Dynamic Pointer Alignment: Tiling and Communication Optimizations for Parallel Pointer-based Computations

Xingbin Zhang
Department of Computer Science
University of Illinois at Urbana-Champaign

Andrew A. Chien
Department of Computer Science
University of Illinois at Urbana-Champaign
and Hewlett Packard Laboratories

{*zhang,achien*}*@cs.uiuc.edu*
http://www-csag.cs.uiuc.edu/

Abstract

Loop tiling and communication optimizations, such as message pipelining and aggregation, can achieve optimized and robust memory performance by proactively managing storage and data movement. In this paper, we generalize these techniques to pointer-based data structures (PBDSs). Our approach, *dynamic pointer alignment* (DPA), has two components. The compiler decomposes a program into non-blocking threads that operate on specific pointers and labels thread creation sites with their corresponding pointers. At runtime, an explicit mapping from pointers to dependent threads is updated at thread creation and is used to dynamically schedule both threads and communication, such that threads using the same objects execute together, communication overlaps with local work, and messages are aggregated. We have implemented DPA to optimize remote reads to global PBDSs on parallel machines. Our empirical results on the force computation phases of two applications that use sophisticated PBDSs, *Barnes-Hut* and *FMM*, show that DPA achieves good absolute performance and speedups by enabling tiling and communication optimizations on the CRAY T3D.

1 Introduction

To achieve robust, high performance on parallel applications, programming systems must overcome two fundamental limitations of modern memory hierarchies: limited transfer bandwidth and long data access latencies. Because bandwidth and latency hierarchies can span several orders of magnitude, both data reuse optimizations to minimize bandwidth usage and latency tolerance optimizations are required. Loop tiling [1, 4] and communication optimizations [20], such as message pipelining (i.e. prefetching) and aggregation, can achieve optimized and robust memory performance by proactively managing storage and data movement. Tiling improves data reuse using a combination of strip-mining and iteration reordering to group iterations that access the same data. Message pipelining hides communication latency, while message aggregation reduces communication overhead by grouping data movement.

In this study, we generalize tiling and communication optimizations to pointer-based data structures (PBDSs). PBDSs are fundamental for efficiency and expressibility for a growing number of important parallel applications. However, loop tiling and communication optimizations do not directly apply to PBDSs due to two inherent compile-time limitations. First, precise aliasing information for general PBDSs is an open problem (e.g. [18, 19]), and thus the compile-time data access pattern information required for static optimizations is unavailable. Second, computations on PBDSs are often expressed as recursive functions or *while* loops with data-dependent termination conditions, causing unknown control flow and thus inhibiting static iteration reordering. As a result, current locality optimizations for general PBDSs often rely exclusively on dynamic techniques, such as caching and multithreading, that reactively control data movement and storage usage.

Our approach, *dynamic pointer alignment* (DPA), has two components. The compiler decomposes a program into non-blocking threads that operate on specific pointers and labels thread creation sites with their corresponding pointers. At runtime, an explicit mapping from pointers to dependent threads is updated at thread creation time and is used to dynamically schedule both threads and communication, such that threads using the same objects execute together, communication overlaps with local work, and messages are aggregated. To be effective, DPA requires low-overhead fine-grained thread operations and relaxed control and data dependency information to enable communication and thread reordering. We implemented DPA using a hybrid compiler/runtime framework that uses a simple and low-overhead execution model and exploits static aliasing and dependence information to enlarge thread granularity and enable reordering. We have found that coarse-grained aliasing and block-level concurrency information are often sufficient to enable optimizations. This result is encouraging because it is practical to obtain coarse-grained aliasing information, and block-level concurrency annotations are both simple to add to programs and already supported by various programming languages [5, 8, 16].

We have implemented DPA in the Illinois Concert system[1] [7] in the context of optimizing concurrent object-oriented programs relying on global, distributed PBDSs on parallel machines. We have experimentally evaluated our technique on the CRAY T3D, on the force computation phases of Barnes-Hut and FMM, two hierarchical N-body programs from the SPLASH-2 suite [38] that rely on non-trivial PBDSs. Our results show that DPA achieves both good absolute performance and speedups (42-fold for Barnes-Hut and 54-fold for FMM on 64 nodes) by enabling tiling to effectively exploit data reuse as well as message pipelining and aggregation to reduce idle time by up to a factor of 13 and communication overhead by 34%. The specific contributions of this study are:

Proceedings of the 1997 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, June 1997, Las Vegas, Nevada.

[1] Concert 4.0 is publically available at `http://www-csag.cs.uiuc.edu`.

- DPA, a locality optimization technique that enables tiling and communication optimizations for PBDSs,
- An implementation framework for DPA that can exploit coarse-grained aliasing and block-level concurrency information for optimization, and
- Experimental demonstrations of DPA's effectiveness for programs relying on non-trivial PBDSs.

We next present our approach (§2) and implementation framework (§3). We then describe our compiler algorithms (§4), followed by experimental results (§5). Finally, we compare DPA with related work (§6) and conclude with future work (§7).

2 Dynamic Pointer Alignment (DPA)

We illustrate our approach by contrasting static loop tiling and communication optimizations with DPA. We first consider the requirements of the static techniques, illustrated in Figure 1. These techniques require a well-defined iteration space, such as a loop nest, and data access pattern information that correlates iterations and the data they access. In addition, tiling requires the ability to reorder iterations within the iteration space, while message pipelining and aggregation require the ability to explicitly schedule data movement (communication). Although these techniques do not apply directly to pointer-based computations due to the lack of static access pattern and control flow information, these requirements motivate our approach.

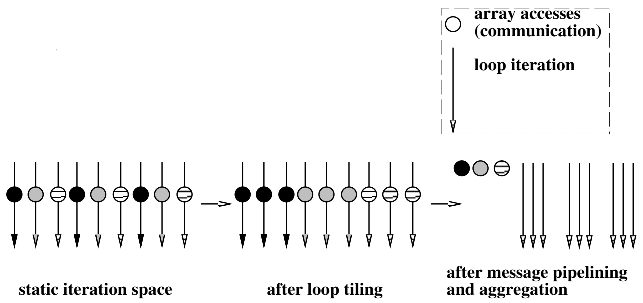


Figure 1: Loop tiling and communication optimizations. (Time flows from top to bottom then left to right within each set of iterations.) Both techniques rely on well-defined static access pattern information and iteration space.

Our approach has two components: the compile-time transformations, shown in Figure 2, and runtime communication and computation scheduling shown in Figure 3. At compile time, DPA first decomposes the program into fine-grained, non-blocking (i.e. run-to-completion) threads (Figure 2(a)). These threads correspond to loop iterations in static frameworks. At runtime, threads enable dynamic reordering, and spawned threads form a well-defined runtime "iteration" space at thread scheduling time, shown in Figure 3(a). However, this is insufficient for optimizations because thread access patterns are not known until thread execution, which is too late. To enable thread access pattern to be discovered earlier, threads are constructed such that all heap-directed pointers accessed within a thread are available at the thread entry, and a second compile-time transformation labels each thread creation site with these pointers (Figure 2(b)). At runtime, an explicit mapping from pointers to dependent threads is updated at thread creation and used to dynamically schedule threads such that threads using the same objects execute together, shown in Figure 3(b). However, when implicit data renaming (such as provided by a cache) is unavailable, thread scheduling alone is insufficient for communication optimizations as data movement is embedded within thread executions. The third compile-time transformation hoists all enclosed object accesses within a thread outside the thread and explicitly renames original accesses, such that no communication occurs inside threads (Figure 2(c)). At runtime, in addition to thread scheduling, the mapping information is also used to separately schedule communication to overlap with local work and to aggregate messages, shown in Figure 3(c). Thus, programs execute in "epochs" of threads where the precise access pattern is known, and communication and threads can be separately and simultaneously scheduled, replicating the static situation at runtime (contrast Figure 3(c) with Figure 1).

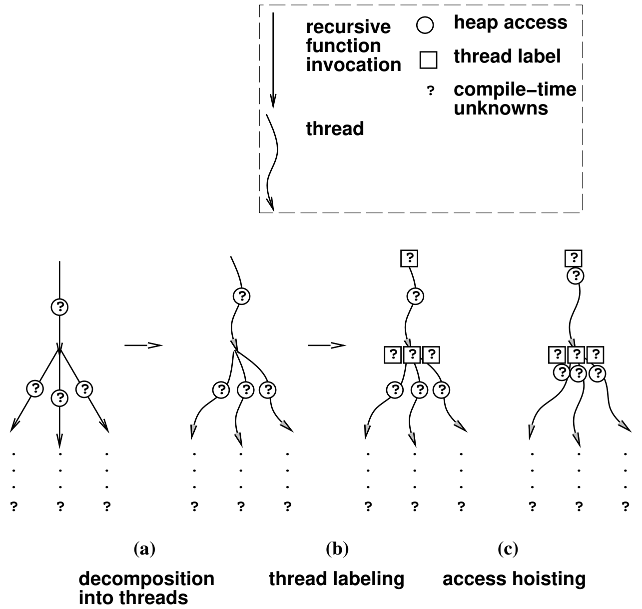


Figure 2: DPA compile-time transformations. (a) DPA decomposes computations into non-blocking threads. (b) DPA labels each thread creation site with heap-directed pointers referenced within the thread. (c) DPA hoists all accesses to outside the thread and renames original accesses to use parameters.

Compared with prior locality optimizations for general PBDSs, such as caching [3, 14], prefetching [23], and multithreading [10], the simultaneous computation and communication scheduling using an explicit runtime mapping is a unique feature provided by DPA. Thread scheduling enables tiling, and communication scheduling enables both message pipelining and aggregation. Our approach is similar in spirit to the inspector/executor model [31] for discovering the access pattern at runtime and using it for dynamic optimizations. However, because DPA uses threads for dynamic reordering, it allows incremental runtime mapping construction without separate inspector phases. As a result, unlike inspector/executor DPA can optimize recursive data accesses, prevalent in pointer-based computations, as well as accesses through indirection arrays.

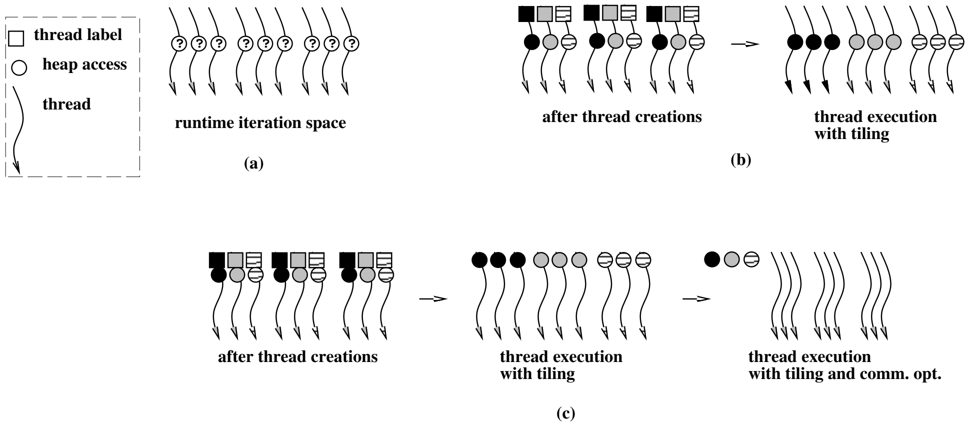


Figure 3: Effect of each static transformation at runtime. (Time flows from top to bottom then left to right within each set of threads.) (a) Spawned threads form a well-defined runtime "iteration" space at thread scheduling time. (b) Thread labeling enables thread access patterns to be known at thread creation, enabling thread tiling. (c) Access hoisting enables data movement to be scheduled separately from threads, enabling prefetching and access aggregation, in addition to tiling.

3 Implementation Framework

DPA requires low-overhead thread operations and relaxed control and data dependency information to enable communication and thread reordering. We have developed a hybrid compiler/runtime implementation framework that uses a simple and low-overhead execution model and exploits static aliasing and dependence information to enlarge thread granularity and enable global dynamic reordering.

3.1 Execution Model

In our execution model, a program consists of non-blocking threads that may create additional threads and executes using a single stack. At runtime, spawned threads become ready when their corresponding hoisted object accesses complete, and ready threads are freely reordered. The compiler determines scheduling points where threads are run and uses runtime data structures to enforce the dependency between threads and their required data by scheduling both communication and threads. The runtime system explicitly maintains two data structures: the set of pointers (thread labels), D, encountered since the last scheduling point and a mapping, M, from pointers to threads. This mapping is an explicit runtime representation of the thread access pattern. The interface to the compiler consists of the *spawn* function for thread creation and iterators for traversing D and M. Each `spawn(ptr,func,arg`$_1$`,arg`$_2$`,...)` call creates a thread `T` for the function `func` by saving the its arguments as state, inserts `ptr` into D, and updates the mapping by adding T to $M(ptr)$. At the beginning of a scheduling point, because threads may create additional threads, iterators take a snapshot of D and M, and new entries are appended to the end.

3.2 Compilation

The compiler extracts threads from serial computation for labeling and hoisting and determines the scheduling points for thread execution. It relies on aliasing and concurrency information,[2] outlined below, to enlarge thread granularity and enable reordering. We describe our compiler algorithms in detail in Section 4.

[2]Both general alias and dependence analyses for PBDSs are challenging research problems (e.g. [17, 18, 19, 30]) and are out the scope of this study.

3.2.1 Aliasing Information

We view the heap memory as a collection of records and assume that each heap access can be normalized to the form `ptr->field`, where `ptr` is a heap-directed pointer and `field` the name of a record field. Essentially, this prohibits unstructured accesses through pointer arithmetic. All static heap analyses must finitely summarize the storage shape by partitioning heap nodes into compile-time partitions, such as by types (coarse-grained) or by individual heap objects (fine-grained). Our compilation is parameterized by this partition granularity. Given a partition of heap nodes $\{G_1, G_2, ...\}$, we require the following for each pointer:

may-point-to(ptr) returns $\{G_i, G_j, ...\}$, the set of partitions `ptr` may point into.

must-point-to(ptr) returns G_k, if `ptr` must point-to an (arbitrary) node in G_k.

3.2.2 Object Consistency Model

We use a phase-based consistency model that matches the given aliasing information. A common access pattern in programs with well-delimited data structure building and use phases is reads and writes to disjoint fields within a given phase. We focus on this type of read access by summarizing access pattern within a program phase – represented statically as a set of methods closed under calls that do not execute concurrently with any outside method – across each heap partition of the given aliasing information. More specifically, for each heap partition G, a read on a field through a pointer that *must-point-to* G is eligible for reordering if the field is never updated through any pointer that *may-point-to* G within the phase. We have found that coarse-grained aliasing information is often sufficient to enable optimizations using this consistency model. More precise aliasing information can enable optimizations of more general access patterns, such as reductions.

3.2.3 Dependence Information

We assume the following form of block-level concurrency information, obtainable from either language constructs [5, 8, 16] or by analyses [19, 30] in restricted cases.

```
conc {                      conc for(i=0;i<5;i++)
 object.method_1();          objects[i].method();
 object.method_2();
}
```

The `conc` specification asserts the absence of indirect dependency among loop iterations or statements within a block and enables interleaved execution.

3.3 Project Context

We implemented DPA in the Illinois Concert compiler [7], which uses program dependence graph (PDG) [15] in gated SSA [12] form as its intermediate representation. The Concert compiler supports an adaptive, context-sensitive interprocedural analysis framework [27, 28] that we use to implement our analysis and code transformations. The Concert compiler also supports a suite of optimizations to reduce overhead due to object orientation. In particular, our example codes benefit from inline allocation of objects [13] to enlarge object granularity that amortizes object access overhead and simplifies communication of object state. Our source language is ICC++ [6, 8], which extends C++ with object-level sequential consistency and block-level concurrency constructs shown in §3.2.3.

3.4 Example

We illustrate our approach using a pair of examples. Let us first consider the array access optimizations shown in Figures 4 and 5, where accesses to the `B` array are tiled, using a combination of strip-mining (line 1) and loop interchange (lines 3–4), and are aggregated (line 2) with explicit renaming using the `temp` array (line 5). Figure 6 shows a similarly-structured pointer-based code that searches a binary tree for a set of keys and a sample search sequence at runtime. Static optimizations do not directly apply; the analog of loop interchange, for example, requires the checking of every tree node due to statically unknown control flow.

```
for(i = 0; i < N; i++)
  for(j = 0; j < M; j++)
    compute(A[i],B[j]);
```

Figure 4: An array-based code

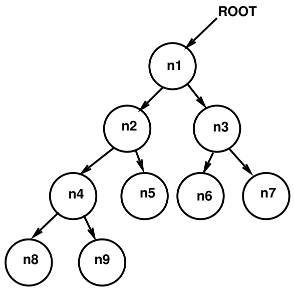
```
for(k = 0; k < M/STRIP; k++)                (1)
 temp[0:STRIP-1]=B[k*STRIP:(k+1)*STRIP-1];(2)
 for(j = 0; j < STRIP; j++)                 (3)
  for(i = 0; i < N; i++)                    (4)
     compute(A[i],temp[j]);                 (5)
```

Figure 5: Optimized array code

DPA enables the static optimizations of accesses to the binary tree. At compile time, the `conc` annotation asserts no global dependency among separate traversals, enabling them to be interleaved for optimization, and all accesses through the pointer, `ptr`,

```
conc for(i = 0; i < N; i++)
  search(ROOT, A[i]);

search(ptr, key) {
  if (ptr->val == key) return;
  else if (ptr->val < key)
    search(ptr->left, key);
  else
    search(ptr->right, key);
}
```



$s_0(n_1), s_0(n_2), s_0(n_4), s_0(n_8)$
$s_1(n_1)$
$s_2(n_1), s_2(n_2), s_2(n_4), s_2(n_9)$
$s_3(n_1), s_3(n_3)$

Figure 6: An pointer-based code (top) that searches a binary tree for a set of keys, an example binary tree (center), and an sample search order at runtime (bottom) for four keys ($s_i(n_j)$ denotes that the search for key A[i] accesses tree node n_j.).

are eligible for optimizations as they are to fields not updated within the phase. Figure 7 shows DPA's static transformations, starting from the original code in Figure 6. First, because all accesses through `ptr` are to the same object, the compiler is able to create a single thread `SEARCH` that encloses them, shown in Figure 7(a). The thread and thread creation using `spawn` replace the original `search` function and calls, so that computation that depends on accesses through `ptr` can be reordered at run time using thread creation. In addition, the thread is constructed such that the value of `ptr` is available at the thread entry for labeling and access hoisting. Second, each thread creation site (spawn) is labeled by the corresponding pointer, shown in bold in Figure 7(b). This labeling enables the mapping from pointers to threads to be updated at thread creation. Finally, to enable separate scheduling of communication due to the accesses, all accesses through `ptr` within the thread are hoisted to outside the thread and explicitly renamed using parameters, shown in bold in Figure 7(c). A separate thread scheduling code (discussed next) schedules the actual accesses outside the thread and passes the results to the thread at thread execution time.

The compiler inserts scheduling code after the concurrent loop to simultaneously schedule threads and communication, as shown in Figure 8. Using both runtime data structures D, the set of referenced pointers, and M, the mapping from pointers to threads, many variations on scheduling are possible to optimize for different criteria. Figure 8 shows that using D, the communication due to accesses to `STRIP` pointers is aggregated (line 2), and using M, threads accessing the same tree node are reordered to execute together (lines 4–5), analogous to Figure 5 (line 2 and lines 4–5). At runtime, the program executes in "epochs" of threads that batch

```
conc for(i = 0; i < N; i++)
   spawn  (SEARCH, ROOT, A[i]);

SEARCH(ptr, key) {
 if (ptr->val == key) return;
 else if (ptr->val < key)
  spawn  (SEARCH, ptr->left, key);
 else
  spawn  (SEARCH, ptr->right, key);
}
```
(a)

```
conc for(i = 0; i < N; i++)
 spawn(ROOT ,SEARCH, ROOT, A[i]);

SEARCH(ptr, key) {
 if (ptr->val == key) return;
 else if (ptr->val < key)
  spawn( ptr->left  ,SEARCH,ptr->left,key);
 else
  spawn( ptr->right  ,SEARCH,ptr->right,key);
}
```
(b)

```
conc for(i = 0; i < N; i++)
 spawn(ROOT,SEARCH, ROOT, A[i]);

SEARCH(key, val, left, right  ) {
 if (val == key) return;
 else if (val  < key)
  spawn( left,SEARCH, left  , key);
 else
  spawn( right ,SEARCH, right  , key);
}
```
(c)

Figure 7: DPA compile-time transformations. Starting from the code in Figure 6: (a) thread decomposition, (b) thread labeling, (c) access hoisting. Successive differences are shown in bold.

the traversals breadth-wise for optimizations, and the scheduling loop iterates until all traversals are complete. Figure 9 shows the runtime execution sequence for the sample sequence in Figure 6. The ensuing thread operation at runtime is cheap: consisting of a hash operation to update the mapping M at thread creation, a direct function call, and a save and restore of function arguments.

```
conc for(i = 0; i < N; i++)
 spawn(ROOT, SEARCH, ROOT, A[i]);
while(D not empty) {                        (1)
 let p_0,p_1,... be the next STRIP pointers in D
   fetch data of p_0,p_1,... into temp      (2)
 for(j = 0; j < STRIP; j++)                 (3)
   for(T = first(M(p_j));T;T = T->next)     (4)
     SEARCH(T->key,temp[j].val,             (5)
            temp[j].left,temp[j].right);
```

Figure 8: Code for scheduling threads and communication, placed after the top-level concurrent loop.

read n_1
$s_0(n_1), s_1(n_1), s_2(n_1), s_3(n_1)$
read n_2, n_3
$s_0(n_2), s_2(n_2), s_3(n_3)$
read n_4
$s_0(n_4), s_2(n_4)$
read n_8, n_9
$s_0(n_8), s_2(n_9)$

Figure 9: Resulting computation and communication order for the example access sequence shown in Figure 6.

4 Compiler Algorithms

In this section, we present our compiler algorithms, focusing on thread extraction. The compilation aims for two main optimizations: first, enlarging thread granularity and minimizing thread states to reduce runtime overhead and second, minimizing dependency among threads to enable reordering. The main idea is to partition heap accesses so that those using the same pointer, or pointers *must-aliased* together, are grouped (§4.1), and construct a single thread for each access partition (§4.2). The compiler then performs labeling, hoisting, and thread scheduling (§4.3) to enable our execution strategy. We assume the input is a set of methods closed under calls, representing a program phase, annotated with the aliasing and concurrency information discussed earlier. We describe the algorithms in the context of optimizing remote memory accesses. In detailed description of the algorithms, we use the notations shown in Figure 10.

R is a PDG region, v is a value in SSA form, n and m are PDG program nodes, P is a set of program nodes, and f is a method.

Region(n), Region(P) return the region of n or P, if all nodes in P fall in the same region.

RegionSucc(n), RegionPred(n) return the dependence successors and predecessors of n in $Region(n)$, respectively.

Dominates(n,m) returns true if node n dominates m interprocedurally, and

Closure(P,func) is the transitive closure of calling the function $func$ starting from the set of nodes P.

Figure 10: Notations for the discussion of compiler algorithms.

4.1 Heap Access Partitioning

To group accesses through the same pointer, we first explicitly mark all accesses eligible for optimization based on our consistency model discussed in Section 3.2.2. To focus optimization on remote accesses, we use a separate interprocedural locality analysis to eliminate accesses to local objects from consideration. We let C be the set of program nodes containing eligible accesses and assume each node contains a single access. We partition C so that accesses through pointers that are *must aliased* are grouped together. Figure 11 shows the partitioning function, *NextAccessPartition*. Given a pointer through which an access in C is made, the *TrueDef* function uses backward interprocedural data flow to find the earliest dominator node, n, of the access such that the value of the pointer is available at n. This node is either a definition of the pointer or an interprocedural confluence of definitions of the pointer. Pointers that share the same $TrueDef$ are must-aliased, and $NextAccessPartition$ returns a partition of C such that all access pointers are must-aliased. By its construction, although members in the same access partition may be scattered in different functions, they all correspond to a single program node containing the *TrueDef* of their pointers, which we refer to as the *definition node* of the pointer of an access.

PTR(n) given an $n \in C$, returns the pointer through which the access is made.

TrueDef(v) returns the interprocedural definitions of v, by flowing backward recursively through moves and function entries with one call site. We use $TrueDef(C)$ as a shorthand for $TrueDef(PTR(n)), n \in C$.

NextAccessPartition(C) returns an arbitrary maximal $C_1 \subseteq C$ such that

$$\forall_{m,n \in C_1} TrueDef(PTR(m)) = TrueDef(PTR(n)).$$

Figure 11: Heap access partitioning using must-aliasing among pointers.

4.2 Thread Extraction

Given an access partition C_1, our thread extraction algorithm constructs a single non-blocking thread that encloses all accesses in the partition, i.e. each thread references only one potentially remote object. Our primary tool to minimize thread state and dependence is to make threads larger by selectively including dependent computations into threads. Because eligible accesses are invariant within the phase, thread labeling and access hoisting corresponds to one constraint on thread extraction: the *definition node* of the access partition must not be included inside the thread. Given this constraint and that each thread references a single remote object, our basic algorithm greedily constructs as large a thread as possible. When the resulting thread is created in a control construct prohibiting reorder, such as a while loop, we apply additional transformations to further decouple dependencies.

The thread extraction problem can be more formally defined as follows: given an access partition C_1, construct a non-blocking thread f satisfying two criteria:

Inclusion $\forall n \in C_1$, the entry of f dominates n, ensuring that the thread includes all given accesses, and

Hoistability $TrueDef(C_1)$ dominates the entry of f, ensuring that $\forall n \in C_1$, the value of $PTR(n)$ is available at the entry of f for labeling and access hoisting.

Figure 12 shows our basic extraction algorithm.[3] Intuitively, the algorithm extends the thread by tracing each access interprocedurally to the *definition node* of its partition: all control constructs, such as loops, on the paths are included as a whole except the last one containing the *definition node*. Within the last control construct, the algorithm includes, transitively, all dependent program nodes into the thread, eliminating local dependence. This final transitive expansion guarantees linearizability [2] required by the function promotion. Applying the basic algorithm on the example in Section 3.4 results in the thread structure shown in Figure 7(a).

When the *definition node* of an access partition falls in a control construct that prohibits reordering, such as a while loop, dependency disables thread reordering. Because of hoistability, we cannot simply extend the thread to above the parent region. For such situations, we apply several enabling transformations in a `PreExtraction` phase before running the basic thread extraction algorithm to localize dependency. For example, we transform while loops into tail recursive calls to allow the basic extraction algorithm to extend over incomplete iterations. Figure 13 shows the overall algorithm. Because thread extraction transforms the code and accesses to different objects may interleave with each other, the algorithm partitions accesses and extracts threads iteratively.

[3] Note that a degenerate solution satisfying the criteria exists, by cloning the accessor functions for every access.

ExpandExcluding(P,n) returns $Closure(P, RegionSucc) - Closure(\{n\}, RegionPred)$, the transitive closure of dependence successors of nodes in P, excluding n and its predecessors, all in the same region.

Promote(P) given a linearizable set P consisting of nodes from the same region R, promotes P into a function f and replaces P with a call to f with upwardly exposed values in P as arguments and values live after P as return values.

BasicExtraction(C_1)
  $R \leftarrow Region(Node(TrueDef(C_1)))$
  $P \leftarrow \bigcup_{m \in Members(R)} \exists n \in C_1, Dominates(m, n) = true$
  $Partition \leftarrow ExpandExcluding(P, Node(TrueDef(C_1)))$
  Promote(Partition)

Figure 12: Basic thread extraction algorithm.

PreExtraction(C) preconditions the code to enhance $BasicExtraction$, such as converting while loops to tail recursive calls and moving code dependent on a conditional into both arms of the conditional.

ThreadExtraction(C)
  while($C \neg = \phi$)
    $C_1 \leftarrow NextAccessPartition(C)$
    $PreExtraction(C_1)$
    $BasicExtraction(C_1)$
    $C \leftarrow C - C_1$

Figure 13: Overall thread extraction algorithm.

4.3 Labeling, Hoisting, and Scheduling

To implement our execution strategy, the compiler labels each thread creation with the pointer of the access partition and hoists all enclosed accesses to outside the thread by renaming original accesses and passing the required data as parameters, resulting in Figure 7(c) for the example in Section 3.4. To determine scheduling points for thread execution, we mark all concurrent control constructs, including not only concurrent blocks and loops but also concurrent tail call cycles where all recursive calls occur in the last concurrent block of the function (corresponding to fanout trees). For each thread creation, we place scheduling points after the outermost construct of the enclosing nest of concurrent loops and tail call cycles. The compiler chooses a scheduling template, such as the ones in Figure 14, for optimization. To limit excess concurrency due to storage constraints, we statically strip-mine all top-level concurrent loops with a given strip size, similar to *k-bounded* loops [11]. In addition, because dynamic data distribution is prevalent in pointer-based computations, the compiler speculatively optimizes thread creation by guarding each thread entry with a locality test on the pointer label, creating the thread only when the corresponding object is remote.

```
while(D not empty)
  fetch next ptr in D
  forall(T belongs to M(ptr))
    run T;

for(i = 0; i < K - 1; i++)
  fetch next B pointers in D
while(D not empty)
  fetch next B pointers in D
  TOUCH earliest fetch of block
  for(i = 0; i < B; i++)
    forall(T belongs to M(ptri))
      run T;
```

Figure 14: Two example scheduling templates: tiling with tile size 1 (top) and prefetching of blocked accesses, with `K` outstanding requests and blocks of `B` accesses (bottom). D and M are runtime data structures described in Section 3.2.2. `T` represents a thread, and `TOUCH` blocks until the given request completes.

5 Experimental Results

We empirically evaluated DPA using two applications from the SPLASH-2 suite [38], Barnes-Hut and FMM, on the Cray T3D [9], a distributed memory machine with support for one-sided remote put/get memory operations. We use Illinois Fast Messages (FM) [22, 26] for our message passing implementation. We selected these applications because they rely on sophisticated PBDSs for efficiency, and previous studies [3, 14, 29, 34, 35, 37] show that locality optimizations are crucial to achieve good performance. Our codes are in ICC++ and are adapted from the SPLASH-2 version. We concentrate on the force computation phases because they dominate the sequential execution time (95% for Barnes-Hut and 90% for FMM) and exhibit ample locality optimization opportunities. After a brief description of the codes and results of DPA's transformations, we first evaluate runtime and storage overhead due to thread reordering. We then evaluate communication optimizations with a message passing implementation and evaluate tiling using the T3D's put/get interface.[4] We also use a manual implementation of object caching that is representative of current dynamic data reuse optimizations[5] as a baseline for comparison. To minimize code generation differences, we use the ICC++ sequential version[6] as the base for speedups. The sequential ICC++ version of Barnes-Hut and FMM are within 25% and 2% of optimized sequential C++/C version, respectively.

5.1 Barnes-Hut and FMM

Both Barnes-Hut and FMM are hierarchical N-body applications relying on global oct- and quad-trees, consisting of spatial cells and particles. In a typical spatial data distribution, both force computation phases consist of pair-wise tree node interactions involving one potentially remote node, whose accesses should be optimized. For Barnes-Hut, all particles concurrently traverse the oct-tree recursively with a termination condition dependent on spatial distances. For FMM, all cells concurrently compute against several interaction lists of tree nodes. Optimization opportunities for data reuse exist because of overlaps in tree traversals or sharing in interaction lists. For both codes, coarse-grained aliasing information, obtained from the type system, enables optimization for all non-local accesses. There are two and six types of pair-wise interactions for Barnes-Hut and FMM, respectively, and DPA constructs a thread for each type of interactions. All thread states are small: five thread functions have a single pointer value (for the local node) as states, and a Barnes-Hut thread function has a pointer (local node) and a float (distance) as states. For both codes, all thread scheduling points are placed after the outermost concurrent loop, enabling both tiling and communication optimizations of remote accesses across multiple tree traversals or cell computations. The outermost loops of both codes are statically strip-mined to limit concurrency.

5.2 Runtime and Storage Overhead

We first evaluate the runtime overhead of thread creation and execution and the storage overhead of runtime data structures. Tables 1 and 2 show the execution time difference between a sequential run with no thread creation and one using DPA for both codes. The tables show that the overall overhead of runtime reordering is low for both codes: the cost of 0.52 $usec$ for Barnes-Hut is cheaper than the fastest remote access on the T3D (around 1.5 μsec).[7] The overhead for FMM, 2.61 $usec$, while less than two remote accesses, is higher than expected. We believe it is mainly due to cache effects, which can be optimized with a better layout of DPA's runtime data structures.

Sequential Time	115.20 secs
Sequential Always Threads	123.13 secs
Total No. Threads	15.2 million
Average Cost per Thread	0.52 μsec

Table 1: Barnes-Hut reordering overhead.

1 Processor Local Time	7.17 secs
16 Processor Local Time	7.34 secs
Total No. Threads	65,242
Average Cost per Thread	2.61 μsec

Table 2: FMM reordering overhead.

Because DPA requires additional storage to hold outstanding thread contexts, it is also important to quantify the storage overhead. Table 3 shows the maximum number of outstanding threads and the number of pointers in D on 16 nodes for strip sizes of 50 and 300 for both codes. The storage requirements are both small: counting data structure overhead, 27,722 threads for Barnes-Hut require less than 700 KB, while for FMM, the storage for strip size of 300 is under 90 KB.

5.3 Communication Optimizations

To evaluate communication optimizations, we measured the execution time breakdown of a message passing implementation of both codes using DPA on 16 nodes, as shown in Figure 15. The figure shows total computation time in terms of idle time, communication overhead, and local computation, with the speedup shown on top of each bar. The `Base` bars of both codes use DPA for

[4] Because of the limitations of T3D's prefetch queue (a single queue that allows up to 16 outstanding words), we did not experiment with communication optimizations using the put/get interface.

[5] We assume cache lookups are placed at procedure boundaries (one lookup for each method). Misses fetch the entire object with no capacity constraints and only self-invalidation, keeping all other factors constant including the hashing function.

[6] The Concert compiler has a sequential mode that generates code without parallelization overhead.

[7] Although reordering computation may also improve sequential performance (see Section 6 and [21]), because of the small L1 cache and the lack of a L2 cache, we believe this is not the case on a single T3D node.

Code	Static Strip Size	Max. no. of Suspended Threads	Max. elements of Data space
Barnes-Hut	50	4,749	495
	300	27,722	833
FMM	50	2,386	1,198
	300	5,472	1,714

Table 3: Barnes-Hut and FMM storage statistics on 16 nodes.

tiling only without communication optimizations and clearly show that data access latency due to communication and target processor unresponsiveness[8] causes significant idle time. The object caching implementation performs comparably to this base case, with speedup of 7.22 for Barnes-Hut and 9.46 for FMM, because caching does not provide communication optimizations. The `Pipe 16` and `Pipe 4` bars (prefetching with 16 and 4 outstanding messages, respectively) show that DPA with message pipelining significantly reduces the idle time: by a factor of 6.1 for Barnes-Hut and a factor of 3.7 for FMM. In addition, the `Base` for Barnes-Hut shows significant communication overhead. DPA with message aggregation also effectively reduces this overhead; the final configuration reduces the communication overhead by 34% with the majority of the remaining overhead due to empty polling and reduces the initial idle time by a factor of 13. As a result of the communication optimizations, the final configuration of DPA significantly outperforms caching for both codes.

5.4 Tiling and Overall Speedup

To evaluate tiling, We show the speedup and execution time of the force phases of Barnes-Hut and FMM implemented using DPA with only tiling (for strip sizes of 50 and 300) and object caching, using the T3D's put/get interface, in Figure 16. The figure shows that DPA enables data reuse to achieve good speedups for both strip sizes, while larger strip sizes enable more reuse, which improves speedup at larger machine configurations. Both configurations of DPA achieve better speedups than the caching scheme, which we believe is due to minimized hashing and better cache performance because of access hoisting. Although configurations differ, our speedup of over 42 on 64 nodes for Barnes-Hut is competitive with other studies both on MPPs [3, 14] and on shared-memory architectures [34], and the speedup of FMM, 54-fold on 64 nodes, also compares favorably with other implementations [29, 35].

6 Discussion and Related Work

The goal of DPA is to generalize loop- and array-oriented tiling [1, 4, 24, 32] and communication optimizations [20] to pointer-based computations. Although developed independently, our use of non-blocking threads labeled by pointers is similar to the recent cache optimization study by Philbin, et al. [21]. In [21], the programmer manually extracts threads and can supply multiple scalar hints as thread labels. DPA is an automatic approach relying on the compiler for thread extraction and integrates communication optimizations, not addressed in [21], to tolerate latency and reduce communication overhead. DPA performs explicit renaming and is also applicable to cache optimizations, which we are currently investigating. The execution strategy of DPA is also similar to the inspector/executor model [31]. However DPA constructs dynamic access pattern incrementally, enabling optimization for recursive

[8] This is despite the manual optimization of poll placement.

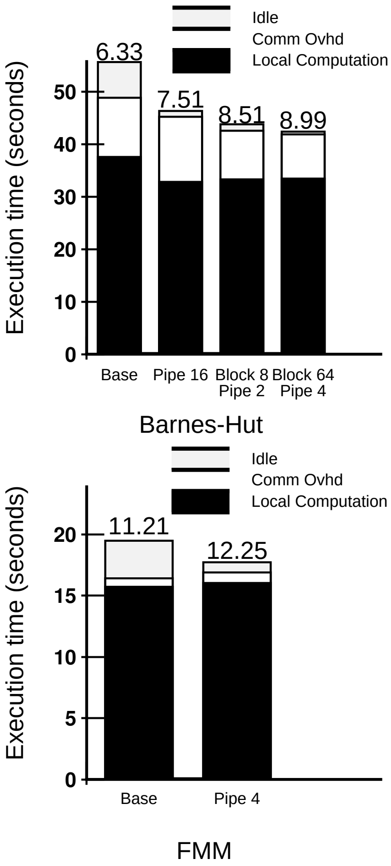
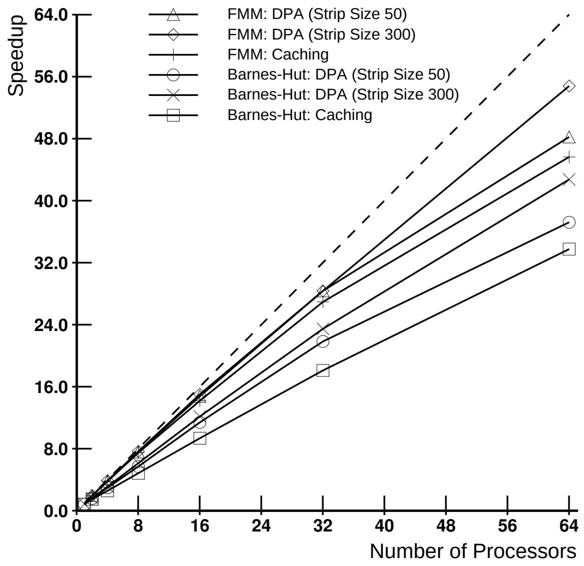


Figure 15: Execution time breakdown into idle time, communication overhead, and local computation for a message passing implementation of Barnes-Hut (16,384 particles) and FMM (32,768 particles, 29 terms) using DPA (strip size 300) on 16 nodes, with speedups shown on top of each bar. `Base` is DPA with tiling only, `Pipe x` adds message pipelining with `x` outstanding requests, and `Block y` adds message aggregation of `y` messages.

accesses where data access patterns are inter-dependent with computation, such as in *Barnes-Hut*. We are not aware of any other approach that enables automatic message aggregation for such computations.

In the context of dynamic locality optimizations, DPA requires inherent program concurrency, similar to multithreading [10], but unlike caching [3, 14] and prefetching [23]. DPA is a specialized form of multithreading with explicit data renaming to integrate data reuse. In addition, DPA uses additional memory to hold outstanding thread states, while caching uses memory for data. Because DPA reorders both computation and data accesses, it is complementary to both caching and prefetching for explicitly managing fast storage.

In the context of other compilation techniques, loop transformations [25], such as loop splitting, can decouple dependence to improve the effectiveness of DPA, while DPA also applies to loop optimizations when precise static access pattern or iteration space is unavailable. Our partitioning algorithm is similar to the dependence sets partitioning of [33, 36] for compiling for TAM [10]. Our use of aliasing to hoist data accesses enables larger threads. DPA is also an example optimization enabled by various aliasing and parallelization analyses such as, connection analysis [17] and commutativity analysis [30], which can help to enable DPA on sequential



BARNES-HUT							
	PROCESSORS						
Version	1	2	4	8	16	32	64
DPA (50)	118.02	61.23	33.05	17.15	8.59	4.48	2.63
DPA (300)	115.17	60.10	31.46	15.95	8.05	4.15	2.29
Caching	115.15	65.77	38.02	20.21	10.46	5.41	2.90

FMM						
	PROCESSORS					
Version	2	4	8	16	32	64
DPA (50)	7.39	3.80	1.91	0.98	0.51	0.30
DPA (300)	7.40	3.73	1.89	0.99	0.51	0.26
Object Caching	7.43	3.82	1.94	1.02	0.54	0.32

Figure 16: Barnes-Hut and FMM speedup and execution time (in seconds) comparisons among DPA with strip sizes 50 and 300 and object caching on the T3D. The sequential version execution time for Barnes-Hut is 97.84 secs (16,384 particles, 4 steps), for FMM is 14.46 secs (32,768-particle, 29 terms, 1 step).

languages, such as C or C++. Finally, DPA is motivated by manual communication optimizations for Barnes-Hut in [37].

Two limitations of our current implementation require source level modification of our codes. Because we have yet to integrate DPA with loop parallelization, we manually split a loop in Barnes-Hut to decouple dependence. In addition, the SPLASH FMM version modified the original algorithm by using remote writes to eliminate an interaction list. Because we do not optimize remote writes, we restored the original list so that only local writes occur. We are currently addressing both limitations.

7 Conclusion and Future Work

We presented DPA, a technique using dynamic reordering to enable tiling and communication optimizations to improve both data reuse and latency tolerance of reads to general PBDSs. Our results show that DPA achieves both good absolute performance and speedup for two hierarchical N-body programs by enabling tiling as well as significantly reducing idle time and communication overhead. The encouraging result is that such an approach can be effective relying on only coarse-grained aliasing and concurrency information. As future work, we are extending our thread partitioning to include multiple remote object references, and investigating locality admission tests for sequential cache optimization. We are also integrating DPA with loop optimizations, caching and multithreading as well as extending DPA to updates to handle a broader class of applications.

Acknowledgments

We thank Julian Dolby, Vijay Karamcheti, and Scott Pakin for critiquing the paper and providing invaluable comments. The Concert System has been the work of Andrew Chien, Hao-Hua Chu, Julian Dolby, Bishwaroop Ganguly, Vijay Karamcheti, John Plevyak, and Xingbin Zhang. We also thank the reviewers for their valuable comments.

The research described in this paper was supported in part by DARPA Order #E313 through the US Air Force Rome Laboratory Contract F30602-96-1-0286, NSF grant MIP-92-23732, ONR grants N00014-92-J-1961 and N00014-93-1-1086, NASA grant NAG 1-613, and supercomputing resources at the Jet Propulsion Laboratory. Support from Intel Corporation, Tandem Computers, Hewlett-Packard, and Motorola is also gratefully acknowledged. Andrew Chien is supported in part by NSF Young Investigator Award CCR-94-57809.

References

[1] Jennifer M. Anderson and Monica S. Lam. Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1993.

[2] Thomas Ball and Susan Horwitz. Constructing control flow from control dependence. Technical Report 1091, University of Wisconsin–Madison, 1992.

[3] Martin C. Carlisle and Anne Rogers. Software caching and computation migration in olden. In *Proceedings of Fifth Symposium on Principles and Practice of Parallel Programming*, 1995.

[4] Steve Carr, Kathryn S. McKinley, and Chau-Wen Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, 1994.

[5] K. Mani Chandy and Carl Kesselman. Compositional C++: Compositional parallel programming. In *Proceed-*

ings of the Fifth Workshop on Compilers and Languages for Parallel Computing, New Haven, Connecticut, 1992. YALEU/DCS/RR-915, Springer-Verlag Lecture Notes in Computer Science, 1993.

[6] A. A. Chien, U. S. Reddy, J. Plevyak, and J. Dolby. ICC++ – a C++ dialect for high performance parallel computing. In *Proceedings of the 2nd International Symposium on Object Technologies for Advanced Software*. Springer-Verlag, LNCS 742, 1996.

[7] Andrew Chien, Julian Dolby, Bishwaroop Ganguly, Vijay Karamcheti, and Xingbin Zhang. Supporting high level programming with high performance: The Illinois Concert system. In *Proceedings of the Second International Workshop on High-level Parallel Programming Models and Supportive Environments*, April 1997.

[8] Andrew Chien and Uday Reddy. ICC++ language definition. Concurrent Systems Architecture Group Memo, Also available from http://www-csag.cs.uiuc.edu/, February 1995.

[9] Cray Research, Inc. *Cray T3D System Architecture Overview*, March 1993.

[10] David Culler, Anurag Sah, Klaus Erik Schauser, Thorsten von Eicken, and John Wawrzynek. Fine-grain parallelism with minimal hardware support: A compiler-controlled threaded abstract machine. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages an Operating Systems*, pages 164–75, 1991.

[11] David E. Culler. *Managing Parallelism and Resources in Scientific Dataflow Programs*. PhD thesis, Massachusetts Institute of Technology, Laboratory of Computer Science, Cambridge, MA, June 1989.

[12] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. An efficient method of computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[13] Julian Dolby. Automatic inline allocation of objects. In *Proceedings of the 1997 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 1997.

[14] B. Falsafi, A. Lebeck, S. Reinhardt, I. Schoinas, M. Hill, J. Larus, A. Rogers, and D. Wood. Application-specific protocols for user-level shared memory. In *Proceedings of Supercomputing '94*, Washington, D.C., 1994.

[15] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–49, July 1987.

[16] High Performance Fortran Forum. High performance Fortran language specification version 1.0. Technical Report CRPC-TR92225, Rice University, January 1993.

[17] Rakesh Ghiya and Laurie J. Hendren. Connection analysis: A practical interprocedural heap analysis for C. In *Proceedings of the Workshop for Languages and Compilers for Parallel Computing*, 1995.

[18] Rakesh Ghiya and Laurie J. Hendren. Is it a tree, a DAG, or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Proceedings of POPL*, 1996.

[19] L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):35–47, January 1990.

[20] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for FORTRAN D on MIMD distributed-memory machines. *Communications of the ACM*, August 1992.

[21] James Philbin, et. al. Thread scheduling for cache locality. In *Proceedings of the Seventh Symposium on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, 1996.

[22] Vijay Karamcheti and Andrew A. Chien. FM—fast messaging on the Cray T3D. Available from `http://www-csag.cs.uiuc.edu/papers/t3d-fm-manual.ps`, February 1995.

[23] Chi-Keung Luk and Todd. C. Mowry. Compiler-based prefetching for recursive data structures. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages an Operating Systems*, pages 62–73, 1996.

[24] Kathryn S. McKinley. Evaluating automatic parallelization for efficient execution on shared-memory multiprocessors. In *Proceedings of the International Conference on Supercomputing*, 1994.

[25] D. Padua and M. Wolfe. Advanced compiler optimizations for supercomputers. *Communications of the ACM*, 29(12), 1986.

[26] Scott Pakin, Vijay Karamcheti, and Andrew A. Chien. Fast Messages (FM): Efficient, portable communication for workstation clusters and massively-parallel processors. *IEEE Concurrency*, to appear in 1997. Available from `http://www-csag.cs.uiuc.edu/papers/fm-pdt.ps`.

[27] John Plevyak and Andrew A. Chien. Precise concrete type inference of object-oriented programs. In *Proceedings of OOPSLA'94, Object-Oriented Programming Systems, Languages and Architectures*, pages 324–340, 1994.

[28] John Plevyak and Andrew A. Chien. Type directed cloning for object-oriented programs. In *Proceedings of the Workshop for Languages and Compilers for Parallel Computing*, pages 566–580, 1995.

[29] William T. Rankin and John A. Board Jr. A portable distributed implementation of the parallel multiple tree algorithm. Technical Report 95-002, Duke University, Department of Electrical Engineering, 1995.

[30] Martin Rinard and Pedro C. Diniz. Commutativity analysis: A new analysis framework for parallelizing compilers. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 54–67, 1996.

[31] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel and Distributed Computing*, 8:303–312, 1990.

[32] Vivek Sarkar and Radhika Thekkath. A general framework for iteration-reordering loop transformations. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1992.

[33] Klaus Erik Schauser, David E. Culler, and Thorsten von Eicken. Compiler-controlled multithreading for lenient parallel languages. In *Proceedings of the Conference on Functional Programming and Computer Architecture*, 1991.

[34] Jaswinder Pal Singh. *Parallel Hierarchical N-Body Methods and Their Implications For Multiprocessors*. PhD thesis, Stanford University Department of Computer Science, Stanford, CA, February 1993.

[35] Jaswinder Pal Singh, Chris Holt, John L. Hennessy, and Anoop Gupta. A parallel adaptive fast multipole method. In *Proceedings of Supercomputing Conference*, pages 54–65, 1993.

[36] Kenneth R. Traub, David E. Culler, and Klause E. Schauser. Global analysis for partitioning non-strict programs into sequential threads. In *Proc. 1992 ACM Conference on Lisp and Functional Programming*, June 1992.

[37] M. Warren and J. Salmon. A parallel hashed oct-tree N-body algorithm. In *Proceedings of Supercomputing Conference*, pages 12–21, 1993.

[38] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the International Symposium on Computer Architecture*, pages 24–36, 1995.