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

Design and Implementation of an Efficient Thread Partitioning Algorithm

José Nelson Amaral,
Guang R. Gao, Erturk Dogan Kocalar,
Patrick O'Neill, Xinan Tang

CAPSL Technical Memo 30
July 1, 1999

(Amaral is the leading author, other authors are listed in alphabetical order.)

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

Recent advances in the development of a fine-grain multi-threaded program execution model based on off-the-shelf microprocessor technology has created an interesting challenge: how to partition a node program into threads that can exploit machine parallelism, achieve latency tolerance, and maintain reasonable locality of reference? A successful algorithm must produce a thread partition that best utilize multiple execution units on a single processing node and handles long and unpredictable latencies.

In this paper, we introduce a new thread partitioning algorithm that can meet the above challenge for a range of machine architecture models. A quantitative affinity heuristic is introduced to guide the placement of operations into threads. This heuristic addresses the trade-off between exploiting parallelism and preserving locality. The main algorithm is surprisingly simple and clean due to the use of a time-ordered event list to account for the multiple execution unit activities. We have implemented the proposed algorithm and our experiments, performed on a wide range of examples, have demonstrated its efficiency and effectiveness.

Contents

List of Figures

1 Introduction

The possibility of developing high-performance computer systems based on a fine-grain multi-threaded program execution and architecture model is attracting increased interest from the research community. Most recently, in his Ph.D. dissertation, Theobald has demonstrated the viability of an evolutionary path toward the implementation of fine-grain multi-threading based on off-the-shelf microprocessors [26]. The Efficient Architecture for Running THreads (EARTH) is such a fine grain multi-threaded execution and architecture model [11]. Implementation of EARTH on a number of hardware platforms has been completed, and preliminary results for real world applications have been reported [8, 13]. A brief description of the EARTH model and a comparison with other multi-threading models are included later in this paper and related references.

A key to the success of fine-grain multi-threading is the development of compilation methods that can efficiently exploit fine-grain parallelism in application programs and match them with the parallelism of the underlying hardware architecture. In particular, partitioning programs into fine-grain threads is a new challenge that is not dealt with in conventional compiler code generation and optimization.

For the purpose of this paper, thread partitioning can be viewed as an extension to a conventional prefetching mechanism, but different in the following sense:

- The prefetching operation and the operation that uses the prefetched value are placed in different threads when the latency is long and unpredictable.
- Such a *split-phase* prefetching combines communication and synchronization: when the fetch is completed, the destination thread, i.e. the thread that contains the operation that depends on the prefetched value, is signaled.

Under the EARTH model, a thread becomes *enabled* for execution if and only if it has received signals from all the split-phase operations that it depends on. Furthermore, threads are non-preemptive: once a thread is scheduled for execution, it holds the execution unit until its completion. Therefore whenever an operation may involve long and/or unpredictable latencies, the role of a compiler (or programmer) is to make the operation "split-phase." We call this requirement *split-phase constraint*, and assume that such constraints are explicitly identified and presented to the thread partitioner. This explicit constraint might be provided by programmer annotation or through compiler analysis, or by a combination of both. Note that the split-phase constraint analysis should be straightforward based on the experience working with its implementation in the EARTH-McCAT compiler [9]. For example, we can simply translate memory operations that are likely to incur high or unpredictable latency to machine-level split-phase operations. For instance, a memory operation that is likely to require a deep access in the memory hierarchy is a candidate to be labeled as a split-phase operation. Once the split-phase constraints are established, the thread partitioning problem studied in this paper can be stated as follows.

Given a machine model M and a weighed data dependence graph G labeled with its split-phase constraints, i.e. certain nodes are labeled split-phase nodes, partition G into threads such that the total execution time of G is minimized subject to the split-phase constraints.

In this paper the execution and architecture models for the machine M are based on the EARTH model outlined by Theobald [26]. A feature of the EARTH execution model is that threads are non-preemptive: once a thread is scheduled for execution, it will not give up the execution unit it is assigned to until completion. The thread execution units interact through a hierarchy of memories. The key architecture parameters are: the number of execution units per processing node, the range of timing cost for local and non-local (split-phase) operations, and the cost of spawning or terminating a thread.

The main contribution of this paper is the development of an heuristic thread partitioning algorithm suitable for this new machine model.[1] Unlike previous related thread partitioning algorithms, ours faces a new challenge: the existence of more than one thread execution unit per node implies a trade-off between the need to generate enough parallel threads per node to utilize these execution units, and the need to assign related operations to the same thread to enhance locality of accesses.

This new algorithm has been implemented and tested. Experiments were performed for a wide range of DDGs on several machine architecture models based on machine parameters gathered and used in [26] and elsewhere. Our main observations include:

- The proposed algorithm is very efficient (compared with the the best possible algorithm) for a wide range of DDGs on all the machine models except the EARTH-Dual model (which only has a single execution unit per node).
- The algorithm is robust in terms of its latency tolerance capacity for a wide range of high remote latencies.
- Key data structures (an event list ordered on a time line) and a specially designed search algorithm (TestGap presented in Section 5) are very effective. Our experimental data shows that the average number of events visited by the search algorithm to decide about the placement of an operation is proportional to the number of EUs in the machine for DDGs with a large number of nodes. Our experiments included several thousands of partitions of very large DDGs – with up to 1000 nodes with several thousand edges and were performed in very reasonable time!

In Section 2, we present the abstract machine model including a brief description of the synchronization mechanism and a high level presentation of the programming model used in this

[1]Notice that the algorithm presented in [25] collapses as much local computation as possible into a single thread, thus making it inadequate for the machine model studied in this paper that has multiple execution units per processing node

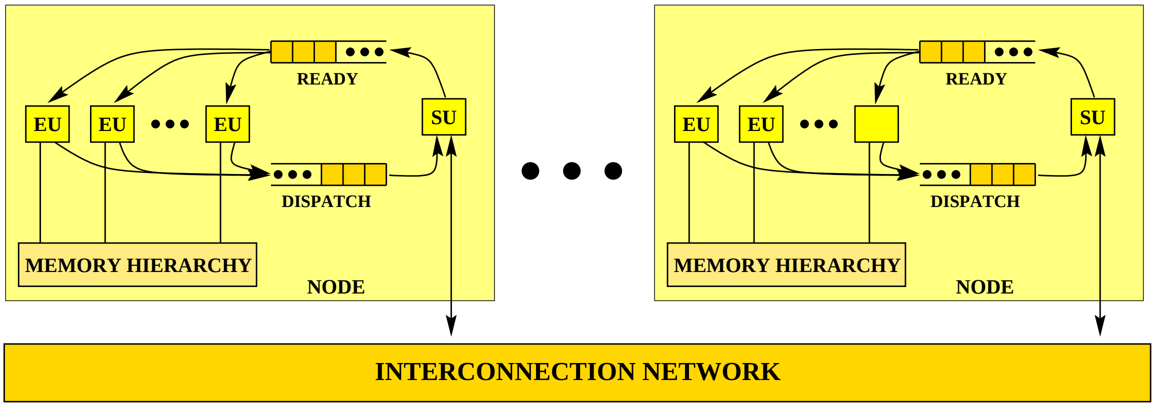

Figure 1: Architecture abstract model.

work. Section 3 illustrates the thread partitioning problem through an example. In Section 4, we introduce the definitions necessary to formalize the algorithm. Section 5 presents the formal specification of the algorithm, highlighting relevant aspects with examples. The experimental results presented in Section 6 demonstrate that our new algorithm obtains thread partitions that are close to the optimal for a large set of randomly generated DDGs. In Section 7 we discuss related work.

2 Thread Partitioning: A Programming and Machine Model

We consider a machine model that can represent both an architecture with multiple and interchangeable functional units and with specific hardware to support synchronization among multiple threads, and a cluster of symmetric multiprocessing (SMPs) nodes that has a private cache memory for each EU. Thus the partition algorithm heuristic is oriented towards improving locality of references among the statements assigned to the same thread.

2.1 Machine Model

The architecture abstract model used to study the thread partitioning problem in this paper is presented in Figure 1. Each processing node has N execution units (EU) and one synchronization unit (SU). Both the EU and the SU perform the functions specified in the EARTH Virtual Machine [26]. Threads that are ready for execution are placed in the ready queue. Whenever idle, an execution unit fetches a thread from the ready queue. When an active thread performs a synchronization operation or requests a long latency data transfer, the request for such a service is placed in the dispatch queue that is serviced by the synchronization unit. The synchronization unit is responsible for the communication with all other processing nodes and for the synchronization between threads within the node.

In the model of Figure 1 each processing node has its own memory hierarchy. Any processor can access the memory locations in any node of the machine. However, an access to a location in the local memory hierarchy has a lower latency and higher bandwidth than an access to a location in the memory region of any other node. Thus, a read or write operation to an address that is not in the local memory region is called a **remote access** while read and write operations to addresses in the local memory region are called **local access**. As in [25], we assume that a cost model is provided that allows for an estimation of the cost of all the local and all the remote operations required in a program statement. We also assume that the cost associated with the termination of a thread and the start of the execution of another thread is δ. Although we will assume that a ready thread can be executed by any one of the local processors, the partitioning algorithm will take into consideration the amount of dependencies among statements when placing them in different threads to favor data locality and benefit from the local caches in the architecture.

To ensure that threads can run to completion, when a compiler has determined that an operation *may* incur long and/or unpredictable latencies, the statement that requests the operation is placed in one thread and the statements that depend on the results of the operation are placed in a separate thread. This mechanism is denominated *split-phase* synchronization. It allows the runtime system to hide long latencies by executing other enabled threads while the long latency operation is performed. This threading mechanism is efficient for applications that have enough fine grain parallelism to keep most of the execution units in a processing node busy. All the synchronization and data communication operations are performed by the *Synchronization Unit.*

A given thread may have to wait for a collection of events to happen, therefore a synchronization *slot* is associated with each thread. The SU decrements a counter associated with a slot whenever the slot receives a synchronization signal. When the counter reaches zero the SU places the newly enabled thread in the ready queue where it waits for an idle EU.

2.2 Cost Model

We assume that a program is written in a sequential language augmented with high level parallel constructs, and that the data has been partitioned among the memory modules in the processing nodes of the machine. Therefore, given a program statement we can determine whether the statement is *local* or *remote*. We also assume that the program has been translated into a Data Dependence Graph (DDG).

Thus the program is represented by a graph $G(V, E)$ where each node in V represents a collection of program statements. A node can be a *simple node* such as an assignment statement or a *compound node* such as a loop. If the execution of the program statements requires accesses to a remote memory module, the node is a *remote node* otherwise the node is a *local node*. Each edge $(v_i,\ v_j)$ in E represents a data dependency from v_i to v_j. An edge departing from a remote node is a *remote edge*, and an edge departing from a local node is a *local edge*. Like in [25] we represent E by an adjacency matrix:

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

Parameter	Dual-processor	Single processor
Latency (cycles)	204.5	122.5
Bandwidth (MB/s)	42.0	28.8

Table 1: Latency and bandwidth for the EARTH-MANNA implementation. The MANNA architecture has a cross-bar interconnection network, reducing the impact of network contention on remote operations.

The partitioning algorithm is based on a cost model that associates a *local cost* c_i^L and a *remote cost* c_i^R to each node v_i. If a node is local, its local cost is the number of cycles that it spends in the EU. The local cost of a remote node is the time that it spends in the EU plus the time that the EU needs to dispatch requests to the SU. The remote cost of a local node is zero, while the remote cost of a remote node is the time required from the dispatch to the completion of the remote operations.

Observe that the remote cost assumes that the program would dispatch the request for the remote operation and suspend the execution until the result comes back. One of the main goals of the partitioning algorithm is to decide when it is advantageous to dispatch the request and terminate the current thread and to start the execution of a new thread when the remote operation is completed.[2] For some interconnection networks a model to predict the network performance will be necessary. This cost will be affected by the load in the network. For the experiments with our partitioning algorithm, we assume that this cost is bounded by a constant.

The thread partitioning and scheduling problem for the multiple EU case has a similar formulation as the one given in [25] for the single EU case. However, the existence of multiple functional units in our machine model creates a tradeoff between collapsing many local nodes into a single thread to avoid thread switching costs and generating multiple threads to exploit parallelism within a processing node. The graph of nodes must be partitioned into threads in such a way that all the dependencies are obeyed, all remote edges become split phase transactions, and the total execution cost is minimized for the machine and cost models adopted.

The cost model presented in this machine can be constructed for any given machine through profiling experiments and examination of the machine specification for the number of cycles required to execute each class of instruction. The network latency and bandwidth can also be readily measured.[3] For instance for the EARTH-MANNA implementation, Theobald reports the values presented in Table 1 for the latency and bandwidth for communication between processing nodes. The values from Theobald's experiments were used in the machine models described in our experimental section.

[2] We informally refer to this separation of a sequence of programming statements into two threads to cope with long latency operations as "cutting" the program block into two threads.

[3] More elaborate models that include network contentions are required for a more precise cost modeling

3 A Thread Partitioning Example

Figure 2 shows a simple DDG, where each node v_i is annotated with a cost tuple (c_i^L, c_i^R) formed by its local and its remote cost. Local edges are represented by solid lines and remote edges are represented by dashed lines. Our goal is to partition the nodes of the DDG into threads to minimize the total execution time while satisfying data dependencies and performing all remote operations as split-phase transactions. Figure 3 shows three partitions of the same annotated DDG. In this section we discuss how each one of these partitions is obtained.

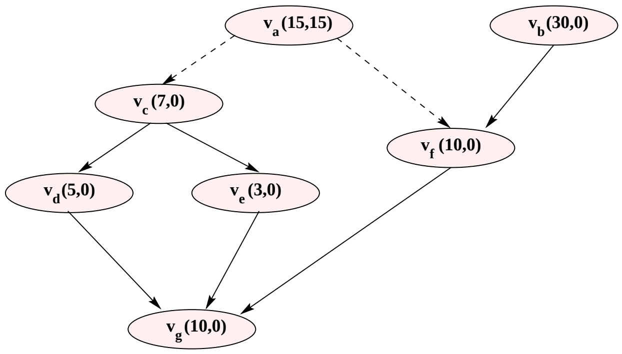

Figure 2: Example of data dependency graph representing a program.

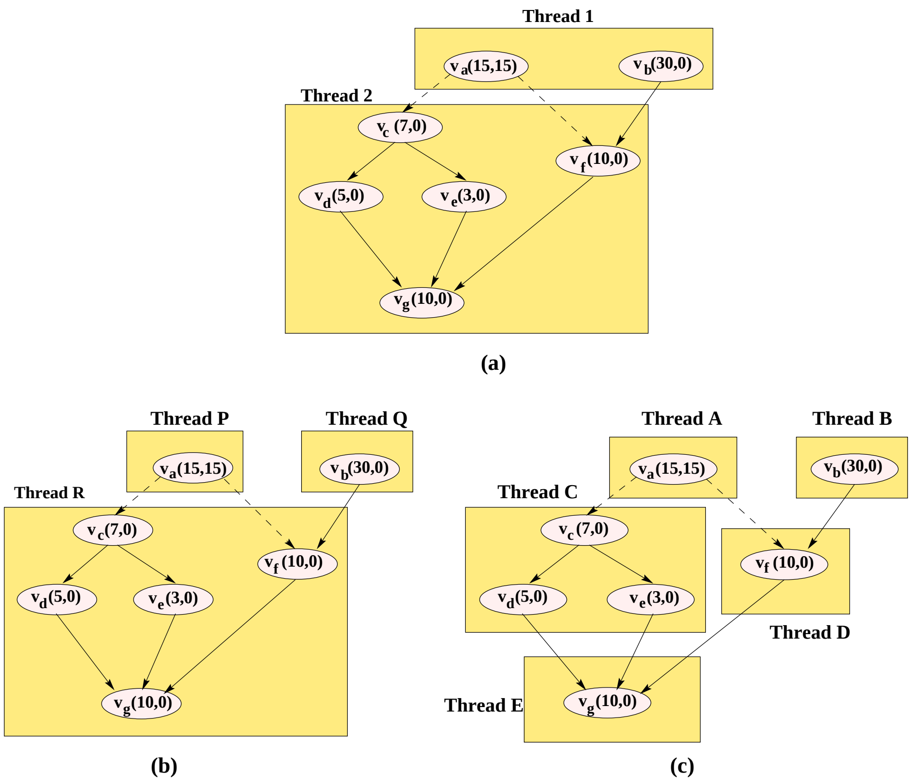

Figure 3: Three different partitions for the DDG in figure 2.

Figure 4 shows possible schedulings for the partitions shown in Figure 3. The first partition in which $T_1 = \{v_a, v_b\}$, and $T_2 = \{v_c, v_d, v_e, v_f, v_g\}$ is obtained for a machine that has a single EU. Note that in all three partitions, the source and the destination of a remote operation are placed in separate threads.

The two schedules on top of Figure 4 are for a machine with a single EU. The partition obtained by the algorithm is the one presented in Figure 3(a). The top schedule assumes a thread switching cost $\delta = 20$ resulting in a total execution time of 100. The second schedule assumes $\delta = 2$ and results in a total execution time of 82.

The partition on Figure 3(b), $T_P = \{v_a\}$, $T_Q = \{v_b\}$, and $T_R = \{v_c, v_d, v_e, v_f, v_g\}$ is obtained for a machine with two EUs per processing node and $\delta = 20$. In this case, there is an advantage on separating nodes v_a and v_b into two threads because the existence of two EUs allows for the overlaping of the local computation in these two nodes. A possible schedule for this partition is given on the third schedule from the top of Figure 4.

For a machine with two EUs and $\delta = 2$, we obtain the partition in Figure 3(c) with $T_A = \{v_a\}$, $T_B = \{v_b\}$, $T_C = \{v_c, v_d, v_e\}$, $T_D = \{v_f\}$, and $T_E = \{v_g\}$. For this machine the partition into smaller threads allows for the overlaping of computation between threads T_C and T_D. Notice that the node v_g cannot be assigned to either T_C or T_D without causing the starting of such threads to be delayed and thus eliminating the possibility of computation overlap.

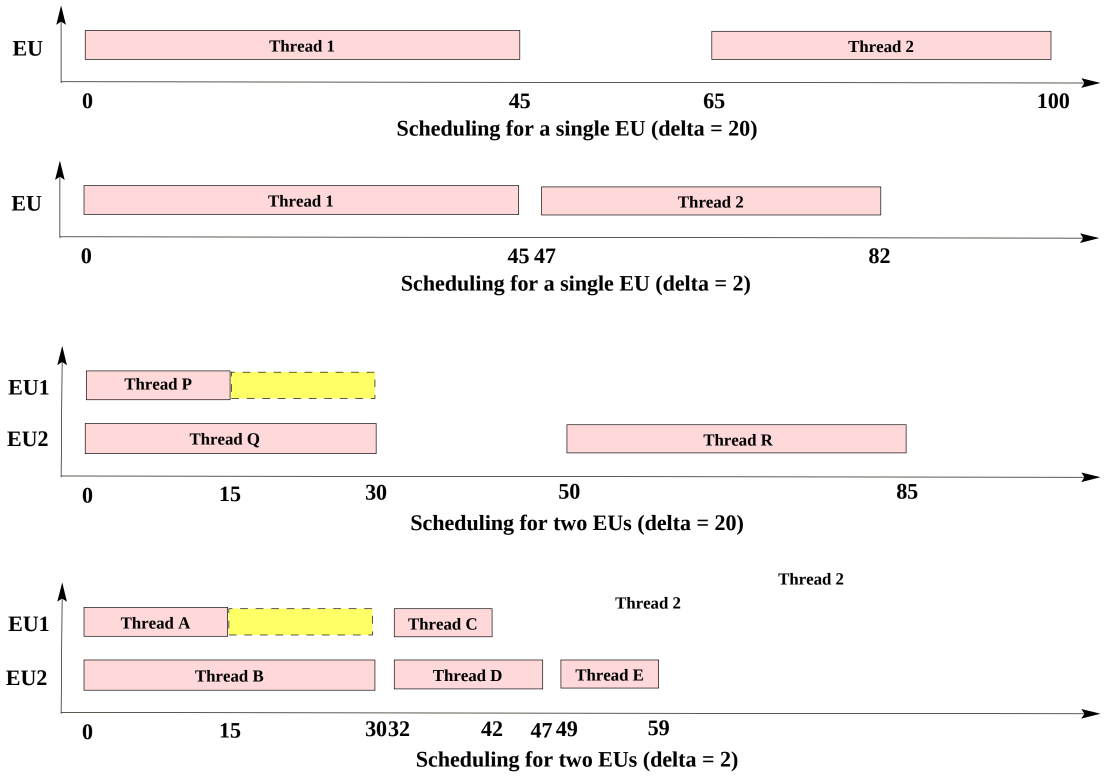

Figure 4: Thread partition and execution timings.

4 Problem Formulation

In this section we use small case letters to refer to quantities associated with nodes of the graph G, and capital letters to refer to quantities associated with threads of the partition P. For instance, l_k is the remote level of the node v_k of G, and L_k is the remote level of the thread T_k in P.

Start Node and End Node: Two nodes, v_{start} and v_{end}, are added to the data dependency graph G. Edges are added from v_{start} to every node without predecessors, and from nodes without successors to v_{end}. After this transformation, G has a single entry point and a single exit point.

Height: The height of a local node v_i, h_i, is defined as

$$h_i = c_i^L + \max_{j:C_{i,j}=1} h_j$$

The height of a remote node v_i is defined as

$$h_i = c_i^L + \max_{j:C_{i,j}=1} h_j + c_i^R + \delta$$

where C is the adjacency matrix defined in Section 2.2. By definition, $h_{end} = 0$. The height of a node is a measurement of the critical path from the starting time of the node execution to the termination of the program. The height of a node is measured in cycles, from the node v_i to the end node v_{end} including the local and remote costs of v_i. If the node is a remote node, then δ must also be added to h_i, because there will be a thread switch after the execution of v_i and before the execution of its successor nodes.

Node Remote Level: the remote level of a node, l_i is defined as:

$$\begin{aligned} l_{start} &= 0 \\ l_i &= \max_{j:C_{j,i}=1}\left(l_j + \left\{\begin{array}{ll} 0 & \text{if } v_j \text{ is local} \\ 1 & \text{if } v_j \text{ is remote} \end{array}\right\}\right) \end{aligned}$$

The remote level of a node v_i is the maximum number of remote edges along any path from v_{start} to v_i. The remote level of v_i is a lower bound on the number of thread switches that must occur before the execution of node v_i can start.

Thread Remote Level: the remote level of a thread, L_i, is equal the remote level of all the nodes that are assigned to the thread.

Notice that by definition, two nodes that have different remote levels cannot be assigned to the same thread.

Node Starting Time: the starting time, s_i, of node v_i is the time when the node starts executing in the schedule of a partition P of G.

Node Finishing time: the finishing time of node v_i is the time when the node operations (local and remote) are completed and the results produced by v_i are available for use by its successors.

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

Ready Time: the ready time r_i of a node v_i is the time when all the data required by the node is available, i.e. all the predecessor of v_i in G have finished their execution:

$$r_i = \max_{j:C_{j,i=1}} f_j$$

Observe that for any node $v_i \in G$ we must have $r_i \leq s_i$.

Thread Starting Time: the starting time, S_k, of thread T_k is the time when the thread starts executing in the schedule of the partition P.

Thread Finishing Time: the finishing time, F_k of thread T_k is the time when all *local* operations in the thread have finished plus the thread switching cost. It is given by

$$F_k = S_k + \delta + \sum_{j:v_j \in T_k} c_j$$

Parent Threads: the set of parent threads, $PT(v_i)$, of a node v_i is formed by all threads $T_k \in P$ such that there is a node $v_j \in T_k$, $C_{ji} = 1$ and $c_j^R = 0$.

Affinity: the affinity of a node v_i of G to a thread T_k of P, $A(v_i, T_k)$, is given by the ratio between the number of dependences between nodes of T_k and v_i, and the total number of incoming edges of v_i.

$$A(v_i, T_k) = \frac{\sum_{v_j \in T_k} C_{ji}}{\sum_j C_{ji}}$$

Observe that if all the incoming edges of v_i are from nodes in T_k, then $A(v_i, T_k) = 1$. On the other hand, if none of the incoming edges of v_i are from nodes in T_k, then $A(v_i, T_k) = 0$.

When a node v_i is assigned to a thread T_k, the resulting thread cannot start before the original starting time of the thread or before the ready time of v_i. Thus, the delay incurred because v_i is assigned to thread T_k can be computed by:

$$d(v_i, T_k) = \begin{cases} F_k - r_i & \text{if } PT = \{T_k\} \\ \max(r_i, S_k) + (F_k - S_k) - r_i & \text{otherwise} \end{cases}$$

We define a penalty function $F(v_i, T_k)$ that is used to determine how well a node v_i fits into a thread T_k:

$$P(v_i, T_k) = D\ d(v_i, T_k) + L\ (1 - A(v_i, T_k))$$

where D and C are arbitrary constants; D represents the influence of a delay in the execution of node v_i in the total execution time. L represents the importance of data locality that is obtained by favoring the addition of v_i to a thread that already contains many of its predecessors. The algorithm that partition nodes into threads will assign a node v_i to the thread T_k in which the node incurs minimal penalty.

PARTITION(G, V)
1. Add v_{start} and v_{end} to G
2. Calculate h_i and l_i for every node v_i in G
3. $P \leftarrow \emptyset$
4. $E \leftarrow \emptyset$
5. $F \leftarrow \{v_{start}\}$
6. **while** $F \neq \emptyset$
7. $v_i \leftarrow$ Highest node in F
8. $T_k \leftarrow$ EarliestThread(P, E, v_i)
9.   InsertIntoThread(v_i, T_k)
10.   UpdateReadyList(F, v_i)
11.   UpdateEventList(E, T_k)

Figure 5: Partition algorithm.

With the definitions above, we can now introduce the statement of the optimization problem that we want to solve in this paper:

Problem Statement

Assume that at runtime, threads are selected for execution from a single queue of ready threads using an efficient scheduler. Given a DDG G with each node $v_i \in G$ annotated with its local cost c_i^L and its remote cost c_i^R, and a constant thread switching cost δ, find a thread partition P that meets the following goals:

Goal 1: Minimizes the total execution time by minimizing the starting time of the end node s_{end}.

Goal 2: Maximizes the affinity between the nodes assigned to the same thread.

These goals should be pursued under the constraint that nodes connected by a remote edge are assigned to different threads.

The first goal is the principal goal of the partition algorithm. The second goal is necessary to favor locality of access because the abstract model assumes that all processors have equal probability of fetching a thread from the ready queue. The constraint enables the multithreading mechanism to hide the latency of remote operations by enabling the processor to switch to other ready threads.

5 Partition and Scheduling Algorithm

The partition algorithm computes a thread partition P based on the local and remote costs recorded in the graph G. The algorithm PARTITION(G, V), presented in Figure 5, initializes a partition P as an empty set of threads (step 3), the event list E as an empty list (step 4), and creates a *Ready List* F, where it puts the node v_{start} in the list F (step 5). In each iteration of the while loop, the algorithm takes the highest node v_i from the ready list (step 7) and either creates a new thread in partition P or selects one of the threads already in P to insert the node v_i (step 8). Then, the algorithm inserts all nodes that are not yet scheduled and whose precedent nodes have all been scheduled in the ready list (step 10), and updates the event list (step 11). This process is repeated until the ready list becomes empty. Each item in the event list contains the time and the type of event. An event of type S represents the starting time of a thread and an event of type F represents the finishing time of a thread. The list E is kept ordered according to the time of the events.

We will expand on the role of the event list when we detail the EarliestThread algorithm. The use of such a list is crucial to reduce the complexity of the algorithm that searches for the best thread to contain the node v_i. Because the event list is maintained in order, both locating the ready time of a thread and inserting an event can be performed in $\log(n)$ time, where n is the number of events already in the list.

A property of the graph G is that there is a path from v_{start} to every node in G. Thus the loop in steps 6-11 of the partition algorithm eventually iterates over all the nodes in G. Each iteration schedules one node. Step 7 selects the *highest* node, v_i, from the ready list F. The EarliestThread algorithm, in step 8, returns the thread to which the node v_i is to be assigned. The InsertIntoThread algorithm, in step 9, finds the best scheduling for node v_i in the thread selected by EarliestThread.

5.1 Earliest Thread Algorithm

The EARLIESTTHREAD(P, E, v_i) algorithm uses the penalty function to trade-off scheduling v_i as early as possible and assigns it to a thread that already contains operations that are closely related to v_i. First, EarliestThread verifies if v_i can start its own thread at its ready time r_i (TestGap in line 2). The TestGap algorithm, detailed in the next section, returns true if the gap specified can accommodate a thread and returns false if it cannot. If there is enough idle time in any EU to fit v_i in its own thread at its ready time, a new thread is created (line 5). The creation of a new thread for a node when there are idle EUs is a mechanism to exploit the maximum parallelism existing in the application, and is a distinct feature of the partition algorithm for the multiple EU architecture.

Next, the EarliestThread algorithm computes the minimum penalty, P_m, among all the threads in the set of parent threads of v_i, $PT(v_i)$ (lines (4-11). Only threads to which v_i can join are accepted as minimum penalty threads. The TestGap* algorithm called in line 11 verifies if a gap in the schedule exists to accommodate the new thread formed by the union of v_i to T_k when the current thread T_k is removed from the schedule.

If a parent thread with minimum penalty was found, it is already an acceptable solution. However there is a possibility that v_i could have less penalty if it would join another thread that finishes earlier than the selected parent. This situation arises when a node v_k has many descendents. Assume that $v_k \in T_p$. After some descendents of v_k join T_p, a node v_i that is

```
EarliestThread(P, E, v_i)
1.  Compute r_i
2.  if TestGap(P, E, r_i, r_i + c_i^L + δ)
3.     then return T_new
4.  P_m ← ∞
5.  for T_k ∈ PT(v_i)
6.       do p ← P(v_i, T_k)
7.          if p < P_m
8.             then S' ← max(r_i, S_k); F' ← S' + F_k − S_k + c_i^L
9.                   if TestGap*(P, E, S', F' + δ, T_k)
10.                    then m ← k
11.                         P_m ← p
12. f ← EarliestFinishEvent(E, r_i)
13. while (F_f − r_i) D < P_m and f < n
14.         if P(v_i, T_f) < P_m and l_i = L_k and TestGap(P, E, F_f, F_f + c_i^L)
15.           then return T_f
16.         if TestGap(P, E, F_f, F_f + c_i^L + δ)
17.           then return T_new
18.         f ← f + 1
19. return T_m
```

Figure 6: Algorithm that finds the earliest starting thread to which v_i can be inserted without causing any scheduled thread to be delayed.

descendent of v_k will be ready as soon as v_k finishes. However, if v_i joins T_p, v_i will be delayed by all the other descendents of v_k. If there are EUs idle earlier, v_i is better off if it joins another thread or creates its own thread at an earlier time.

Next the algorithm finds the thread, T_f, that is the earliest thread to finish after the ready time, r_i, of v_i (line 12). The EarliestThread algorithm assumes that the event list E is kept ordered according to the time of the finish event associated with each thread. Thus, the position of T_f in the list can be computed in $\log(n)$ steps in the worst case, where n is the total number of events in the list. The while loop in lines 13-18 scans the event list from the position of the earliest finish event to find the earliest gap in the schedule that is large enough to fit v_i. The search stops when there are no more events in the list or when the search has advanced far enough in time for any position found for v_i to result in a penalty that is higher then the minimal penalty P_m among the parent threads of v_i.

If the penalty for v_i to join T_f, $P(v_i, T_f)$, is smaller than P_m, the minimum penalty for v_i to join one of its parent threads, and a suitable gap is found (test in line 14), v_i can join T_f. In this case the TestGap in line 16 tests for a gap big enough to contain v_i plus one thread switching cost.

On the other hand, if v_i cannot join T_f, line 16 tries to find a a gap at the end of thread T_f. To contain v_i this gap needs to fit the local cost of v_i, c_i^L, plus the thread switching cost, δ. If such a gap is found, v_i can be in a thread by itself.

If both TestGaps fail, f is updated in line 18 to test the next thread finishing time and

```
TestGap(P, E, s, f)
  a ← 0
  for T_k ∈ P
      if S_k ≤ s and F_k > s
         then a ← a + 1
              if a ≥ N
                 then return false
  find i such that time(E(i − 1)) < s ≤ time(E(i))
  for i to length[E]
      if type(E(i)) = F
         then a ← a − 1
      else a ← a + 1
         if a ≥ N
            return false
      if time(E(i)) > s
         then return true
```

Figure 7: Algorithm that verifies if there is at least one EU idle in the interval $[s; f]$.

verify if a gap for v_i is found at that time. If the search has advanced past the point where there is hope to find a position for v_i that is better than joining T_m, then the EarliestThread returns T_m.[4]

A much simpler algorithm could be considered in which the penalties for v_i to join all the threads of P is computed and the one with the minimum penalty is chosen. We claim that our algorithm is more efficient because it only computes the penalty for v_i to join the threads in its parent threads set, $PT(v_i)$, and a limited number of threads that terminate after the ready time of v_i, and before the starting time of the parent with minimum penalty.

5.2 Test Gap Algorithm

The TestGap(P, E, s, f) algorithm is at the core of the partition algorithm presented in this paper. It searches a list of events to verify if at least one EU is available during the time interval $[s; f]$. Intuitively, it counts how many threads are scheduled at time s, and then traverses the event list starting at s adding to the counter when a starting event is found and subtracting from the counter when a finishing event is found. If at any time during the interval $[s; f]$ the counter is equal to the number of EUs available in the machine, it returns false. If it reaches the end of the interval f without ever having more than $N-1$ threads at the same time, where N is the number of EUs in the machine, it returns true.

In Figure 7, a is the counter that stores the number of active threads, time($E(i)$) is the time of the event in position i of the ordered list of events E, and type($E(i)$) is F if the event is the finishing of a thread and S if the event is the starting of a thread.

[4]If a suitable parent thread T_m was not found in the for loop in lines 5-11, a new thread is always created for

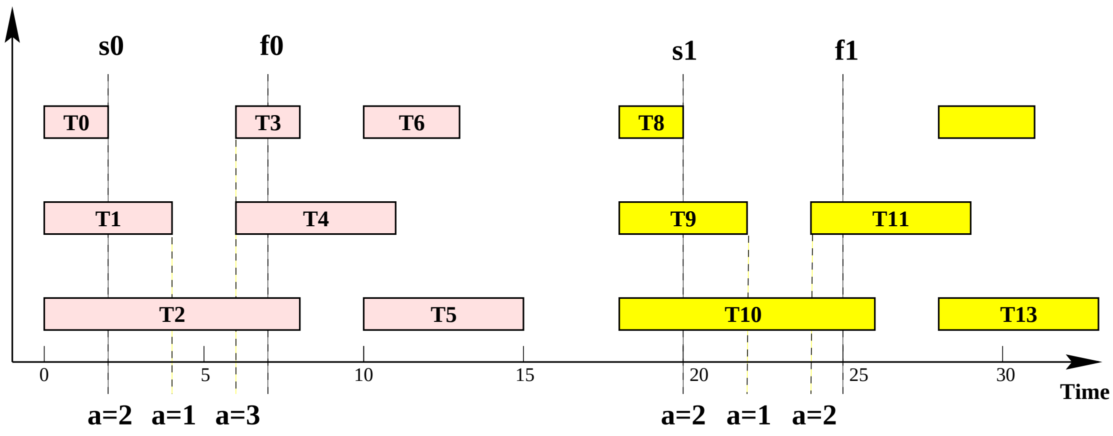

Figure 8: Example of operation of the algorithm TestGap.

Figure 8 presents an example to illustrate the operation of the TestGap algorithm. This example assumes an architecture with three EUs per processing node and presents two separate calls for the TestGap algorithm. The first call specifies the interval $[2, 7]$ while the second call specifies $[20, 25]$. The following events are relevant for the first call:

$$E = \{(0, S_1); (0, S_2); (4, F_1); (6, S_3); (6, S_4)\}$$

First, the algorithm counts the number of active threads at time s_0, and finds $a = 2$. At time 4, thread T_1 finishes making $a = 1$. At time 6, the starting of threads T_3 and T_4 causes the number of active threads to be 3, and the TestGap algorithm returns false because it is not possible to fit a thread of length 5 in the gap starting at time 2.

For the second call to FindGap for the interval $[20; 25]$, the following events are relevant:

$$E = \{(18, S_9); (18, S_{10}); (24, S_{11})\}$$

Now, at the beginning of the interval there are two active threads, $a = 2$. At time 22, thread T_9 finishes making $a = 1$; at time 24 thread T_{11} starts making $a = 2$, and the next event, at time 26, is after the end of the interval. The counter a was never equal to the number of EUs, therefore TestGap returns true.

5.3 Inserting a Node Into a Thread

The insertion of a node in a thread follows the algorithm presented in [25]. For completeness this algorithm is shown in Figure 9. To avoid delaying threads that depend on v_i, when inserting v_i into a thread T_k, v_i should be scheduled as early as possible. The InsertIntoThread algorithm puts v_i at the end of the thread and tries to move it towards the beginning. There are two constraints that prevent a node from moving towards the beginning. First, there may be a data dependency between the previous node and v_i, and therefore v_i cannot move before the

v_i at the end of the schedule in line 17.

```
InsertIntoThread(v_i, T_k)
1.  j ← index of the last node in T_k
2.  repeat
3.  if C_{j,i} = 1 or max(h_i, c_i^L + h_j) ≥ max(h_j, c_j^L + h_i)
4.     then insert v_j after v_i
6.           return
7.     else  j ← index of the previous node of v_j
9.  until v_j is the head of the list
10. insert v_i at the head of thread T_k
```

Figure 9: Algorithm that inserts a node into a thread.

previous node. Second, the previous node may be on the critical path and by moving before that node, v_i may delay it.

To decide whether v_i can be scheduled earlier than another node v_j, the following conditions are verified:

- Check for Data dependency. If $C_{j,i} = 1$ then v_i depends on v_j and therefore should not be moved before v_j. Insert v_i after v_j and return.

- Check for Critical Path. The decision of which node, v_i or v_j, might be in the critical path is based on the heights of v_i and v_j. The height of a node is a lower bound on the number of cycles from the beginning of the node to the end of the program. In line 3, the InsertIntoThread algorithm compares the height of the pair (v_i, v_j) for both orders of execution.

6 Experimental Results

6.1 Experimental Framework

We use the Thread Partition Test Bed presented in [25] to generate random DDGs to test the partition algorithm. To represent a variety of user programs, we vary several properties of the DDGs generated, including the number of nodes, the average number of outgoing edges from a node, the percentage of remote nodes in the graph and the distribution of local and remote costs in the nodes.

The distribution of execution costs for the nodes is as follows: A local node can be of three types: 1) Local I/O (20%), 2) Local function call (10%), 3) Other(math,etc.) (70%). A local I/O is assigned a cost of 10 cycles. A local function call is assigned 10 cycles and a node of other type has 3 cycles. Remote nodes can be of type: 1) Remote I/O (80%), 2) Remote function call (20%). A remote I/O is assumed to take 300 cycles, and the costs of remote function calls are uniformly distributed between 400 cycles and 4000 cycles. These are summarized in Figure 10.

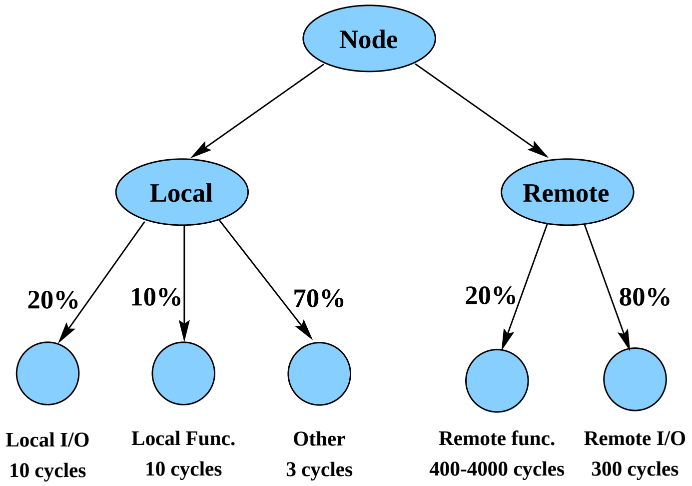

Figure 10: Distribution of node types.

6.2 Summary of Main Experimental Results

The main results of our experiments can be summarized as follow.

1. **Absolute Efficiency** (For details see Section 6.4). The new algorithm presented in this paper is very efficient for a wide range of DDGs on all the machine models except the EARTH-Dual model (see Section 6.3) where only a single EU is available per node and there is a high cost associated with thread switching. The algorithm performs remarkably well when the machine model has multiple execution units – e.g. for SMP and SCMP models – the average absolute efficiency is above 99%.

2. **Effectiveness of Search Heuristics** (For details see Section 6.5.) The key data structure (an event list on a time line) and the search step (see lines 2, 9, 14 and 15 in the algorithm presented in Figure 6) are very effective. The low growth on the number of events visited in order to decide to which thread to assign an operation is key to the effectiveness of the algorithm.

3. **Latency Tolerance Capacity** (For details see Sections 6.1, 6.6, and 6.4.) The algorithm is robust in terms of its latency tolerance capacity.

6.3 Machine Architectures

We define four different machine architectures for our experiments. MANNA is a multiprocessor machine with 40 processors distributed in 20 processing nodes interconnected by a crossbar switch. MANNA is the first platform in which the EARTH model was implemented [14].

EARTH-MANNA-DUAL: An implementation of the EARTH architecture on the MANNA machine. The second processor is used for the SU function. Therefore, according to our model, this is a machine with a single EU per node. The thread switching cost in this machine is $\delta = 36$ cycles (see measurements reported in [26]).

EARTH-MANNA-SPN: The two processors in the machine are used to implement the SU functions. Therefore, this machine has two EUs per processing node. The thread switching cost is $\delta = 16$ (see measurements reported in [26]).

EARTH-SU: This is the EARTH architecture with a custom hardware SU. We consider a machine with a single execution unit per processing node and with a thread switching cost $\delta = 2$ (see measurements reported in [26]).

SMP: This is an Symmetric Multi-Processor machine with 4 processors per node. In such a machine we expect the thread switching cost to be similar to the one for the MANNA-SPN, therefore we will use $\delta = 16$.

Single-Chip Multi-threaded Processor (SCMP) In this case we consider a hypothetical machine that has multiple functional units with multi-threading support. We assume a machine with 8 EUs and with a thread switching cost of $\delta = 10$.

6.4 Measuring Absolute Efficiency

The actual execution time of a program depends on the runtime scheduler that dynamically schedules thread executions. Our partitioning algorithm assumes that the threads are scheduled to run in the order in which they become ready by an efficient FIFO scheduler. Even an optimal partition cannot result in an execution time that is shorter than the critical path of the program or shorter than the total amount of work to be performed divided by the number of EUs available. Therefore, a lower bound for the execution time is given by:

$$T_{lowest} = \max(h_{start}, \frac{\sum_{i \in V} c_i^L}{N})$$

where h_{start}, the height of the start node is the length of the program critical path, and the sum of the local cost c_i^L of all nodes is the total amount of work to be performed by the program.

We use the definition of *absolute efficiency* from [25],

$$E = T_{lowest}/T_{end}$$

where T_{end} is the starting time of the v_{end} node in the graph, i.e., the execution time for the program with the thread partition produced by the algorithm running under an efficient FIFO scheduler. $E = 100\%$ means that the partition algorithm found a partition that can result in the optimal execution time.

To measure the efficiency of the algorithm, we varied the percentage of remote nodes in the randomly generated DDG from 25% to 75%, and the size of the graph from 10 nodes to 1000 nodes. Each graph has three times as many edges as the number of nodes. Then we generated twenty distinct random DDGs and applied the partition algorithm to each one of

Machine	% Remote Edges	Nodes						
		10	20	50	100	200	500	1000
EARTH-MANNA-DUAL	25%	87.6	91.8	90.0	83.6	82.3	77.0	78.1
	50%	96.3	95.8	96.8	93.7	83.8	71.6	71.2
	75%	96.5	98.7	98.9	98.2	95.6	69.5	66.0
EARTH-MANNA-SPN	25%	95.2	98.2	95.9	94.2	91.5	87.1	88.6
	50%	95.7	97.6	98.7	97.7	97.8	88.5	83.7
	75%	99.3	99.8	99.9	99.8	99.8	99.3	80.3
EARTH-SU	25%	95.3	98.3	92.4	93.2	96.7	99.9	100.0
	50%	95.9	95.4	99.2	97.0	92.0	92.8	98.1
	75%	98.4	99.3	98.4	97.8	98.5	90.3	94.1
SMP	25%	99.2	99.7	99.8	99.9	99.3	99.9	99.9
	50%	99.7	99.8	99.9	99.9	99.9	99.9	100.0
	75%	99.8	99.9	99.9	99.9	99.9	100.0	100.0
SCMP	25%	99.2	99.8	99.8	99.9	99.3	99.9	99.9
	50%	99.7	99.8	99.9	99.9	99.9	99.9	100.0
	75%	99.7	99.9	99.9	99.9	99.9	100.0	100.0

Table 2: Average partition algorithm efficiency measured by equation 6.4 for the architectures defined in Section 6.3.

them. We computed the average of the execution time for each run and compared with the absolute optimal that could be obtained for that DDG by an *oracle* as defined by equation 6.4, and present the average efficiency in Table 2.

Table 2 presents the algorithm efficiency which is a percentage of the absolute optimal as defined in equation 6.4. As the results in the table demonstrate, the algorithm did remarkably well for machines with four or eight execution units per processing node (SMP and SCMP). The algorithm also did quite well both for the EARTH-SU that has a single EU and a very low thread switching cost and for the EARTH-MANNA-SPN that has two EUs per processing node. The results for graphs with a large number of nodes for the EARTH-MANNA-DUAL are not as good. This should not come as a surprise because this architecture has a single EU and very large thread switching costs.

6.5 Effectiveness of Searching Heuristics

In this section we present data that evidences the effectiveness of the algorithm that searches through the schedule to find the earliest gap in which a node can be scheduled without conflicting with already scheduled nodes. Table 3 presents the average total number of calls to the TestGap algorithm for the experiments with DDGs in which 50% of the edges are remote. Each entry in the table is the average over the partitioning of 20 DDGs. The last row of the table is an average over all the machines. As the results indicate, the number of calls to TestGap, which is a direct indication of the actual algorithm complexity, is super-linear but sub-square. Our estimation is that the average algorithm complexity is close to $O(n^1.7)$.

Machine	Nodes						
	10	20	50	100	200	500	1000
EARTH-MANNA-DUAL	16.1	37.9	157.4	773.2	5526.1	45627.9	193916.5
EARTH-MANNA-SPN	14.3	35.0	110.0	447.0	2606.8	36761.8	182164.3
EARTH-SU	16.4	38.5	155.8	739.6	4314.5	48071.0	203341.6
SMP	11.0	22.4	65.6	152.8	486.7	7086.0	69870.4
SCMP	11.0	21.8	65.6	152.8	489.0	7260.9	68633.2
Average	13.8	31.1	110.9	453.1	2704.6	28961.5	143585.2

Table 3: Average number of calls to the TestGap algorithm for the architectures defined in Section 6.3, for DDGs with 50% of remote edges.

Table 4 presents the average number of events visited in the list of events in each call to the TestGap algorithm. For comparison purpose we implemented a competing procedure that visits all the events in the schedule that happen before the starting of the gap The results of this alternative algorithm are presented as "Other" in the table. As the results evidence, the use of a sorted list significantly reduces the complexity of the procedure to identify the earliest viable gap in the schedule that can accommodate the node to be scheduled.

Machine	Algorithm	Nodes						
		10	20	50	100	200	500	1000
EARTH-MANNA-DUAL	TestGap	0.63	0.81	0.94	1.10	1.09	1.03	1.02
	Other	2.86	3.66	6.19	8.82	9.27	6.61	7.58
EARTH-MANNA-SPN	TestGap	1.47	1.68	1.82	2.11	2.18	2.13	2.07
	Other	5.87	7.64	11.84	13.68	16.97	25.36	30.53
EARTH-SU	TestGap	0.67	0.83	0.96	1.09	1.11	1.05	1.02
	Other	2.74	4.00	6.43	8.30	11.72	11.13	10.02
SMP	TestGap	2.62	4.17	5.75	6.34	7.65	8.57	8.56
	Other	9.66	16.28	25.91	34.93	42.97	47.96	132.29
SCMP	TestGap	2.62	4.14	5.82	6.34	7.59	8.58	8.60
	Other	9.66	16.62	26.43	34.93	42.40	52.77	133.88
Average	TestGap	1.60	2.33	3.06	3.40	3.92	4.27	4.25
	Other	6.16	9.64	15.36	20.13	24.67	28.77	62.86

Table 4: Average number of events visited in the event list by the TestGap algorithm and by another algorithm that searches all events that happen before the ready time of the node to be inserted in a thread (DDGs with 50% of remote edges).

6.6 Latency Tolerance Capacity

Figure 10 shows the distribution of the nodes in the random graphs generated. In our experiments we varied the percentage of remote nodes in the graph between 25% and 75%. As indicated in the figure, the cost for a remote operation was varied between 400 and 4000 cycles while the local cost was distributed around 10 cycles. Thus, a very wide range of latencies were tested, and the results presented in Table 2 indicates that the partition algorithm produces thread partitions that are able to tolerate varied latencies.

7 Related Work

The thread partition problem for multithreaded architectures is similar to the task partitioning and scheduling problem [1, 7, 20, 29]. In both problems a program has to be divided into smaller pieces with respect to some constraints (dependencies). The focus of task partition is to allocate N tasks onto M processors in order to reduce the total execution time. Usually this problem can be represented as a graph partition problem in which nodes denote tasks and edges represent constraints. One type of constraint is the *precedent* constraint, i.e., one task must complete before another task can start. Another type of constraint is the *communication* constraint, i.e., how much data is transmitted between two tasks. When the *communication* constraint is taken into consideration the task partitioning problem is an NP-complete problem [5]. Therefore, the optimization goal is often reduced to minimize total communication volume in order to indirectly reduce the total execution time [20].

Although scant work on thread partitioning for fine grain multithreading architectures has been published, the literature is prolific in studies of related problems such as task partitioning and scheduling [1, 7, 20, 3, 6], task sequencing [29], and instruction reordering [21].

In the area of functional languages, solutions were proposed both for non-strict (lenient) languages such as Id [16, 17], and for strict languages such as Sisal [15]. These partitioning algorithms are based on a program intermediate representations called *structured dataflow graph* [27] that consist of a collection of *basic blocks* and *interfaces*. *Dependence set partitioning* algorithms find a correct partitioning by merging nodes together with the same dependence set [12]. The *dependence set* of a node is the union over the input of all its predecessors. *Demand set partitioning* works similarly to dependence set partitioning, but finds a correct partitioning by grouping together nodes with the same demand set [10, 23]. The *demand set* of a node is the union over the output of all successor nodes. The combination of dependence set partitioning with demand set partitioning, together with global analysis, is called *iterated partitioning* [28, 4]. It combines the power of both partitioning algorithms by applying them iteratively. Moreover, larger threads can be generated by applying *separation constraint partitioning* plus inter-procedural analysis [22]. For a review of those algorithms, readers are referred to [24].

In compiling for strict languages, top-down and bottom-up approaches are proposed in [2, 18]. The idea is to merge two clusters A and B if either A's output is the superset of B's input or A's input is the subset of B's output. Please refer to [19] for details.

8 Conclusion

We designed, implemented and evaluated an efficient, effective, and robust algorithm to partition a program into threads for the case in which multiple execution units are available in each processing node of a parallel architecture. The algorithm is efficient because it generates a partition that results in an execution time that is very close to the best possible execution time determined by the length of the critical path and the total amount of computation existing in the program. The algorithm is robust because it worked efficiently for a varied set of architectures and a wide range of latencies between processing node. The algorithm is effective because it employs a data structure associated with a searching algorithm that reduce the time complexity of the algorithms. On our experimental framework we tested the algorithm with several thousand data dependency graphs with up to a thousand nodes and several thousand connections.

Acknowledgements

We would like to thank Andres Marquez, Arthour Stoutchinin, Gagan Agarwal and Kevin Theobald for the productive discussions in our seminars, and Mark Butala for the valuable help typing/formating tables. We acknowledge the support of DARPA, NSA and NASA through a subcontract with JPL/Caltech. The current EARTH research is partly funded by the NSF.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] Gerd Heber, Rupak Biswas, and Guang R. Gao. A new approach to parallel dynamic partitioning for adaptive unstructured meshes. In *Proceedings of the 13th International Parallel Processing Symposium and the 10th Symposium on Parallel and Distributed Processing*, pages 360–364, San Juan, Puerto Rico, April 12–16, 1999. IEEE Computer Society and ACM SIGARCH.

[9] Laurie J. Hendren, Guang R. Gao, Xinan Tang, Yingchun Zhu, Xun Xue, Haiying Cai, and Pierre Ouellet. Compiling C for the EARTH multithreaded architecture. ACAPS Technical Memo 101, School of Computer Science, McGill University, Montréal, Québec, March 1996. In ftp://ftp-acaps.cs.mcgill.ca/pub/doc/memos.

[10] J. E. Hoch, D. M. Davenport, V. G. Grafe, and K. M. Steele. Compile-time partitioning of a non-strict language into sequential threads. In *Proceedings of the Fifth IEEE Symposium on Parallel and Distributed Processing*, Dallas, Texas, December 1–4 1993.

[11] 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.

[12] 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.

[13] Ashfaq A. Khokhar, Gerd Heber, Parimala Thulasiraman, and Guang R. Gao. Load adaptive algorithms and implementations for the 2D discrete wavelet transform on fine-grain multithreaded architectures. In *Proceedings of the 13th International Parallel Processing Symposium and the 10th Symposium on Parallel and Distributed Processing*, pages 458–462, San Juan, Puerto Rico, April 12–16, 1999. IEEE Computer Society and ACM SIGARCH.

[14] Olivier Maquelin, Guang R. Gao, Herbert H. J. Hum, Kevin B. Theobald, and Xin-Min Tian. Polling Watchdog: Combining polling and interrupts for efficient message handling. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture*, pages 178–188, Philadelphia, Pennsylvania, May 22–24, 1996. ACM SIGARCH and IEEE Computer Society. *Computer Architecture News,* 24(2), May 1996.

[15] J. R. McGraw et al. SISAL: Streams and iteration in a single assignment language—language reference manual version 1.2. Technical Report M-146, Lawrence Livermore National Laboratory, 1985.

[16] R. S. Nikhil. Id (Version 88.0) reference manual. CSG Memo 284, Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, Massachusetts, March 1988.

[17] Rishiyur S. Nikhil and Arvind. Id: a language with implicit parallelism. CSG Memo 305, Computation Structures Group, MIT Laboratory for Computer Science, Cambridge, Massachusetts, February 1990.

[18] 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.

[19] 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.

[20] 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).

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] Xinan Tang, Jian Wang, Kevin B. Theobald, and Guang R. Gao. Thread partitioning and scheduling based on cost model. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 272–281, Newport, Rhode Island, June 22–25, 1997. SIGACT/SIGARCH and EATCS.

[26] Kevin Bryan Theobald. *EARTH: An Efficient Architecture for Running Threads.* PhD thesis, McGill University, Montréal, Québec, May 1999.

[27] Kenneth R. Traub. A compiler for the MIT tagged-token dataflow architecture. Technical Report MIT/LCS/TR-370, MIT Laboratory for Computer Science, Cambridge, Massachusetts, August 1986. Master's thesis.

[28] Kenneth R. Traub, David E. Culler, and Klaus E. Schauser. Global analysis for partitioning non-strict programs into sequential threads. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, pages 324–334, San Francisco, California, June 22–24, 1992. ACM SIGPLAN, SIGACT, and SIGART. *Lisp Pointers* (5)1, January–March, 1992.

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