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

Automatically Partitioning Threads Based on Remote Paths

Xinan Tang
Guang R. Gao

CAPSL Technical Memo 23
July 20, 1998

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

In order to program multithreaded architectures effectively compiler support to automatically partition programs into threads is essential. This paper proposes a *remote-path* based thread partitioning framework, which can generate low-level threads from procedural programs automatically. The framework has been implemented in the EARTH-C compiler, which uses Data Dependence Graph (DDG) as an intermediate representation for thread partitioning. To make the compiler work fast, a practical $O(n^2)$ algorithm is designed to build *non-redundant* DDG. To generate correct and efficient threaded code, the remote-path heuristic is employed to satisfy thread partitioning constraints and schedule threads to run fast. The experimental results show that the DDG building algorithm is fast and the remote path based heuristic is very effective in partitioning programs into "optimized" threads.

KEYWORDS: data dependency graph, thread partition, multithreaded compilers, parallelizing compilers, parallel programming environments.

Contents

List of Figures

List of Tables

1 Introduction

Multithreaded architectures have become increasingly popular because of the detrimental impact of long communication and synchronization latencies on the performance of high-end computing systems. By supporting many threads of control and fast switching among threads, multithreaded architectures can tolerate inherent communication and synchronization latencies by switching to the next ready thread whenever a long-latency operation is encountered [3, 23, 11, 2, 8, 18, 16, 26, 37, 24]. As long as there is enough parallelism in an application, a multithreaded architecture can hide these latencies and utilize available communication bandwidth effectively. Since the amount of parallelism exposed to the fine-grain multithreaded architectures may be enormous, it is a demanding task for a programmer to manage available parallelism well to hide latencies. Therefore it is essential for the compiler to automatically partition a program into threads if multithreaded architectures can be programmed effectively.

In addition to traditional compiling issues, compiler writers for multithreaded machines face a 'new' challenge: *how to partition a program into threads so that the underlying multithreaded features can be exploited?* Although this problem may look similar to the task partitioning (scheduling) problem [1, 13, 38] in the sense that a program has to be divided into smaller pieces with respect to data dependencies, it poses different optimization goal, such as using parallelism to hide communication latencies rather than reducing communication volumes. Furthermore, the required constraints of putting two operations into one thread is more stringent than those of putting two tasks onto one processor. This problem becomes even more demanding for *non-preemptive* multithreaded models since the *split-phase* constraint must be obeyed on these machines.

In contrast to *preemptive (blocking)* thread models, a *non-preemptive* (non-blocking) thread once starts will run to completion without suspension. To support *non-preemptive* execution within a thread, before a thread can start, all data needed by that thread must be ready and presented in the processor. Therefore remote operations such as a load from remote memory must be handled by using *split-phase* transactions, i.e., it is performed in two phases: 1) the load operation itself is initiated (issued) in one thread; 2) other operations that will use the returned values must wait in another thread and be notified upon the completion of the remote load operation. The second thread is not started until the remote read operation finishes. In order to produce correct partition, it is therefore the compiler's responsibility to identify *split-phase* operations and respect the constraint by placing the related operations into two different threads.

Some research work on thread partitioning has been carried out within the functional language community [36, 11, 31, 26]. Comparatively, there is little attention to the issues of partitioning threads for procedural languages. With the coming of MTA TERA machines into the market [3], we believe that there will be more and more demand for including the automatic thread partitioner within the multithreaded compiler to make the multithreaded machine available to the general public.

This paper addresses the fundamental problem in compiling programs written in procedural languages for multithreaded machines: how to partition threads from a procedural program automatically? We have developed a thread partitioning framework based on the *remote path* heuristic to generate threads from user programs automatically.

In addition, we also studied the issues of building such a compiler efficiently. Since data dependence graph is a widely used data structure in thread partition, we designed an efficient algorithm to compute a *non-redundant* data dependence graph (DDG), which only retains the minimum number of dependency edges, to reduce the DDG traversal cost.

We have done experiments within our compiler framework, and the main results related to this paper are as follows:

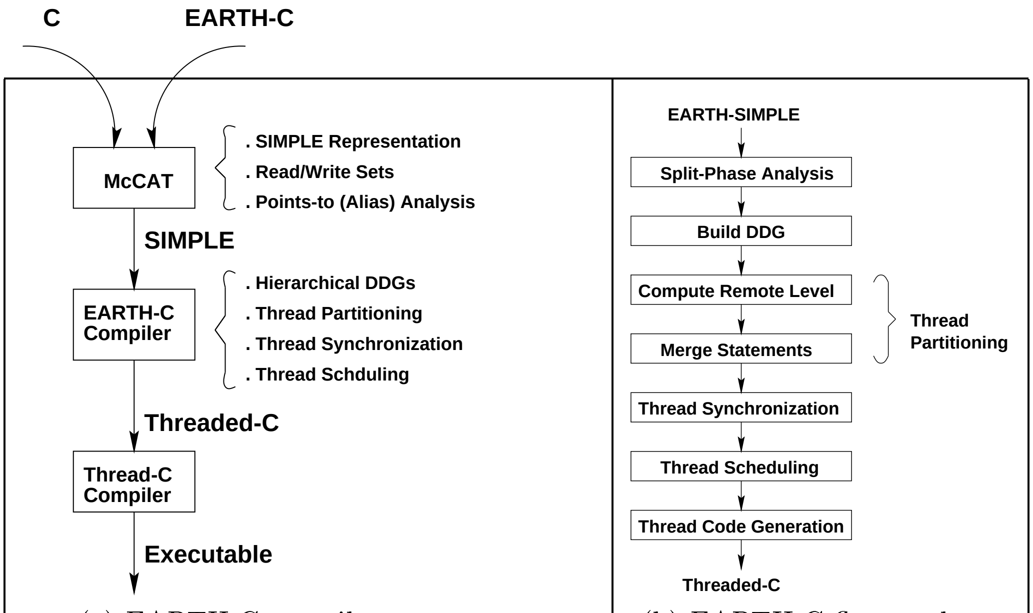
- a remote path based partition framework, in which 1) *remote-level* is used to generate correct threaded code (see Section 5.2); 2) *remote-height* is used to produce fast threaded code (see Section 5.3);
- an *incrementally updated ancestor-sets* based algorithm to build a *non-redundant* DDG to reduce the DDG traversal cost. The amortized time complexity of the algorithm is $O(n^2)$ in practice.

We believe that our method is very general. It can be applied to any compiler which supports non-preemptive multithreaded architectures without modification; it can also be applied to preemptive multithreaded models with slight modification.

The remainder of the paper is organized as follows. In Section 2, we summarize the EARTH-C compiler environment and describe the flow of thread generation analyses. In Section 3, we formulate the thread partition problem and discuss the impact of such an formulation. In Section 4, we introduce a sparse DDG representation which does not contain *redundant* edges and propose an efficient algorithm to compute DDG in quadratic time in practice. In Section 5, we present the remote path based partitioning algorithm, focusing on the correctness and efficiency of thread partitioning. In Section 6, we analyze our experimental data on a suite of irregular applications. In Section 7, we review related work and discuss the generality of our approach. Finally, we present conclusions and future work in Section 8.

2 EARTH-C Compiler Environment

EARTH-C is a parallel C dialect with simple extensions to express: *control parallelism*, *data locality*, and *collective communication*[15, 34]. The EARTH-C compiler takes either C or EARTH-C programs as input, performs thread partitioning based on data-flow analyses such as side-effect (read-write) and points-to (alias) analysis provided by the McCAT compiler [14], and generates the Threaded-C code as output [16, 21]. The Threaded-C code contains lower-level thread primitives. The final executable code, which runs on the MANNA parallel architecture [6], is compiled by the Portland `pgcc` compiler by linking with the EARTH threaded runtime library. The component structure of the EARTH-C compiler is shown in Figure 1(a). In this section, we describe the core part of the EARTH-C compiler: the EARTH-C thread generator.



(a) EARTH-C compiler components (b) EARTH-C flow analyses

Figure 1: EARTH-C Compiler Environment

2.1 Thread Generation in the EARTH-C Compiler

As shown in Figure 1(b), the input to the EARTH-C thread generator (partitioner) is an EARTH-SIMPLE representation, which is based on McCAT SIMPLE representation [32] and is simplified further such that each assignment statement contains at most one *remote* variable. A *remote* variable is the one which may access memory of another processor. For instance, global variables, and references through non-local pointers are all remote variables.

Based on the EARTH-SIMPLE representation, the thread generator performs *split-phase* analysis, identifying whether a statement involves a split-phase operation or not. We assign a type for each statement. If a statement may contain a split-phase operation, its type is *remote*; otherwise *local*. A *remote* statement contains a remote variable; a *local* does not have any remote variable. This type information is fundamental and important. It will be used in the whole process of thread partitioning.

The thread generator walks on data dependence graphs to partition threads. DDGs for a program are built in a hierarchical fashion. Compound statements like conditionals or loops are treated as one node within their own hierarchical level. Going into a compound statement, another layer of DDG is exposed. We can label two types of edges in a DDG. Edges emitting from a remote node are called *remote*; edges emitting from a local node called *local*. In order to satisfy the split-phase (remote) constraint introduced by split-phase transactions, the thread partitioner needs to "cut" all remote edges to separate the nodes incident on the remote edges.

Once a DDG is built, the thread partitioner first computes the *remote* level for each node in the DDG. The remote level of a node is the maximum number of remote nodes along any path from the root to the node itself. The remote level of a node represents the minimum number of thread switching times from the root to the current node. Merging nodes with the same remote level into one thread guarantees that the remote edges are cut and thus the remote constraints introduced by split-phase transactions are satisfied.

Once the thread boundaries of threads are known, we can build *Thread Data Dependence Graph* (TDDG) to compute the synchronization relation among threads. In a TDDG, each node represents a thread, which consists of all statements belong to this thread. An edge exists between two TDDG nodes T_i and T_j if and only if there is an edge between two statements s_i and s_j where s_i is in T_i and s_j is in T_j. The number of *remote* edges across two TDDG nodes is the number of synchronization signals sent between the two nodes at runtime.

Even though threads have been formed, the execution order within a thread has not been specified yet. Thread scheduling is a step to arrange the execution order within a thread and make code generated run faster (like instruction scheduling). We will describe a *remote-height* based scheduling algorithm in section 5.3 to enhance the performance of scheduled threaded code.

Finally, the statements marked as remote are translated to lower-level *split-phase* Read (Write) operations and thread boundaries are created by inserting the thread creation and synchronization primitives.

2.2 Thread Partition Example

In this section, we use a simple example shown in Figure 2 to illustrate the thread partitioning process.

```
int treeadd(Tree *r)
{
  int s1, s2;

  if (r==NULL)
    return (0);
  else {
    s1 = treeadd(r->left);
    s2 = treeadd(r->right);
    s3 = r->value;
    return (s1 + s2 + s3);
  }
}
```

(a) C source code

```
int treeadd(Tree *r)
{ int s1, s2;
  Tree *lptr, *rptr;
  if (r==NULL)
    return (0);
  else {
    lptr = r->left;
    rptr = r->right;
    s3   = r->value; -----------|
    END_THREAD();               |
THREAD_1:                       |
    s1 = TOKEN(treeadd,lptr);   |
    s2 = TOKEN(treeadd,rptr);   |
    END_THREAD();               |
THREAD_2:       <---------------|
    return (s1 + s2 + s3);
  }
}
```

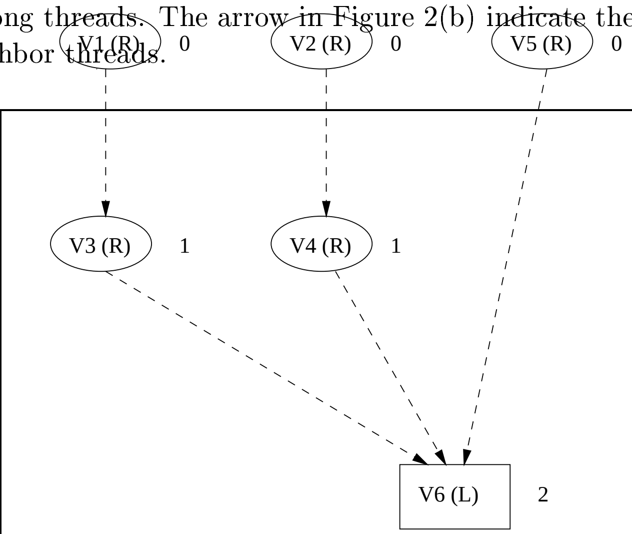
(b) Optimal Threaded-C code

Figure 2: Example: traverse a tree in postorder

Figure 2(a) lists a C program, which traverses a binary tree in postorder to add up the value of whole tree nodes. Figure 2(b) lists corresponding *pseudo* Threaded-C code, which parallelizes the C code by invoking two recursive calls simultaneously. Each thread is started by a label THREAD_ (except the first one) and ended with the END_THREAD primitive, which means terminating execution of the current thread and switching to another ready thread. The TOKEN primitive in Threaded-C is like UNIX *fork*, which spawns off a child thread while continuing its own execution. Usually, the thread after an END_THREAD has to wait until all remote operations in the previous thread have been finished. However, EARTH allows any arbitrary synchronization relation among threads. The arrow in Figure 2(b) indicate there is a synchronization between two non-neighbor threads.

```
S1: lptr = r->left;
S2: rptr = r->right;
S3: s1   = treeadd(lptr);
S4: s2   = treeadd(rptr);
S5: s3   = r->value;
S6: return (s1 + s2 + s3);
```

(a). Simplified Statements



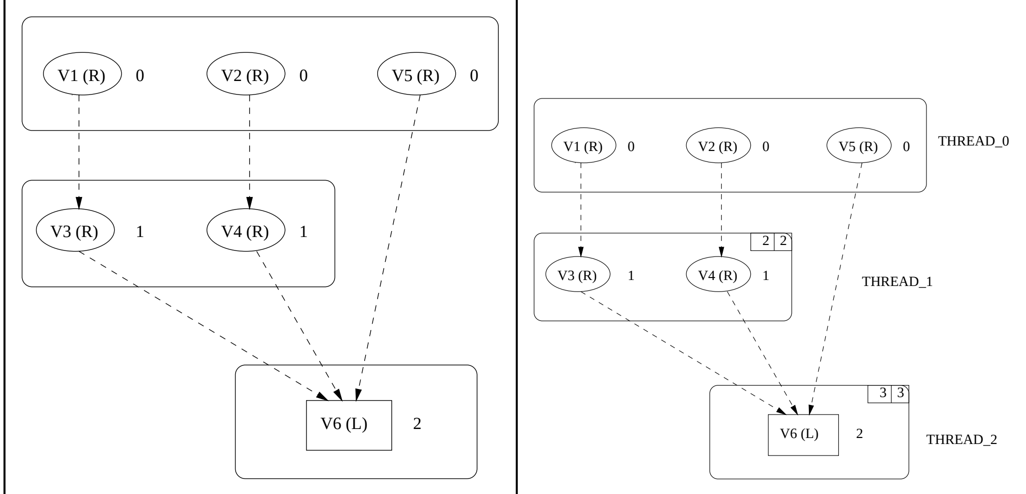
(b). An annotated DDG

Figure 3: Simplifying statements and building DDG

Let's take a close look at the *else* branch, which is the critical path of the program execution. First, statements are simplified so that in each statement there is only one remote operation involved. Assume the tree is distributed among processors and a child of the tree rooted from `r` may point to a subtree located in another processor. Therefore fetching the left or right child of the tree is a remote operation. Both function calls are simplified to expose such remote (split-phase) operations clearly. This is shown in Figure 3(a) at statement s_1 and s_2. Based on the tree distribution assumption, the *split-phase* analysis will assign each node a type. In Figure 3(a), all statements are *remote* except the last one since this statement does not access the distributed tree.

Then, we can build a DDG for the statement sequence listed in Figure 3(a), which is shown in Figure 3(b). Moreover, the DDG drawn in Figure 3(b) is annotated with types (L/R inside nodes) and remote level (numbers right beside nodes). For example, node v_5 (s_5) is a remote node and its remote level is 0. Node v_6 (s_6) is a local node with remote level 2. The dashed edges represent remote edges and all edges in Figure 3(b) are remote.

Thread partitioning can be done by merging all nodes with the same remote level together into one thread. Figure 4(a) lists the partitioning result, which has three threads. Figure 4(b) gives a partitioned thread graph, which can be used to compute the synchronization relation among threads. For example, there are two remote edges between thread 1 and thread 2. Therefore thread 2 has to wait for two signals sent from thread 1. The third signal to thread 2 is sent from thread 0. In EARTH, each thread is protected by a sync slot ($sync\_count, reset\_count$)



(a). Partitioned Graph (b). Compute Sync Signals

Figure 4: Thread formation and synchronization

to guarantee that only after receiving all signals (sync_count is reduced to 0), the associated thread becomes eligible to start.

```
THREAD_0:
S1: lptr = r->left;
S2: rptr = r->right;
S5: s3   = r->value;
    END_THREAD();

S3: s1   = treeadd(lptr);
S4: s2   = treeadd(rptr);
    END_THREAD();

S6: return (s1 + s2 + s3);
```

(a) Schedule 1

```
THREAD_0:
S5: s3   = r->value;
S1: lptr = r->left;
S2: rptr = r->right;
    END_THREAD();

S3: s1   = treeadd(lptr);
S4: s2   = treeadd(rptr);
    END_THREAD();

S6: return (s1 + s2 + s3);
```

(b) Schedule 2

Figure 5: Thread scheduling

Now, it is time to schedule the nodes within a thread. Thread scheduling is to decide the sequential execution order within a thread. Figure 5 lists two schedules for thread 0. The first issues statement s_5 at last and the second issues it at the beginning. We will show in Section 5 that the first schedule is better since the remote heights of v_1 and v_2 are greater than that of v_5. Therefore, the corresponding statements s_1 and s_2 should be issued first.

3 Problem Statement

Since the thread generator forms the core in a multithreaded compiler, it is worth formulating the thread partition problem to investigate it thoroughly. We first describe a non-preemptive multithreaded architecture model, and then formulate the thread partition problem based on such a model.

3.1 Thread Execution Model

A thread is a sequence of operations which run to completion without interruption. Similar to the macro-dataflow model [28], threads can synchronize with each other through explicitly synchronization or communication. More specifically,

1. Within a thread:
   - Split-phase operations: a *long-latency* (*remote*) operation is performed in the split-phase manner, i.e., if an operation is issued in one thread then the operation waiting for the returned values must *not* be in the same thread.
   - Sequential sequence: a sequence of operations within a thread are sequentially issued or executed to completion without interruption.
2. Among threads:
   - Data availability: a thread may begin execution until all its data available, i.e., data being placed in the local processor.
   - Processor availability: a ready thread is allowed to be executed only if the processor is free.

Unlike the micro-dataflow model, remote communication operations can start in between the execution of a thread since these operations are now performed in the split-phase manner. Since most multithreaded machines require extra cycles to switch from one thread to another, we use δ to denote this cost.

In the EARTH model [17, 16], 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 communication operation is first issued by the EU, then dispatched to the SU. It is the job of SU to finish this split-phase operation, including network communication and thread synchronization. Even though our execution model is based on the EARTH model, it is not restricted to EARTH and can be applied to any architecture that supports the split-phase execution model.

3.2 Problem Formulation

We will use capital letters to represent sets and small letters to represent elements in this section.

We assume the input program has been translated into a hierarchical DDG, $DDG = (V, E)$, which is defined as follows:

- A set of nodes $V = \{v_1, \ldots, v_n\}(|V| = n)$.
- A set of edges (v_i, v_j) representing *precedence constraints* imposed by data dependencies. For convenience, we add two special nodes: *start* and *end* nodes, denoted by v_s and v_e respectively. The *start* node is the only node in G which has no predecessors and the *end* node is the only one which has no successors.

Based on the previous discussions, we can see that the thread partition problem can be abstracted as:

- A graph partition problem, in which nodes are divided into a set of non-intersecting groups with respect to the remote constraints.
- A scheduling problem in which the execution order within a thread must be specified in such a way that execution time is minimized.

Formally, a *thread* T_k is a non-empty subset of V with a total order $\prec$ defined over the nodes that belong to T_k. For convenience, if a vertex v_i belongs to thread T_k ($v_i \in T_k$) we say that $T(v_i) = T_k$. Let $\mathcal{P}$ be a set of threads $T_1 \ldots T_k$ with a partial order $\sqsubset$ defined over the threads. A *legal* partition $\mathcal{P}$ is defined as:

- $\mathcal{P}$ is a partition of V;
- C1: $\forall$local edge $(v_i, v_j) \in E \Longrightarrow (T(v_i) = T(v_j) \wedge v_i \prec v_j) \vee (T(v_i) \sqsubset T(v_j))$;
- C2: $\forall$remote edge$(v_i, v_j) \in E \Longrightarrow (T(v_i) \neq T(v_j)) \wedge (T(v_i) \sqsubset T(v_j))$

The first constraint $C1$ ensures *local* data dependencies, and the second $C2$ enforces the *remote 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 two separate threads ($T(v_i) \neq T(v_j)$), and v_i's thread must execute first ($T(v_i) \sqsubset T(v_j)$). If the edge from v_i to v_j is local, then v_i must execute before v_j. If these nodes are in the same thread, then v_i must come before v_j in that thread ($v_i \prec v_j$). If these nodes are in different threads, then v_i's thread must execute first.

Assume data and task partition have been done in advance so that the type of each node (local or remote) is known at compile time. The thread partition problem studied in this paper can then be formulated as:

Given a DDG, determine a partition $\mathcal{P}$, which satisfies constraints $C1$ and $C2$, and minimizes the total number of threads generated.

Please note that the optimization goal of this formulation is *not* directly to minimize total execution time because such a formulation is NP-*hard* and it needs a more advanced program cost model to derive efficient heuristic algorithms [35, 33].

By reducing the number of threads generated, we try to minimize the thread switching cost (δ), which may *indirectly* reduce total execution time. The advantage of such a formulation is that it does not need to know the exact execution time of each node. Type information computed at compile time provides good hints to partition a program and produce efficient code. Furthermore, an optimal solution can be found for the above problem formulation (see Section 5).

4 Algorithms to Build Sparse DDGs

Because a DDG is traversed many times during the partitioning process, designing an efficient DDG representation is a rewarding effort to improve the speed of thread partition algorithms. In this section, we describe the *non-redundant* DDG representation, compare two algorithms used to build such a DDG, and analyze the time complexities of the two algorithms.

4.1 Non-redundant DDG

To reduce the DDG traversal cost, unnecessary DDG edges should be eliminated since the number of nodes cannot be changed. An edge (v_i, v_j) is *redundant* if there exists a path from v_i to v_j, i.e., v_i, v_1, ..., v_k, v_j. The idea behind this definition is that if there exists a transitive-closure data dependence between two nodes v_i and v_j, then the direct edge becomes an unnecessary constraint since it is automatically satisfied. For example, assume node v_k is data dependent on nodes v_j and v_i, and v_j is dependent on v_i. Then, edge (v_i, v_k) is redundant.

The non-redundant DDG is used in our compiler. In the following we will discuss data structures and algorithms to represent and build such DDG.

There are many ways to represent a graph. Adjacency matrix and adjacency-list are two commonly used presentations [9]. In our previous work, the adjacency matrix was selected to represent a DDG since it is easy for implementation. The main drawback of using adjacency matrix on sparse graphs is that it wastes too much space ($O(n^2)$). The EARTH-C compiler profiling results [34] show that DDGs derived from irregular applications are usually sparse. This motivates us to implement the thread partitioner based on adjacency-list representation.

Since an EARTH-C program is represented as a hierarchical DDG, on each hierarchical level the corresponding DDG is a *directed acyclic graph* (DAG). Therefore, the level of a node can be defined as the maximum distance from the root to the node itself. Because level information

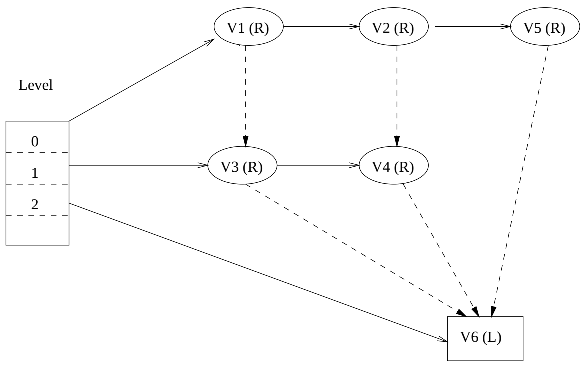


Figure 6: Level-directed DDG

is very useful in guiding the top-down (bottom-up) search of graph, we also include it in our graph representation.

Figure 6 is a *level-directed* DDG derived from Figure 3(b). Nodes in the same DAG level form a linked list, which is connected by solid lines in Figure 6. The actual adjacency-list of a node, representing its *successors* or *predecessors*, is represented as original edges in order to show the level structure clearly.

4.2 Building Non-redundant DDG

Figure 7 lists two algorithms that build a non-redundant DDG from scratch. Please note that our approach is to eliminate unnecessary edges during the graph building phase rather than running transitive reduction algorithms after a DDG has been built.

Assume first $i-1$ nodes have been inserted into a DDG, at step i, procedure $insert\_node\_into\_DDG()$ is called to insert s_i into the DDG.

The algorithm listed in 7(a) proceeds as follows:

First, marking all DDG nodes as *unvisited.* Then, start to search the DDG from bottom to top. The node's level is used to direct this bottom-up search.

If a unvisited node n is selected to visit, we first check whether there exists a data dependence between n and s_i. If a dependence exists, s_i is added at the next level below n, and corresponding edges are also inserted. Then, we mark all ancestors of n as visited so that any ancestor of n will not be visited in the next iteration of searching (rejected by IF statement at L2).

Since the FOR loop (at L1) will visit n DDG nodes and the most expensive operation within the loop is at L3, which takes $O(n+e)$ to mark all ancestors. Therefore the time complexity of this ancestor-marking based algorithm is $O(n(n+e))$, i.e., $O(n^3)$ in the worst case.

Figure 7(b) gives an improved version, in which the ancestor set of a node is *incrementally* updated. The main idea of the ancestor-set based algorithm is to:

```
    Input:    ddg and statement node s_i;
    Output:   non-redundant ddg with s_i inserted;
    Procedure insert_node_into_DDG(ddg, s_i)
    {
L0:   mark all nodes in ddg as unvisited;
L1:   FOR each node n in ddg, searching in bottom-up order
      /* i.e. visit nodes at the greatest level first */
      {
L2:     IF not visited(n) THEN
           IF dependence(n, s_i)== true THEN  {
              IF not is_in_ddg(s_i, ddg) THEN
                insert s_i at level(n) + 1;
              insert edges between s_i and n;
L3:           mark all ancestors of n as visited;
           }
      }
    }
```

(a)Algorithm marking ancestors visited

```
    Input:    ddg and statement node s_i;
    Output:   non-redundant ddg with s_i inserted;
    Procedure insert_node_into_DDG(ddg, s_i)
    {
L0:   ancestors(s_i) = empty;
L1:   FOR each node n in ddg, searching in bottom-up order
      /* i.e. visit nodes at the greatest level first */
      {
L2:     IF not is_ancestor(n, s_i) THEN
           IF dependence(n, s_i)==true THEN {
              IF not is_in_ddg(s_i, ddg) THEN
                insert s_i at level(n) + 1;
              insert edges between s_i and n;
L3:           ancestors(s_i) += ancestors(n) + {n};
           }
      }
    }
```

(b) Algorithm updating ancestor sets incrementally

Figure 7: Algorithms of building non-redundant DDG

- use an ancestor set to remember partial ancestors;
- update the set incrementally.

Thus, instead of marking whole ancestors as visited at L3, ancestors of n and n itself is added into the ancestor set of s_i incrementally. The time complexity of the ancestor-sets based algorithm depends on that of set union (+) and member (is_ancestor) operations. Given m MAKE-SET, UNION and FIND-SET operations, in which the number of MAKE-SET operations is k, the amortized time complexity of such a sequence is $O(m\alpha(m,k))$ [9], where α is the inverse of Ackermann's function. During each FOR-loop iteration in Figure 7(b), it invokes maximum $m = 3n$ set operations and creates $k = n$ sets. Therefore, the amortized time complexity of the improved algorithm per iteration is $O(3n\alpha(3n,n))$.

The total time it takes is:

$$\sum_{i=1}^{n} 3i\alpha(3i,i) <= 3\alpha(3n,n)(\sum_{i=1}^{n} i) = O(n^2\alpha(3n,n))$$

In practice α can be treated as a constant and thus the improved algorithm has a quadratic time complexity in practice. The price we pay for the faster algorithm is that we have to allocate extra space to keep the ancestor set.

5 Thread Partition Based on Remote Path

Since a remote operation takes much longer time to finish than a simple local operation, it is highly likely that the remote operation lies on the *critical path* of program execution. This is one of the main reasons we weight the remote type information as the most valuable heuristic in our thread partition framework.

In this section, we will first give a formal definition of *remote path*. Then, we explain how its two components: *remote level* and *remote height* can be used in guiding the thread partitioning process.

5.1 Formal Definitions

A remote path of node v_i is a path from root v_s to *end* node v_e passing through v_i, and it includes *remote* nodes only, i.e, the length of a remote path is equal to the number of remote nodes along the path. The *remote path length* of v_i, RPL_i, is the maximum length of all its remote paths.

Because remote operations are very expensive, we have high confidence that if $RPL_i > RPL_j$ then the remote path passing through v_i is more likely to be in the critical path than that passing through v_j. Thus, more attention should be paid to partition and schedule well of nodes along the path passing through v_i.

However the remote path only provides a general direction in thread partition. Its two components: *remote level* and *remote height* are more convenient to use in practice.

A remote level of node v_i, RL_i, is the maximum number of nodes along a remote path from the *start* node v_s to node v_i. A remote height of node v_i, RH_i, is the maximum number of nodes along a remote path from node v_i to the *end* node v_e.

The formal definition is as follows:

$$\begin{cases} RL_s & = & 0 \\ RL_i & = & \max_{v_j \in pred(v_i)} (RL_j + 1 * remote(v_j)) \end{cases}$$

where $pred(v_i)$ is the set of v_i's predecessors and function $remote(v_j) = 1$ if v_j is remote;otherwise 0.

Similarly, remote height can be defined as:

$$\begin{cases} RH_e & = & 0 \\ RH_i & = & \max_{v_j \in succ(v_i)} (RH_j + 1 * remote(v_i)) \end{cases}$$

where $succ(v_i)$ is the set of v_i's successors and remote height takes the remote type of the node itself into account.

Table 1 lists the remote path metrics computed for the DDG shown in Figure 3(b). These metrics will be used to guide thread partition and schedule. Please note that *remote level* and *remote height* are computed according to the node's type and we do not need to know the execution cost of each node to compute these two metrics.

Remote	v_1	v_2	v_3	v_4	v_5	v_6
RL_i	0	0	1	1	0	2
RH_i	2	2	1	1	1	0

Table 1: Remote path metrics

5.2 Thread Partition Based on Remote Level

Remote level is an indicator of the maximum number of *split-phase* operations encountered from the *start* node to the node itself. Therefore, merging nodes with the same remote level will guarantee that the *remote* constraint is satisfied.

This can be proved as follows. Assume two nodes v_i and v_j have the same remote level and there exists an edge from v_i to v_j. If v_i is *local* then there is no remote constraint to be satisfied. On the other hand, if v_i were *remote*, then RL_j would be greater than RL_i by one according to the remote level definition. Therefore, it is contrary to the assumption $RL_i = RL_j$.

This proof shows that merging the nodes with the same remote level into one thread produces the *correct* partition with respect to the remote constraint.

In Figure 4(a), nodes v_1, v_2, and v_5 are all merged into thread 0 since their remote level is all 0.

We can also show that the number of threads generated is also *minimal* in terms of the formulation given in Section 3. First the minimum number of thread switching times is the same as the remote level of *end* node v_e, since one remote node will cause one thread switching. Second, the number of thread derived by the remote-level based algorithm is equal to the remote level of *end* node v_e plus 1, because one thread boundary splits two threads. The $RL(s_e) + 1$ is the minimal number of threads required to guarantee the split-phase constraint being obeyed. Thus, the remote-level based partitioning algorithm will derive the minimal threads.

5.3 Thread Scheduling Based on Remote Height

The above merging heuristic guarantees that the $C2$ type of constraints is satisfied.

For nodes within the same thread, the general thread scheduling strategy is to hoist them as high as possible. When a node v_i is moved up, the following scheduling strategies are applied:

1. If v_j has a local dependence with v_i then v_i cannot cross v_j. This guarantees that the $C1$ type of constraints is satisfied.
2. For nodes that have no data dependencies, sort them according to their types. The binary types used in Section 2 can be extended to multiple types, including $\{$*simple-remote, simple-local, compound-remote, compound-local, local-call*$\}$. The type order is their text order listed from left to right and it is arranged in such a way that remote operations such as represented by *simple-remote* type will be moved up and issued earlier.

3. If there is a type tie, use *remote height* as a heuristic to break the tie by putting the nodes with bigger remote height in front of other nodes with smaller ones.

The reason for using remote height to resolve the type tie is based on our believe that nodes with bigger remote heights are more likely to lie on the critical path than those with smaller ones.

In Figure 5, schedule 1 is better than schedule 2 because $RH_1 = RH_2 = 2$, which is greater than $RH_5 = 1$. Thus, node v_1 and v_2 should be put in front of v_5. Actually, the successors of v_1 and v_2 are two recursive function calls (v_3 and v_4). Thus, the operations of fetching their arguments represented by v_1 and v_2 should be executed as early as possible.

6 Experimental Results and Analysis

The *remote-path* based partition framework has been implemented in the EARTH-C compiler. To compare the two non-redundant graph building algorithms, they were also implemented. We report our experiments on a suit of irregular benchmarks using the multithreaded EARTH-MANNA emulator [6, 16]. The performance features of the EARTH-C compiler have been reported elsewhere [15, 34]. We list the major experimental results closely related to this paper.

- The *ancestor-sets* based algorithm is much faster than the *marking* based algorithm. Our experimental data verify our analytical results showed in Section 4.
- The remote height based scheduling algorithm improve execution performance for certain benchmarks.

In the following, we first introduce the benchmarks and the machine we used in our experiment. Then we report the timing comparison of two DDG building algorithms. At last, the impact of the remote height based thread scheduling algorithm is discussed.

6.1 Benchmarks

The benchmarks we used are from the Olden [7, 25] benchmarks: *treeadd*, *em3d*, *perimeter*, *power* and *mst*. All are irregular applications and pointer based data structures are pervasively used. We have rewritten them in EARTH-C and run the threaded code on the EARTH/MANNA emulator, which has a maximum of 20 processor nodes.

Treeadd sums up all tree-node values by traversing a binary tree in parallel.

Power solves a power system optimization by computing the best price for the benefit of the community that is made up of a four-level tree structure[19].

Em3d is a computational problem based on an irregular bipartite graph containing nodes representing electric and magnetic field values (E nodes and H nodes, respectively) [20].

Perimeter computes the perimeter of a quad-tree encoded raster image [27].

Mst computes the minimum spanning tree of a graph using Bentley's algorithm [4].

Table 2 lists main data structures and the running parameters for each benchmark.

Our experiments have been done using a multithreaded emulator, EARTH-MANNA [16], built on top of the MANNA parallel machine [6]. Each MANNA node consists of two Intel i860 XP CPUs, clocked at 50MHz, 32MB of dynamic RAM and a bidirectional network interface capable of transferring 50MB/s in each direction. The EARTH runtime system supports efficient remote operations. Sequentially, loading a remote word takes about $7\mu s$, calling a remote function can be performed in $9\mu s$, and spawning a new remote thread takes about $4\mu s$. When issued in a pipeline the operations take only one third of these times. Thus, placing many independent remote operations into one thread and issuing them in a pipeline may reduce the total execution time.

Benchmark	Main Data Structure	Parameters
treeadd	binary tree	5 or 6 levels
em3d	adjacent list	32K nodes and 50% of neighbors are remote
perimeter	quad-tree	Maximum 11 levels
power	four-level tree	10,000 leaves
mst	hash table and linked list	512 nodes

Table 2: Data Structure and Running Parameters

6.2 Compiler Timing Measurements

In this subsection we evaluate the execution time of two algorithms used to build a non-redundant graph.

Algorithm	Benchmarks							Average
	treeadd	tsp	power	perimeter	em3d	health	mst	
ancestor-sets	35	190	730	190	255	225	190	
marking	45	400	2,030	285	410	405	310	
Improvement	22%	52%	64%	33%	37%	44%	39%	**43%**

Table 3: Timing (ms) comparison between two graph building algorithms.

Table 3 gives the timing comparison of two algorithms. Th timing is measured by compiling the programs on a SUN sparc (SUN4m) workstation with 96 MB memory and running Sun OS 4.1.3.

From Table 3, we can see that the ancestor-sets based algorithm is much faster than the marking based one. On average, the ancestor-sets based algorithm is 43% faster than the marking based algorithm. For the *power* benchmark, the execution time of the ancestor-sets based algorithm is almost reduced to one third of that of the marking based algorithm. This shows that in practice the ancestor-sets based algorithm is a fast DDG building algorithm.

6.3 Effectiveness of Remote Height Based Scheduling

Benchmark	Schedule with remote height	Processors					Position diff in remote
		1	2	4	8	16	
treeadd level=5	off	0.70	0.61	0.41	0.40	0.39	
	on	0.70	0.59	0.40	0.38	0.35	
	Imp.(%)	0.00	2.77	2.57	4.75	2.45	Y
em3d 32K nodes	off	41,423	23,438	15,408	8,394	5,039	
	on	41,436	23,498	15,274	8,206	4,988	
	Imp.(%)	-0.03	-0.26	0.87	2.24	0.80	Y
perimeter level=11	off	5,469	3,914	1,945	1,030	726	
	on	5,360	4,080	1,885	961	672	
	Imp.(%)	2.00	-4.24	3.08	6.70	**7.44**	Y
power 10K leave nodes	off	6,3778	33,885	17,877	9,685	5,600	
	on	6,3787	33,968	17,828	9,660	5,400	
	Imp.(%)	-0.01	-0.24	0.27	0.26	3.57	N
mst 512 nodes	off	10,424	5,319	2,650	1,150	554	
	on	10,445	5,323	2,645	1,142	550	
	Imp.(%)	-0.20	-0.08	0.19	0.70	0.72	N

Table 4: The impact of the remote height optimization (time is in millisecond).

Table 4 lists the impact of applying the remote height heuristic to arrange the node's execution order within a thread[1]. For each benchmark, we give two types of timings, measured by turning *off* or *on* the remote height heuristic. The last column tells whether remote statements have been scheduled into different positions within the same thread, by comparing two versions of Threaded-C code generated. The 'N' means there are no changes of positions of *remote* statements compared to the code generated without using remote-height heuristic.

This table shows:

- In general the remote height heuristic has a positive impact on program performance. Except for the cases when 1 or 2 processors are used, the execution times of all benchmarks have been reduced.

- For some benchmarks, there is a significant improvement if remote height heuristic is applied. When running *perimeter* on 16-processors, the execution time is reduced by almost 7.5%.

[1] In writting timings are rounded off to two digits after the decimal point

- The impact of the remote height may be more visible when more processors are used. This is because when more processors are used, the more remote operations will be involved. Thus, more opportunities for the remote height heuristic to come into play.
- The effectiveness of the remote-height heuristic depends on parallelism existing among remote operations. For the *power* and *mst* benchmarks, since there is no parallelism exposed to the remote-height algorithm, there is no different schedule in terms of the position difference of remote statements between two versions of threaded code generated. Therefore, we hardly see any influence of remote-height on performance.

Therefore, we can conclude that in general the remote-height heuristic can improve the execution performance of threaded code generated. This thread scheduling algorithm requires more parallelism to be exposed so that there are more opportunities for the remote height heuristic to schedule parallel remote operations.

Since the remote height is only a heuristic, sometimes it actually slows down the execution of threaded code. The causes for such performance degradation is still under investigation.

7 Related Work

Because compiling for multithreading architectures is a relatively new field, research on this subject is scant. For a complete coverage, please refer to [31, 26, 35].

Many explicitly parallel languages have been proposed to support multithreaded program execution models, including Split-C [10], Cid [22], Cilk [5], EM-C [30]. However, automatic thread partition is not a main goal of the corresponding compilers.

Our remote-path heuristic was influenced by research work on task sequencing [38], instruction reordering [29], and instruction scheduling [12]. However, our approach is different in two ways: 1) Our compiler target model is a multithreaded machine and thus the latency hiding issue needs to be addressed; 2) we use *type* information to estimate the critical path rather than the *numeric* execution cost of each node.

The most related work to this paper is our remote-level based partition method [15]. We have identified the limitation of the method and improved it by considering the remote-path heuristic, which provides more general view for the thread partitioning process. Furthermore, we use a more efficient data structure *non-redundant DDG* replacing the adjacency matrix used in previous implementation. The non-redundant DDG can be built in $O(n^2)$ time in practice and the corresponding algorithm works very fast for real benchmarks.

Even though our target machines are *non-preemptive* multithreaded type, with slight modification our partitioning strategy can also be applied to a *preemptive* multithreaded model. We can further refine the DDG edges into two types: *definite* and *maybe*. A *definite* edge means there exists a definite data dependence between two nodes; a *maybe* edge means there may exist a data dependence between two nodes and at compile time we cannot disambiguate the nonexistence of such dependencies, such as those caused by pointer dereferences and array references.

Then, only *remote definite* edges is considered to introduce the *split-phase* constraint since if we put them into one thread, the thread will *definitely* hang itself as soon as the related remote operations are executed. Therefore, by splitting them into smaller threads, we can not only reduce the thread suspension cost but also provide more opportunities for fast thread switching. For the two nodes connected by *maybe* edges, we can assign them into one thread since we may rely on some detection mechanism to resolve data dependence at runtime. Therefore, our partitioning strategy will not increase any execution cost for a preemptive multithreaded machine but enhance performance of multithreaded code for any preemptive multithreaded model.

8 Conclusion and Future Work

In this paper we have described a general partition framework based on remote-path heuristics. We have designed an efficient non-redundant data structure to represent DDG. This representation can reduce the cost of later thread partitioning process. To build such a DDG, a practical $O(n^2)$ algorithm has been developed. We have implemented these algorithms in the EARTH-C compiler, and experiments show that the results are very encouraging.

The current thread partitioning algorithm is based on a very simple cost model, i.e., every remote operation takes "infinite" time and thus thread merging is always profitable. We plan to implement a more sophisticated thread-partition algorithm based on more refined program cost model in the EARTH-C compiler and evaluate its performance in real applications.

Acknowledgments

We thank L.J Hendren and Y.C Zhu (from McGill University) for providing source code of the marking based algorithm. This work was pereformed for the agreement with Jet Propulsion Laboratory, California Institute of Technology, and was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the National Security Agency (NSA) through an agreement with the National Aeronautics and Space Administration. The authors also acknowledge partial support from NSF (MIPS-9707125).

References

[1] T. Adam, K. Chandy, and J. Dickson. A comparison of list schedulers for parallel processing systems. *Comm. of the ACM*, 17:685–690, Dec. 1974.

[2] A. Agarwal, R. Bianchini, D. Chaiken, K. L. Johnson, D. Kranz, J. Kubiatowicz, B.-H. Lim, K. Mackenzie, and D. Yeung. The MIT Alewife machine: Architecture and performance. In *Proc. of ISCA-22*, pages 2–13, Santa Margherita Ligure, Italy, Jun. 1995.

[3] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Conf. Proc., 1990 Intl. Conf. on Supercomputing*, pages 1–6, Amsterdam, The Netherlands, Jun. 1990.

[4] J. Bentley. A parallel algorithm for constructing minimum spanning trees. *J. of Algorithms*, 1, 1980.

[5] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proc. of the 5th PPOPP*, pages 207–216, Santa Barbara, Calif., Jul. 1995.

[6] U. Bruening, W. K. Giloi, and W. Schroeder-Preikschat. Latency hiding in message-passing architectures. In *Proc. of the 8th Intl. Parallel Processing Symp.*, pages 704–709, Cancún, Mexico, Apr. 1994.

[7] M. C. Carlisle. *Olden: Parallelizing Programs with Dynamic Data Structures on Distributed-Memory Machines.* PhD thesis, Princeton University Department of Computer Science, June 1996.

[8] D. Chiou, B. S. Ang, R. Greiner, Arvind, J. C. Hoe, M. J. Beckerle, J. E. Hicks, and A. Boughton. StarT-NG: Delivering seamless parallel computing. In *Proc. of EURO-PAR '95*, number 966 in LNCS, pages 101–116, Stockholm, Sweden, Aug. 1995. Springer-Verlag.

[9] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* MIT Press; McGraw-Hill Book Co., Cambridge, Mass.; New York, N. Y., 1990.

[10] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. von Eicken, and K. Yelick. Parallel programming in Split-C. In *Proc. of Supercomputing '93*, pages 262–273, Portland, Ore., Nov. 1993.

[11] D. E. Culler, S. C. Goldstein, K. E. Schauser, and T. von Eicken. TAM – a compiler controlled threaded abstract machine. *J. of Parallel and Distrib. Computing*, 18:347–370, Jul. 1993.

[12] P. B. Gibbons and S. S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proc. of the SIGPLAN '86 Symp. on Compiler Construction*, pages 11–16, Palo Alto, Calif., Jun. 1986.

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

[14] L. Hendren, C. Donawa, M. Emami, G. Gao, Justiani, and B. Sridharan. Designing the McCAT compiler based on a family of structured intermediate representations. In *Proc. of the 5th Intl. Work. on Languages and Compilers for Parallel Computing*, number 757 in LNCS, pages 406–420, New Haven, Conn., Aug. 1992. Springer-Verlag. Publ. in 1993.

[15] L. J. Hendren, X. Tang, Y. Zhu, G. R. Gao, X. Xue, H. Cai, and P. Ouellet. Compiling C for the EARTH multithreaded architecture. In *Proc. of the 1996 Conf. on Parallel Architectures and Compilation Techniques*, pages 12–23, Boston, Mass., Oct. 1996.

[16] H. H. J. Hum, O. Maquelin, K. B. Theobald, X. Tian, G. R. Gao, and L. J. Hendren. A study of the EARTH-MANNA multithreaded system. *Intl. J. of Parallel Programming*, 24(4):319–347, Aug. 1996.

[17] H. H. J. Hum, K. B. Theobald, and G. R. Gao. Building multithreaded architectures with off-the-shelf microprocessors. In *Proc. of the 8th Intl. Parallel Processing Symp.*, pages 288–294, Cancún, Mexico, Apr. 1994.

[18] Y. Kodama, H. Sakane, M. Sato, H. Yamana, S. Sakai, and Y. Yamaguchi. The EM-X parallel computer: Architecture and basic performance. In *Proc. of ISCA-22*, pages 14–23, Santa Margherita Ligure, Italy, Jun. 1995.

[19] S. Lumetta, L. Murphy, X. Li, D. Culler, and I. Khalil. Decentralized optimal power pricing: The development of a parallel program. In *Proc. of Supercomputing '93*, pages 240–249, Portland, Ore., Nov. 1993.

[20] N. Madsen. Divergence preserving discrete surface integral methods for maxwell's curl equations using non-orthogonal unstructured grids. Technical Report 92.04, RIACS, Feb. 1992.

[21] O. C. Maquelin, H. H. J. Hum, and G. R. Gao. Costs and benefits of multithreading with off-the-shelf RISC processors. In *Proc. of EURO-PAR '95*, number 966 in LNCS, pages 117–128, Stockholm, Sweden, Aug. 1995. Springer-Verlag.

[22] R. S. Nikhil. Cid: A parallel, "shared-memory" C for distributed-memory machines. Tech. rep., Cambridge Res. Lab., Digital Equipment Corporation, Jul. 1994.

[23] M. D. Noakes, D. A. Wallah, and W. J. Dally. The J-Machine multicomputer: An architectural evaluation. In *Proc. of ISCA-20*, pages 224–235, San Diego, Calif., May 1993.

[24] K. Okamoto, S. Sakai, H. Matsuoka, T. Yokota, and H. Hirono. Multithread execution mechanisms on RICA-1 for massively parallel computation. In *Proc. of the 1996 Conf. on Parallel Architectures and Compilation Techniques*, pages 116–121, Boston, Mass., Oct. 1996.

[25] A. Rogers, M. C. Carlisle, J. H. Reppy, and L. J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM Trans. on Programming Languages and Systems*, 17(2):233–263, Mar. 1995.

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

[27] H. Samet. Computing perimeters of regions in images represented by quadtrees. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 3(6), Nov. 1981.

[28] V. Sarkar. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Res. Monographs in Parallel and Distrib. Computing. Pitman, London and The MIT Press, Cambridge, Mass., 1989. Revised version of the author's Ph.D. dissertation (Stanford U., Apr. 1987).

[29] V. Sarkar. Instruction reordering for fork-join parallelism. In *Proc. of SIGPLAN PLDI '90*, pages 322–336, White Plains, N. Y., Jun. 1990.

[30] M. Sato, Y. Kodama, S. Sakai, and Y. Yamaguchi. EM-C: Programming with explicit parallelism and locality for the EM-4 multiprocessor. In *Proc. of the Conf. on Parallel Architectures and Compilation Techniques*, pages 3–14, Montréal, Qué., Aug. 1994.

[31] K. E. 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, Calif., 1995.

[32] B. Sridharan. An analysis framework for the McCAT compiler. Master's thesis, McGill U., Montréal, Qué., Sep. 1992.

[33] X. Tang and G. R. Gao. How "hard" is thread partitioning and how "bad" is a list scheduling based partitioning algorithm? In *the Proceedings of Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, Puerto Vallarta, Mexico, Jun. 1998.

[34] X. Tang, R. Ghiya, L. J. Hendren, and G. R. Gao. Heap analysis and optimizations for threaded programs. In *Proc. of the 1997 Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 14–25, San Francisco, Calif., Nov. 1997.

[35] X. Tang, J. Wang, K. B. Theobald, and G. R. Gao. Thread partitioning and scheduling based on cost model. In *Proc. of SPAA '97*, pages 272–281, Newport, Rhode Island, Jun. 1997.

[36] K. R. Traub. Sequential implementation of lenient programming languages. Tech. Rep. MIT/LCS/TR-417, MIT Lab. for Comp. Sci., Oct. 1988. PhD thesis, Sep. 1988.

[37] D. M. Tullsen, S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, and R. L. Stamm. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proc. of ISCA-23*, pages 191–202, Philadelphia, Penn., May 1996.

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