0	100.0	100.0
	25	68.02	99.99	78.21	88.12	88.27	84.50
	50	88.76	99.98	78.85	93.11	77.47	87.63
	75	92.88	100.0	99.80	100.0	92.93	97.16
	100	100.0	100.0	99.98	99.99	100.0	99.99
Average		89.93	99.98	91.40	96.24	91.73	**93.86**

Table 4: E measured under $R_{lrf} = 1 : 2$

programs.

Table 4 lists E for the increased upper bound of local function costs ($R_{lrf} = 1 : 2$). This setting doubles the local-cost portion of a DDG, reducing the possibility of the overlapping execution of local and remote nodes. Not surprisingly, E drops compared to the numbers listed in Table 3. However, the general trend still holds. On average, performance of the new algorithm is within 7% of that of the ideal one. Moreover, when the remote node percentage rises to 75%, E goes back to 96.67% and 97.16% for the EARTH model and future machine, respectively.

Even though $T_{optimal}$ is not an ideal measurement, the experimental results show that the new algorithm works very well under a variety of situations.

5.4 Algorithm Relative Efficiency

In order to compare performance between the new algorithm and the two simpler algorithms, we define the *algorithm relative efficiency* $E' = T_{new}/T_{other}$. T_{new} and T_{other} are the starting times of the *end* node produced by the corresponding algorithms. We expect to see $E' <= 1$. Thus, a small E' means a big performance difference between two algorithms.

Table 5 compares the new algorithm to the other two algorithms, using E'. In this table, the number also represents the speed of the partition generated by the alternate strategy as a fraction of the speed of the partition produced by the new algorithm. Due to the space limitation, only the results for EARTH-MANNA are shown. Because all ratios in Table 5 are not greater than 1, we conclude that the new algorithm beats both the simplified and remote-level based algorithms. In particular, when the size of a DDG increases, the ratio becomes lower, indicating that the new algorithm is a clear winner. On the other hand, for

Algorithm	Remote (%)	Node				
		10	50	100	150	200
Simplified	0	100.0	100.0	100.0	100.0	100.0
	25	100.0	96.47	80.91	84.77	80.40
	50	100.0	100.0	88.36	68.30	85.32
	75	100.0	89.58	65.76	88.58	64.56
	100	100.0	100.0	73.56	80.61	88.61
Average		100.0	97.21	81.72	84.45	83.80
Remote level based	0	100.0	100.0	100.0	100.0	100.0
	25	100.0	91.01	76.61	82.81	78.02
	50	99.97	96.34	80.55	68.87	65.01
	75	100.0	97.02	51.82	83.39	57.56
	100	100.0	100.0	71.29	74.71	75.65
Average		99.97	96.87	76.05	81.96	75.25

Table 5: E' for the EARTH model and $R_{lrf} = 1:4$

smaller DDGs ($n <= 50$), the performance difference on average is within 4%, suggesting that simplified algorithms could be used to reduce the implementation complexity and algorithm execution time.

5.5 Thread Length

Machine	Remote (%)	Nodes					Ave.
		10	50	100	150	200	
	0	12.0	52.0	102.0	152.0	202.0	
EARTH	25	4.00	5.78	8.50	8.00	10.63	7.38
	50	2.00	2.89	4.25	5.43	6.52	4.22
	75	1.71	3.06	3.52	4.43	4.21	3.39
	100	1.50	2.36	3.62	3.04	3.96	2.70
Average		2.30	3.52	4.72	5.23	6.33	**4.42**
Future Machine	0	12.0	52.0	102.0	152.0	202.0	
	25	4.00	5.78	6.00	10.86	10.10	7.35
	50	2.00	2.89	5.37	5.24	5.32	4.36
	75	1.71	3.06	3.52	3.04	3.31	2.84
	100	1.50	2.36	2.43	2.71	2.81	2.31
Average		2.35	4.04	5.18	5.98	6.23	**4.16**

Table 6: Average thread length for $R_{lrf} = 1:4$

The thread length is measured by the average number of vertices per thread. Table 6 gives the average thread length under $R_{lrf} = 1:4$. In order not to distort the whole picture, the case of 0% remote nodes (100% local nodes) is excluded from the average, because there is only one thread produced in this case.

On average, there are 4.42 nodes per thread for EARTH-MANNA and 4.16 for the future machine. Both numbers are greater than 3, which is the average degree of the DDG. Larger threads for EARTH-MANNA follow from the fact that merging threads is profitable when the thread switching cost is high.

Furthermore, we observe that the more remote nodes there are, the smaller threads become. On average, the thread length decreases from 7.38 when there are 25% remote nodes to 2.70 when the remote fraction is 100%. The reason is that the split-phase requirement forces the source and destination nodes linked by a remote edge into different threads.

Finally, as the size of the DDG increases, so does the thread length. When the remote ratio is 50% in the EARTH model, the thread length more than triples, from 2.00 to 6.52, as the size increases from 10 to 200 nodes. One of the reasons is that big DDGs reveal more parallelism for Rule 3 to merge vertices into threads.

5.6 Effectiveness of Heuristic Rules

Table 7 shows how the usage frequency of the five heuristic rules on the EARTH model. We observed that:

Cases	Nodes	Average				
		Rule 1	Rule 2	Rule 3	Rule 4	Rule 5
Overall	10	40.51	46.67	8.08	1.67	3.08
	50	19.64	54.28	12.91	11.04	2.13
	100	11.17	50.81	9.16	12.33	16.53
	150	8.65	50.87	9.38	13.88	17.22
	200	6.86	46.42	8.02	15.26	23.33
Average		17.37	49.81	9.51	10.84	12.46
50% of remote nodes	10	46.15	38.46	7.69	0.00	7.69
	50	26.92	44.23	15.38	13.46	0.00
	100	14.18	39.01	9.22	9.33	27.66
	150	8.54	44.22	6.53	17.09	23.62
	200	9.22	37.94	8.87	15.06	28.37
Average		21.00	40.77	9.54	10.99	17.47

Table 7: Rule-usage distributions

First, Rule 2, which merges a node into the same thread with one of its *local* predecessors, applies nearly 50% of time overall. It suggests that *local* data dependence is a dominant factor in thread partitioning. In addition, the sum of the usage frequencies of the first two rules makes up more than two-thirds of all rule invocations. This demonstrates that more than two-thirds of the total thread-formation time is spent satisfying the data dependence constraints.

Second, Rule 3 is used to throttle the excess parallelism to save switching cost and Rule 4 to create a new thread to exploit the parallelism. The usage of these two rules is almost balanced, 9.51% against 10.84% overall and 9.54% vs. 10.99% for 50% remote nodes. This indicates that the new algorithm does a good job on addressing the second difficult problem raised in Section 3.

Third, Rule 5, which tries to make full use of local nodes to cover remote latencies, is used more often than either of Rule 3 or 4. For the overall case, its usage is increased more than 10 times, from 2.13% for 50 nodes to 23.33% for 200 nodes. However, we found one case (50% of nodes are remote and node count is 50) where this rule is never used. This indicates that the use of this rule is heavily dependent on the structure of the DDG.

6 Related Work

Compared to task partitioning and scheduling for multicomputers [2, 14, 27, 7, 11, 12], there is scant research on thread partitioning. One similar problem is task sequencing [35], which derives a total execution order of the tasks allocated to one processor. Another similar problem is instruction reordering [28], which determines the *fork* and *join* order of tasks. Both applications are based on a single-thread execution model, and using multiple threads to hide remote latencies is not their concern.

In the past, research on thread partitioning for multithreaded execution models has been mainly done for functional languages within the data-flow community. Two types of thread partitioning problems have been studied, one targeted for non-strict languages such as Id, and the other for strict languages such as Sisal. Thread partitioning for non-strict languages has more constraints than for strict languages, such as avoiding deadlock. Many algorithms for thread partitioning have been proposed for functional languages. The dependence set method merges threads with the same inputs [19], while the demand set algorithm combines threads with the same output [16, 30]. The combination of both, together with global analysis, is reported in [34, 8]. Moreover, the larger size threads can be generated by applying separation constraint partitioning plus inter-procedural analysis [29]. For a review of those algorithms, readers are referred to [31]. In compiling for strict languages, top-down and bottom-up approaches are proposed in [6, 24] and summarized in [25].

The previous work most relevant to this paper is the *remote level* partitioning algorithm [15]. This algorithm is implemented on the EARTH-C compiler, which generates the threaded code from parallel EARTH-C programs. It is the goal for both the remote level algorithm and those for functional languages to generate threads as large as possible to reduce the thread switching costs [20]. Their assumption is that a thread is not going to be large due to inherent constraints such as split-phase requirements. We assume a cost model for the architecture and the program. The more accurate information provided by this model enables us to explore more heuristics to minimize thread execution time.

7 Conclusions and Future Work

The thread partitioning problem for non-preemptive threads and strict programming languages has been studied in this paper. A cost model to characterize multithreaded architectures and programs was built to enable us to formulate the thread execution time as an optimization goal. Based on this formulation, an efficient heuristic algorithm was designed, and implemented in our TPTB. We experimented on a variety of architecture and program parameters that affect thread partitioning. The experimental results show that our algorithm is very robust and highly efficient under many different configurations.

For the time being, TPTB can only accept DDGs as input. It will be a challenge to apply the new heuristic algorithm to a real compiler. Such a compiler would first build hierarchical DDGs from user programs, estimate cost information using a profiling tool, apply our new partition algorithm to generate threads based on our cost model, and finally produce the threaded code based on the resulting partition. Our research continues along this direction. Our goal is to build an automatic tool which can partition user parallel programs and generate executable code for multithreaded machines.

Acknowledgments

We would like to thank NSERC (Natural Sciences and Engineering Research Council) and Micronet (both in Canada), and NSF-MIPS (NSF-MIPS-9707125 for funding our research. Many people have given assistance, especially Olivier Maquelin, who provided data to model the EARTH-MANNA system, and Sean C. Mcguire, who coded part of the random graph generator. We would like to thank all of ACAPS members who have contributed to this paper.

References

[1] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Summer Usenix Conference Proceedings*, pages 93–112, 1986.

[2] Thomas L. Adam, K. M. Chandy, and J. R. Dickson. A comparison of list schedules for parallel processing systems. *Communications of the ACM*, 17(12):685–690, December 1974.

[3] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An evolutionary processor design for multiprocessors. *IEEE Micro*, 13(3):48–61, June 1993.

[4] Robert Alverson, David Callahan, Daniel Cummings, Brian Koblenz, Allan Porterfield, and Burton Smith. The Tera computer system. In *Conference Proceedings, 1990 International*

Conference on Supercomputing, pages 1–6, Amsterdam, The Netherlands, June 11–15, 1990. ACM. *Computer Architecture News*, 18(3), September 1990.

[5] Boon Seong Ang, Arvind, and Derek Chiou. StarT the Next Generation: Integrating global caches and dataflow architecture. CSG Memo 354, Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, Massachusetts, August 1994.

[6] A. P. W. Böhm, W. A. Najjar, B. Shankar, and L. Roh. An evaluation of bottom-up and top-down thread generation techniques. In *Proceedings of the 26th Annual International Symposium on Microarchitecture*, pages 118–127, Austin, Texas, December 1–3, 1993. IEEE-CS TC-MICRO and ACM SIGMICRO.

[7] Vipin Chaudhary and J. K. Aggarwal. A generalized scheme for mapping parallel algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 4(3):328–346, March 1993.

[8] David E. Culler, Seth C. Goldstein, Klaus E. Schauser, and Thorsten von Eicken. TAM – a compiler controlled threaded abstract machine. *Journal of Parallel and Distributed Computing*, 18:347–370, July 1993.

[9] Jack B. Dennis and Guang R. Gao. Multithreaded architectures: Principles, projects, and issues. In Robert A. Iannucci, Guang R. Gao, Robert H. Halstead, Jr., and Burton Smith, editors, *Multithreaded Computer Architecture: A Summary of the State of the Art*, chapter 1, pages 1–72. Kluwer Academic Publishers, Norwell, Massachusetts, 1994. Book contains papers presented at the Workshop on Multithreaded Computers, held in conjunction with Supercomputing '91 in Albuquerque, New Mexico, November 1991.

[10] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freemann and Co., New York, New York, 1979.

[11] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4):276–291, December 1992.

[12] Apostolos Gerasoulis and Tao Yang. On the granularity and clustering of directed acyclic task graphs. *IEEE Transactions on Parallel and Distributed Systems*, 4(6):686–701, June 1993.

[13] M. Gien. Micro-kernal architecture: Key to modern operating system design. *UNIX Review*, November 1990.

[14] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: A survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[15] Laurie J. Hendren, Xinan Tang, Yingchun Zhu, Guang R. Gao, Xun Xue, Haiying Cai, and Pierre Ouellet. Compiling C for the EARTH multithreaded architecture. In *Proceedings of*

the 1996 Conference on Parallel Architectures and Compilation Techniques (PACT '96), pages 12–23, Boston, Massachusetts, October 20–23, 1996. IEEE Computer Society Press.

[16] J. E. Hoch, D. M. Davenport, V. G. Grafe, and K. M. Steele. Compile-time partitioning of a non-strict language into sequential threads.

[17] Herbert H. J. Hum, Olivier Maquelin, Kevin B. Theobald, Xinmin Tian, Guang R. Gao, and Laurie J. Hendren. A study of the EARTH-MANNA multithreaded system. *International Journal of Parallel Programming*, 24(4):319–347, August 1996.

[18] Herbert H. J. Hum, Kevin B. Theobald, and Guang R. Gao. Building multithreaded architectures with off-the-shelf microprocessors. In *Proceedings of the 8th International Parallel Processing Symposium*, pages 288–294, Cancún, Mexico, April 26–29, 1994. IEEE Computer Society.

[19] Robert A. Iannucci. A dataflow/von Neumann hybrid architecture. Technical Report MIT/LCS/TR-418, MIT Laboratory for Computer Science, Cambridge, Massachusetts, July 1988. PhD thesis, May 1988.

[20] Robert A. Iannucci. Toward a dataflow/von Neumann hybrid architecture. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 131–140, Honolulu, Hawaii, May 30–June 2, 1988. IEEE Computer Society and ACM SIGARCH. *Computer Architecture News,* 16(2), May 1988.

[21] Janusz S. Kowalik, editor. *Parallel MIMD Computation: HEP Supercomputer and its Applications.* Scientific Computation Series. MIT Press, Cambridge, Massachusetts, 1985.

[22] Sape J. Mullender, Guido van Rossum, Andrew S. Tanenbaum, Robbert van Renesse, and Hans van Staveren. Amoeba: A distributed operating system for the 1990s. *Computer*, 23(5):44–53, May 1990.

[23] R. S. Nikhil, G. M. Papadopoulos, and Arvind. *T: A multithreaded massively parallel architecture. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 156–167, Gold Coast, Australia, May 19–21, 1992. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 20(2), May 1992.

[24] Lucas Roh, Walid A. Najjar, Bhanu Shankar, and A. P. Wim Böhm. An evaluation of optimized threaded code generation. In Michel Cosnard, Guang R. Gao, and Gabriel M. Silberman, editors, *Proceedings of the IFIP WG 10.3 Working Conference on Parallel Architectures and Compilation Techniques, PACT '94*, pages 37–46, Montréal, Québec, August 24–26, 1994. North-Holland Publishing Company.

[25] Lucas J. Roh. *Code Generation, Evaluation, and Optimizations in Multithreaded Executions.* PhD thesis, Computer Science Department, Colorado State University, Fort Collins, Co 80523-1873, 1995.

[26] Shuichi Sakai, Kazuaki Okamoto, Hiroshi Matsuoka, Hideo Hirono, Yuetsu Kodama, and Mitsuhisa Sato. Super-threading: Architectural and software mechanisms for optimizing parallel computation. In *Conference Proceedings, 1993 International Conference on Supercomputing*, pages 251–260, Tokyo, Japan, July 20–22, 1993. ACM.

[27] Vivek Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Research Monographs in Parallel and Distributed Computing. Pitman, London and The MIT Press, Cambridge, Massachusetts, 1989. Revised version of the author's Ph.D. dissertation (Stanford University, April 1987).

[28] Vivek Sarkar. Instruction reordering for fork-join parallelism. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 322–336, White Plains, New York, June 20–22, 1990. *SIGPLAN Notices,* 25(6), June 1990.

[29] Klaus E. Schauser, David E. Culler, and Seth C. Goldstein. Separation constraint partitioning — A new algorithm for partitioning non-strict programs into sequential threads. In *Conference Record of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 259–271, San Francisco, California, January 22–25, 1995.

[30] Klaus Eric Schauser, David E. Culler, and Thorsten von Eiken. Compiler-controlled multithreading for lenient parallel languages. Report No. UCB/CSD 91/640, Computer Science Division, University of California at Berkeley, 1991.

[31] Klaus Erik Schauser. *Compiling Lenient Languages for Parallel Asynchronous Execution.* PhD thesis, Department of Electrical Engineering and Computer Science, Division of Computer Science, University of California, Berkeley, California, 1995.

[32] Andrew S. Tanenbaum. *Distributed Operating Systems.* Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1995.

[33] Kenneth R. Traub. Sequential implementation of lenient programming languages. Technical Report MIT/LCS/TR-417, MIT Laboratory for Computer Science, Cambridge, Massachusetts, October 1988. PhD thesis, September 1988.

[34] K.R. Traub, D. Culler, and K.E. Schauser. Global analysis for partitioning non-strict programs into sequential threads. In *Proc. the ACM Conf. on LISP and Functional Programming*, June 1992.

[35] T. Yang and A. Gerasoulis. List scheduling with and without communication delay. *Parallel Computing*, 19:1321–1344, 1993.

A Thread Partition Problem is NP-Hard

Thread partition for lenient languages is an NP-Complete problem [33]. In this section, we show that the problem as formulated in section 3 is NP-hard. Our proof is based on a demonstration that even a simple subset of the problem(e.g. threads are totally ordered (instead of a partial order)) is already NP-hard.

To prove this problem is NP-hard, a reduction from a well-known NP-Complete problem, 3-partition, to the thread partition problem is made as follows.

3-partition: given an integer b and a set of integers $S = \{a_1, \ldots, a_{3n}\}$ with $b/4 < a_i < b/2$, is there a partition of S into n disjoint subsets $S_1, \ldots, S_n$ such that $\sum_{i \in S_j} a_i = b$, for $j = 1, \ldots, n$?

3-partition is an NP-Complete problem [10]. The NP-hard proof of the thread partition problem is based on a reduction from the 3-partition problem.

We consider a restrict form of thread partition problem in which an input DDG has a linear chain of data dependence. We will prove that thread partition for this DDG instance is NP-hard.

Given an arbitrary 3-partition instance, we construct the corresponding DDG instance which is to be partitioned. We need to prove that the instance of 3-partition has an affirmative answer if and only if the corresponding DDG has an appropriate thread partitioning.

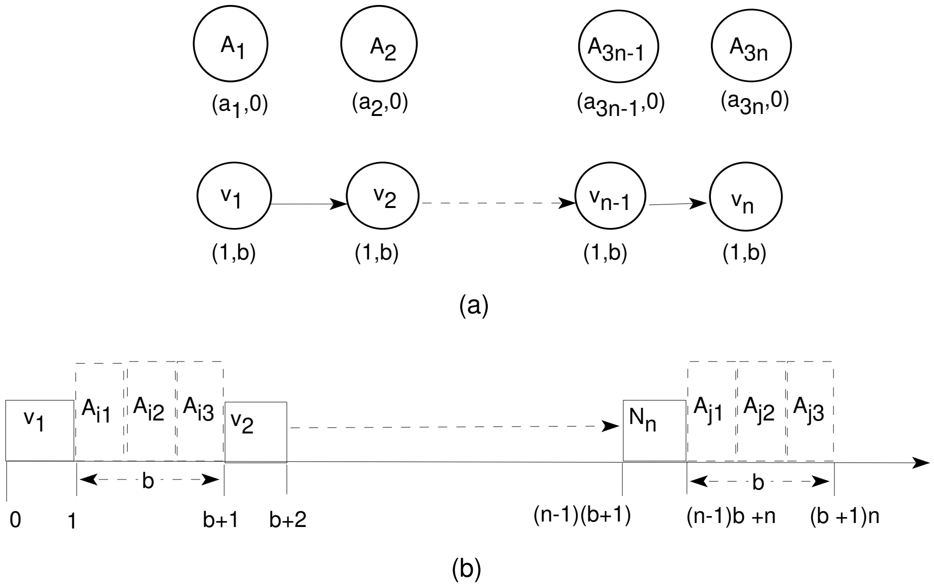


Figure 5: Linear chain of DDG and the schedule of the partitioned threads

The mapping is as follows:

- Each number a_i corresponds to a local node A_i with $c_i^L = a_i$ and $c_i^R = 0$.
- There is a chain of new n tasks,$v_1, \ldots, v_n$, in which:
  1. Each node v_i is a *remote* node with the cost of $c_i^L = 1$ and $c_i^R = b$, $i \in [1,\ n]$.
  2. Node v_i depends on its predecessor $v_{i-1}(i > 1)$ and thus the dependence relation forms a linear *remote* chain.

The corresponding DDG is shown in Fig. 5(a). A node is represented in circle and its execution cost is shown in below. In Fig. 5(a), the newly formed n tasks are called *separation* tasks in the final schedule and they are used to separate the tasks derived from the original 3-partition instance. There are no precedence constraints between node $A_i, i = 1, \ldots, 3n$, which are derived from the original 3-partition instance. But each v_j depends on its predecessor $v_{j-1}(j > 1)$. In Fig. 5(a), the remote nodes are connected by a linear chain.

Suppose that there exists a partition of S into $S_1, \ldots, S_n$ that yields an affirmative answer to 3-partition. A feasible thread partition with makespan no more than $n(b+1)$ is then obtained as follows:

- The chain of tasks $v_i(i = 1, \ldots, n)$ is partitioned first. Since there are n remote nodes, the chain is partitioned into n threads. Each thread contains only one remote node. This results in the following partition:

$$\begin{array}{rcl} T_1 & = & \{v_1\} \\ T_2 & = & \{v_2\} \\ & \vdots & \\ T_n & = & \{v_n\} \end{array}$$

  Because node $v_i(i \in [1, n-1])$ in thread T_i is a remote node with the remote cost $R_i = b$, the following thread T_{i+1} must wait b time to start after the local execution of the previous thread is finished. The interval of idle time is created between two consecutive threads. The intervals are: $[(i-1)b+i, ib+i](i = 1, \ldots, n)$. As shown in Fig. 5(b), The idle interval with length b exists between any two tasks v_i and $v_{i+1}(i \in [1, n-1])$ due to the remote dependence.

- Then schedule the local task $A_i(i = 1, \ldots, 3n)$ into the *idle* intervals. Because $\sum_{j \in S_i} a_j = b, i = 1, \ldots, n$, each S_i fits exactly into each idle interval. For instance, three tasks, $A_{i_j} \in S_1$, j=1,2,3, can be placed into interval [1, b+1]. This type of schedule is what multi-threading proponents like most. It uses local computation tasks from S_i to cover the remote operation from $v_j(j \in [1,\ n])$.

This scheme of thread partition generates the minimum execution time $n(b+1)$. If we assign the three local tasks from the same S_i to its predecessor thread, the final partition looks like:

$$
\begin{array}{rcllll}
T_1 & = & \{v_1, & a_{i_1}, & a_{i_2}, & a_{i_3}\} \\
T_2 & = & \{v_2, & a_{i_4}, & a_{i_5}, & a_{i_3}\} \\
 & \vdots & & & & \\
T_n & = & \{v_n, & a_{i_{3n-2}}, & a_{i_{3n-1}}, & a_{i_{3n}}\}
\end{array}
$$

There is no idle time between two consecutive threads.

Conversely, suppose that there exists a feasible thread partition with makespan no more than $n(b+1)$. It is clear that the schedule from this thread partition contains no idle time. Let S_1 be the index set of tasks that are completed in the interval [1, b+1]. It is impossible that $\sum_{i \in S_1} a_i > b$ due to the definition of S_1. The case $\sum_{i \in S_1} a_i < b$ cannot occur either, since this would lead to idle time in [1, b+1]. Therefore, we must have that $\sum_{i \in S_1} a_i = b$. Similarly, we can show that $S_2, \ldots, S_n$ constitutes a solution to 3-partition.

It is easy to see that the above transformation between 3-partition and the DDG based thread partition can be performed in polynomial time. Indeed, such a transformation can be done in $O(n)$ time.

Thus, we find a polynomial-time reduction from 3-partition to a subclass of the thread partition problem. It shows that this type of the thread partition problem is NP-hard.