A Compiler Optimization Algorithm for Shared-Memory Multiprocessors

Kathryn S. McKinley

July 1, 1998

Abstract

This paper presents a new compiler optimization algorithm that parallelizes applications for symmetric, shared-memory multiprocessors. The algorithm considers data locality, parallelism, and the granularity of parallelism. It uses dependence analysis and a simple cache model to drive its optimizations. It also optimizes across procedures by using interprocedural analysis and transformations. We validate the algorithm by hand-applying it to sequential versions of parallel, Fortran programs operating over dense matrices. The programs initially were hand-coded to target a variety of parallel machines using loop parallelism. We ignore the user's parallel loop directives, and use known and implemented dependence and interprocedural analysis to find parallelism. We then apply our new optimization algorithm to the resulting program. We compare the original parallel program to the hand-optimized program, and show that our algorithm improves 3 programs, matches 4 programs, and degrades 1 program in our test suite on a shared-memory, bus-based parallel machine with local caches. This experiment suggests existing dependence and interprocedural array analysis can automatically detect user parallelism, and demonstrates that user parallelized codes often benefit from our compiler optimizations, providing evidence that we need *both* parallel algorithms and compiler optimizations to effectively utilize parallel machines.

Index Terms—Program parallelization, parallelization techniques, program optimization, data locality, restructuring compilers, performance evaluation.

In *IEEE Transactions on Parallel and Distributed Systems*, VOL. 9, NO. 8, August, 1998.

K. McKinley is with the Department of Computer Science, University of Massachusetts, Amherst, MA 01003-4610. E-mail: mckinley@cs.umass.edu.

1 Introduction

One lesson from vectorization is that users rewrote programs based on feedback from vectorizing compilers. These rewritten programs were independent of any particular vector hardware and were written in a style amenable to vectorization. Compilers were then able to generate machine-dependent vector code with excellent results. We believe that just as automatic vectorization was not successful for dusty deck programs, automatic parallelization of dusty decks is unlikely to yield complete success. Finding medium to large grain parallelism is more difficult than single statement parallelism and compilers have had limited success on dusty deck programs [10, 19, 42, 43, 17]. We believe that only a combination of algorithmic parallelization and compiler optimization will enable the effective use of parallel hardware. This paper provides evidence that this approach is viable.

Since the amount of parallelism a dusty deck program contains is unknown, measuring the success of parallelizing compilers on them is tenuous. The programs may actually be inherently sequential, parallel, or somewhere in between. Since a different version of the algorithm could potentially achieve linear speed-up, only linear speed-ups (performance improvements that scale with the number of processors) can be declared a complete success. Linear speed-up is rare, even for parallel applications due to communication overheads and Amdahl's Law. In practice, parallel programs often require algorithms and data structures that differ from their sequential and vector counterparts. To achieve good parallel performance, the intellectual and programming costs required for good parallel performance need to be paid. Our ultimate goal is to provide compiler technology that detects and exploits parallelism on a variety of parallel machines, so that the programming cost will only have to be paid once. Users will concentrate on parallel algorithms at a high level. The compiler will be responsible for machine-dependent details such as exploiting the memory hierarchy. In this paper, we consider optimizing Fortran programs for symmetric shared-memory, bus-based parallel machines with local caches.

We present an advanced parallelizing algorithm for complete applications that exploits and balances data locality, parallelism, and the granularity of parallelism. The algorithm uses existing dependence analysis techniques and a new cache model to drive the following loop optimizations: loop permutation, tiling, fusion, and distribution. It tries to organize the computation such that each processor achieves data locality, accessing only the data in its own cache, and such that the granularity of parallelism is large. Since we find that large grain parallelism often crosses procedure boundaries, the algorithm uses existing interprocedural analysis and new interprocedural optimizations. The main advantage of our optimization algorithm is that it yields good results, and is polynomial with respect to loop nesting depth (it does however use dependence analysis which is, of course, more expensive). In this paper, we present a new parallelizing algorithm, but clearly good analysis is a prerequisite to this work and we discuss the analysis that enables our parallelization algorithm to succeed. We present our parallelization algorithm in detail in Section 3.

To test the algorithm, we performed the following experimental study. We collected Fortran programs written for a variety of parallel machines. Most of the programs in our suite are published versions of state-of-the-art parallel algorithms. The programs use parallel loops and two also use critical sections. We created sequential versions of each program by converting the parallel loops to sequential loops and by eliminating the critical sections. The algorithm then optimized this version. We do not recommend that users convert their programs to serial versions before handing it to the compiler, but we use this methodology to assess the ability of the compiler to find, exploit, and optimize known parallelism. We used ParaScope [12, 27], an interactive parallelization tool, to systematically apply the transformations in the algorithm to the sequential programs. ParaScope implements dependence analysis, interprocedural analysis, and safe application of the loop transformations (tiling, interchange, fusion, and distribution), but not the interprocedural optimizations, nor the optimization algorithm itself. Sections 4 and 5 detail this experiment.

In Section 6, we show that the algorithm matched or improved the performance of seven of nine programs on a shared-memory, bus-based parallel machine with local caches. In addition to dependence analysis, many of the programs require interprocedural and symbolic analysis to find parallel loops. We also analyze which parts of the algorithm are responsible for the improvements. Most of the improvements occur in cases where data

locality and parallelism intertwine. The programmers were not able to exploit both when they conflict, but our algorithm does. This result suggests a combination of algorithmic parallelism and compiler optimization will yield the best performance.

To explore whether a machine-independent parallel programming style exists, we also examined programming styles in light of the algorithm's successes and failures in Section 6. We found that for the most part, these parallel programmers use a clean, modular style that is amenable to compiler analysis and optimization. Although the test suite is small, we believe this result adds to the evidence that programmers can write portable, parallelizable programs for scientific applications from which compilers can achieve very good performance.

The remainder of this paper is organized as follows. Section 2 briefly reviews the technical background. Section 3 describes the parallelization algorithm. Section 5 presents our experimental framework and the program test suite and its characteristics. Section 4 measures the effectiveness of our algorithm at parallelizing and optimizing the programs in our test suite. Section 7 compares our work to other research in this area, and Section 8 summarizes our approach and results.

2 Technical Background

This section overviews the technical background on dependence and reuse that is need to understand the parallelization algorithm.

Data Dependence. We assume the reader is familiar with *data dependence* [18, 29]. Throughout the paper, $\vec{\delta} = \{\delta_1 \ldots \delta_k\}$ represents a hybrid direction/distance vector for a data dependence between two array references. Each entry in the vector describes the distance or direction in loop iterations between references to the same location. Dependence vectors are written left to right from the outermost to innermost loop enclosing the references. Data dependences are *loop-independent* if the references to the same memory location occur in the same loop iteration and *loop-carried* if they occur on different iterations. *Parallel loops* have no loop-carried dependences and *sequential loops* have at least one.

Sources of Data Reuse. The two sources of data reuse are *temporal* reuse, multiple accesses to the same memory location, and *spatial* reuse, accesses to nearby memory locations that share a cache line or a block of memory at some level of the memory hierarchy. (Spatial reuse is sometimes referred to as stride 1 or unit stride access.) Spatial reuse may result from *self-reuse*, consecutive accesses by the same array reference to the same cache line, or from *group-reuse*, multiple array references accessing the same cache line. Similarly, temporal reuse may arise from multiple accesses to the same memory location by a single array reference or by multiple array references. Without loss of generality, we assume Fortran's column-major storage.

Augmented Call Graph. We use an *augmented call graph* G_{ac} to describe the calling relationships among procedures and loop nest structures in the program [20]. This flow-insensitive call graph contains procedure nodes and call nodes. For each procedure p that makes a procedure call at site s, an edge connects node p to node s. For each call site s to procedure q, an edge connects node s to node q. The G_{ac} also adds loop nodes for every loop and edges to represent nesting. For each outer loop l in procedure p, the G_{ac} contains an edge from node p to node l. An inner loop is also connected to its outer loop with an edge. If loop l in procedure p surrounds a call to a procedure q, the usual edge from node p to the call to q is replaced by an edge from node p to node l and an edge from node l to the call node q. If an outer loop surrounds all the other statements in a procedure, it is marked *enclosing*. Note that call and loop nodes will have only one predecessor, but procedure nodes may have multiple predecessors. For example, Figure 1(b) illustrates the G_{ac} for the program in Figure 1(a).

3 The Parallelization Algorithm

This section describes a new algorithm for parallelizing programs. The algorithm is unique in its ability to exploit both data locality and parallelism, to increase the granularity of parallelism, and to optimize across procedure boundaries. Section 3.1 begins by presenting the basic structure of the driver, an overview of each component of

Figure 1: Two Adjacent Calls to *dmxpy* from *Linpackd*

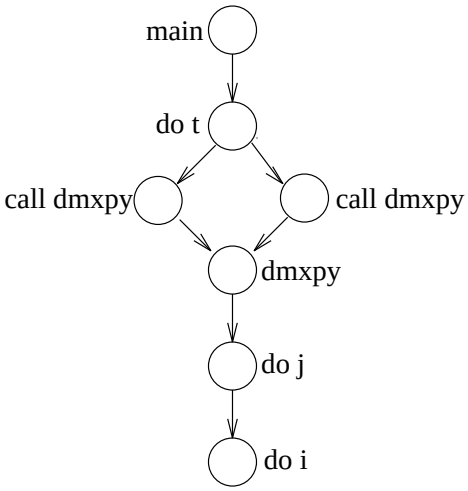
(a) original program

```
subroutine dmxpy(n1, y, n2, ldm, x, m)
double precision y(*), x(*), m(ldm, *)

do j = 1, n2
      do i = 1, n1
            y(i) = y(i) + x(j) * m(i,j)
      enddo
enddo

program main
do t = 1, timesteps
      . . .
      call dmxpy(n1, y, n2, ldm, x, m)
      call dmxpy(n1, a, n3, ldm, b, r)
enddo
```

(b) Augmented Call Graph (G_{ac})



(c) optimized kernel

```
subroutine dmxpy(n1, y, n2, ldm, x, m)
double precision y(*), x(*), m(ldm, *)
parallel do ii = 1, n1, tile
      do j = 1, n2
            do i = ii, min(ii + tile - 1,n1)
                  y(i) = y(i) + x(j) * m(i,j)
            enddo
      enddo
end parallel do

program main
do t = 1, timesteps
      . . .
      call dmxpy(n1, y, n2, ldm, x, m)
      call dmxpy(n1, a, n3, ldm, b, r)
enddo
```

(d) optimized program

```
subroutine dmxpyE(n1, y, n2, ldm, x, m, ii, tile)
double precision y(*), x(*), m(ldm, *)
do j = 1, n2
      do i = ii, min(ii + tile - 1,n1)
            y(i) = y(i) + x(j) * m(i,j)
      enddo
enddo

program main
do t = 1, timesteps
      . . .
      parallel do ii = 1, n1, tile
            call dmxpyE(n1, y, n2, ldm, x, m, ii, tile)
            call dmxpyE(n1, a, n3, ldm, b, r, ii, tile)
      end parallel do
enddo
```

the algorithm, and an example. The subsequent sections then detail the individual components of the algorithm.

3.1 Driver

The basic structure of the driver for the parallelization algorithm appears in Figure 2. The *Driver* algorithm proceeds top down recursively from the root of the G_{ac}, such that a node n is only visited once all its predecessors have been visited. All nests and procedures nodes are initialized to *unvisited* and *unoptimized*. Notice that *Driver* is fairly specialized to the node type. For a call node, it does book keeping for *visited* and *optimized* and then recurses further down the call chain. Notice the second *if* statement. If all the predecessors of a procedure p are marked *optimized*, then parallelism has been introduced in all the calling contexts and the procedure is not optimized further. Otherwise for a procedure, it collects the outer loop nodes that are adjacent, and calls *Parallelize*. *Parallelize* uses a variety of transformations that act individually and collectively on the nests. If *Parallelize* does not introduce parallelism, *Driver* recurses further down the call chain. This algorithm performs

Figure 2: **Driver**: Driver for Parallelization Algorithm

INPUT: n is a procedure or call node in the G_{ac}, loop nodes are handled within *Driver*
ALGORITHM:
procedure Driver(n)
if n is a procedure and any predecessor of n is not visited **return**
mark n *visited*
if n is a procedure and all predecessors of n are marked *optimized* **return**
if n is a call node to procedure p **Driver** (p) {skip over call nodes}
if n is a procedure node
partition the outer loops nodes $\mathcal{L} = \{l_1, \ldots, l_k\}$ of n into sets of adjacent outer loops
$\mathcal{R}_i = \{\{l_1, \ldots, l_j\}, \ldots, \{l_r, \ldots, l_k\}\}$
forall i **Parallelize**($\mathcal{R}_i$)
forall l_k
if l_k now contains a parallel loop
mark it and call nodes nested within the parallel loop *optimized* and *visited*
forall s, call nodes nested in l_k not surrounded by a parallel loop **Driver**(s)
endforall
endif

only intraprocedural loop transformations. In Section 3.5, we extend *Driver* to produce multiple optimized versions of a procedure for different calling contexts, and handle interprocedural transformations. For purposes of explanation, we divide the components of our algorithm into the following steps.

Optimize - uses loop permutation and tiling on a single nest to exploit data locality and parallelism.

Fuser - performs loop fusion and distribution to enable *Optimize* on a single nest and to increase the granularity of parallelism across multiple nests.

Parallelize - combines *Optimize* and *Fuser* resulting in an effective intraprocedural parallelization algorithm for loop nests.

Enabler - uses interprocedural analysis and transformations to enable the *Parallelize* to be applied across procedure calls. In particular, it parallelizes and optimizes loop nests containing calls and spanning calls. It uses the interprocedural transformations (*loop embedding*, *loop extraction*, and *procedure cloning*) as needed to enable loop transformations.

Example. To introduce the algorithm, we overview how the algorithm optimizes the program in Figure 1(a) for a shared-memory multiprocessor. Ideally, we want to organize the computation such that each processor only uses data in its private cache. The computation would thus exhibit locality, and would never need to get data from main memory or another processor's cache. To achieve locality, we want the references in the loop to the same memory location or adjacent locations to occur within a short period of time. The shorter the period of time, the more likely the cache line on which the item resides will still be in the cache. In *dmxpy* in Figure 1(a), the references to $y(i)$ have spatial locality on the i loop and invariant, temporal locality on the j loop, the reference $x(j)$ has spatial locality on the j loop and temporal invariant locality on the i loop, and the reference $m(i, j)$ has spatial locality on the i loop and no locality on the j loop. The original loop ordering, with the i loop in the inner position varying most rapidly, thus achieves temporal or spatial locality for all the references.

Parallelism is usually most effective when it achieves the largest possible *granularity*, the largest amount of work per parallel task. In *dmxpy* in Figure 1(a), the outer j loop carries a recurrence and the inner i loop is

parallel. Maximizing the granularity of this nest thus prefers the i loop in the outer position, but as we showed above, exploiting locality prefers i in the inner position. If we choose to permute the i loop to the outer position and parallelize it, we can ruin locality in two ways:

1. If the runtime system assigns adjacent i iterations to different processors, multiple processors will share cache lines. For example, each processor will update one element of $y(i)$, causing consistency traffic between the local caches that share $y(i)$ and memory. This effect is called *false sharing*, since each processor is actually only using and updating independent elements in each cache line. This effect can dramatically degrade performance [24].

2. Even if the runtime system assigns adjacent elements of $y(i)$ to the same processor, each pair of processors may share cache lines. Also, because the reuses of $y(i)$ are further apart in time than with an inner i loop, it is more likely that $m(i, j)$ or $x(j)$ will map to the same cache line causing additional cache misses. This effect also degrades performance [24].

To achieve locality and parallelism, we thus strip-mine and interchange the i loop as shown in Figure 1(c). The outer ii loop is parallel, and the inner i loop still attains spatial and temporal locality given a large enough tile. The *Optimize* algorithm drives this process with a cache model.

Notice also that in *main* there are two adjacent, independent calls to *dmxpy*. In Figure 1(c), processors must wait for the previous call to *dmxpy* to complete before proceeding. Our analysis uses the G_{ac} and interprocedural array section analysis to detect independence and loop structure. The algorithm *Enabler* then extracts the outer loop from each call and *Fuser* puts them into one loop, as illustrated in Figure 1(d). This transformation eliminates barrier synchronization and can also improve locality. The next section describes each of these steps in detail.

3.2 Optimize: Data Locality and Parallelism

The most effective and essential component of our parallelization algorithm uses a simple memory model to drive optimizations for data locality and parallelism [34, 24]. We employ loop permutation to improve data locality and tiling to introduce parallelism. Using a memory model and loop transformations, our algorithm places the loops with the most reuse innermost and parallel loops outermost, where each is most effective. It also balances tradeoffs between the two when they conflict.

To simplify locality analysis, we concentrate on reuse that occurs between small numbers of inner loop iterations. Our memory model assumes there will be no conflict or capacity cache misses in one iteration of the innermost loop.[1] The algorithm performs the following five steps.

1. It puts array references that exhibit group-temporal and/or group-spatial locality into the same *reference group*.
2. It determines the cost of loop nest organizations in terms of the number of cache lines accessed.
3. It determines *memory order*, the permutation of the loops in the nest that yields the best data locality in terms of the fewest cache lines accessed.
4. It achieves memory order or a *nearby* loop nest order through loop permutation.
5. It introduces outer loop parallelism by *tiling* the nest to to maintain locality on individual processors as the computation is divided among multiple processors. The tiling step strip-mines and if necessary, permutes the nest to position an outermost parallel loop.

[1] McKinley and Temam support this assumption [35], and McKinley et al. demonstrate that this memory model works well for uniprocessor caches [34].

Figure 3: **Optimize**: Data Locality and Parallelization Algorithm

INPUT: A loop nest $\mathcal{L} = \{l_1, \ldots, l_k\}$
OUTPUT: An optimized loop nest $\mathcal{P}$
ALGORITHM:

```
procedure Optimize(L)
    compute RefGroup for all references in L
    MO = MemoryOrder(L)
    P = NearbyPermutation(L, MO)
    for j = 1, m { outermost to innermost loop of P }
      if p_j parallel
        strip mine p_j and parallelize r_j, the resulting outer loop
        if (j ≠ 1) permute r_j into the outermost legal position in P
        mark nest optimized
        break
      endif
    endfor
```

The first four steps determine the amount of reuse for the nest considering each loop as if it were innermost. Based on this measure, the algorithm then permutes the nest to achieve the lowest possible cost over the entire nest while preserving correctness. Figure 3 contains the procedural version of the *Optimize* algorithm, which we explain in detail below.

3.2.1 Reference Groups

The goal of the *RefGroup* algorithm is to avoid over counting cache lines accessed by multiple references that generally access the same set of cache lines. *RefGroup* finds references with group-spatial and group-temporal locality with respect to a candidate inner loop and places them into *reference groups*. For every loop l in the nest, it considers l as a candidate for the innermost position.

RefGroup: References Ref_1 and Ref_2 belong to the same reference group with respect to loop l if:

1. $\exists$ $Ref_1 \ \vec{\delta} \ Ref_2$, and
   (a) $\vec{\delta}$ is a loop-independent dependence, or
   (b) δ_l is a small constant d ($|d| \leq 2$) and all other entries are zero,
2. or, Ref_1 and Ref_2 refer to the same array and differ by at most d' in the first subscript dimension, where d' is less than or equal to the cache line size in terms of array elements. All other subscripts must be identical.

Condition 1 accounts for group-temporal reuse and condition 2 detects most forms of group-spatial reuse. Note that a reference can only belong to one group.

3.2.2 Loop Cost in terms of Cache Lines

Step 2 determines the cost in cache lines of each reference group. Using a representative array reference from each group, the algorithm *LoopCost* in Figure 4 determines for each candidate inner loop, the number of cache lines the reference will access. Intuitively, given a candidate inner loop l with *trip* iterations and a cache line size *cls* in array elements, an array reference is classified and assigned a cost as follows.

Loop invariant - (temporal locality) if the subscripts of the reference do not vary with l, then it requires only one cache line for all iterations of l (these references should end up in registers). Loop invariant references have temporal locality.

Figure 4: **LoopCost**: LoopCost Algorithm

INPUT:

$$\begin{aligned}
\mathcal{L} &= \{l_1, \ldots, l_n\} \text{ a loop nest with headers } lb_l, ub_l, \textit{step}_l \\
\mathcal{R} &= \{\textit{Ref}_1, \ldots, \textit{Ref}_m\} \text{ representatives from each reference group} \\
trip_l &= (ub_l - lb_l + \textit{step}_l)/\textit{step}_l \\
\textit{cls} &= \text{the cache line size,} \\
\textit{coeff}(f, i_l) &= \text{the coefficient of the index variable } i_l \text{ in the subscript } f \\
\textit{stride}(f_1, i_l, l) &= |\ \text{step}_l * \textit{coeff}(f_1, i_l)|
\end{aligned}$$

OUTPUT:

$$\textit{LoopCost}(l) = \text{number of cache lines accessed with } l \text{ as innermost loop}$$

ALGORITHM:

$$\textbf{LoopCost}(l) = \sum_{k=1}^{m} (\textbf{RefCost}(\textit{Ref}_k(f_1(i_1, \ldots, i_n), \ldots, f_j(i_1, \ldots, i_n)), l) \prod_{h \neq l} \textit{trip}_h$$

$$\textbf{RefCost}(\textit{Ref}_k, l) = \begin{cases}
1 & \textit{if } ((\textit{coeff}(f_1, i_l) = 0) \wedge \ldots \wedge (\textit{coeff}(f_j, i_l) = 0)) \quad \textbf{Invariant} \\
\dfrac{\textit{trip}_l}{\left(\frac{\text{cls}}{\textit{stride}(f_1, i_l, l)}\right)} & \textit{if } ((\textit{stride}(f_1, i_l, l) < \textit{cls}) \wedge (\textit{coeff}(f_2, i_l) = 0) \wedge \ldots \wedge (\textit{coeff}(f_j, i_l) = 0)) \quad \textbf{Unit} \\
\textit{trip}_l & \textit{otherwise} \quad \textbf{None}
\end{cases}$$

Consecutive - (spatial locality) if only the first subscript dimension (the column) varies with l, then it requires a new cache line every *cls* iterations, resulting in *trip/cls* number of cache lines accessed. (The algorithm in Figure 4 adjusts for non-unit strides less than the cache line size.) Consecutive references have spatial locality.

No Reuse – if the subscripts vary with l in any other manner, then the array reference is assumed to require a different cache line every iteration, yielding a total of *trip* number of cache lines accessed.

To determine the *reference cost* over the entire nest when loop l is innermost, *LoopCost* multiplies the above cost by the trip counts of the remaining loops. These loops would enclose l if l is innermost. Since *LoopCost* only measures reuse in the innermost loop, the order of the remaining loops does not affect *LoopCost*. *LoopCost* then sums the cost over all the reference groups for a candidate inner loop l. The next section shows how we use *LoopCost* to find the best loop order for the entire nest. This method evaluates imperfectly nested loops, complicated subscript expressions, and loops with symbolic bounds.

Example. Consider again the subroutine *dmxpy* from *Linpackd* in Figure 5. In this example, the reference groups are the same for the i and j loops. Since there is only one reference to the arrays x and m, *RefGroup* place each in a reference group by itself. Since the two references to y are connected by a loop-independent dependence, they make a single group. We assume for the example that 4 elements of each array fit on a cache line. As illustrated by the table in Figure 5, the reference $y(i)$ has spatial locality and is thus consecutive in the i loop, and has temporal locality because it is invariant in the j loop. The reference $x(j)$ has spatial locality on the j loop and has temporal invariant locality on the i loop. The reference $m(i, j)$ has spatial locality on the i loop and has no reuse on the j loop. Notice when the i loop is the candidate inner loop, the j loop must be the outer loop and therefore *LoopCost* multiplies the reference costs by $n2$, the j loop's trip count. Similarly, when j is the candidate inner loop, *LoopCost* multiplies the reference costs by $n1$, the i loop's trip count.

Figure 5: Subroutine *dmxpy* from *Linpackd*

```
do j = 1, n2
      do i = 1, n1
            y(i) = y(i) + x(j) * m(i,j)
      enddo
enddo
```

Cost in Cache Lines, *cls = 4*

reference group	candidate inner loop	
	loop i	loop j
y(i)	1/4 n1 $*$ n2	1 $*$ n1
x(j)	1 $*$ n2	1/4 n2 $*$ n1
m(i,j)	1/4 n1 $*$ n2	n2 $*$ n1
loop cost	1/2 n1 $*$ n2 + n2	5/4 n1 $*$ n2 + n1

3.2.3 Memory Order

Even though *LoopCost* does not directly measure reuse across outer loops, we can use it to determine the loop permutation for the entire nest which accesses the fewest cache lines by relying on the following observation:

> *If loop l promotes more reuse than loop l' when both are considered for the innermost loop, l will promote more reuse than l' at any outer loop position.*

We therefore simply rank the loops using their loop cost, ordering the loops from outermost to innermost $\{l_1 \ldots l_n\}$ such that the loop cost of l_i is less than or equal to l_{i-1}. We call this permutation of the nest with the least cost *memory order*. Although contrived counterexamples exist to the above observation for 3 or more levels of loop nesting, previous work demonstrates that in practice, the model is extremely accurate and always gets the inner loop right [34]. We assume that each of the loop bounds is greater than 1, which is the only interesting case. If the constants are comparable with the number of cache items on a line, this model loses accuracy. In this case however, reuse across outer loops is likely, and thus this algorithm still produces good results. If the bounds are symbolic, we compare the dominating terms. If the dominating term is a function of all loop bounds, then regardless of the relative sizes of the loop bounds, the model is accurate.

Example. Assuming $n1, n2 > 1$ in *dmxpy*, loop i accesses fewer cache lines than j and should be placed innermost, yielding a memory order of $\{l_j, l_i\}$.

3.2.4 Achieving Memory Order

Memory order specifies the permutation of the nest with the least cost. To determine if the order is a legal one for a perfect nest,[2] we permute the corresponding entries in the distance/direction vector. If the result is lexicographically positive (the majority of the time it is [34]), the permutation is legal and we transform the nest. If not, we use the algorithm *NearbyPermutation* in Figure 6.

Given a memory ordering $\mathcal{L} = \{i_{\sigma_1}, i_{\sigma_2}, \ldots, i_{\sigma_n}\}$ of the loops $\{i_1, i_2, ..., i_n\}$ where i_{σ_1} has the least reuse and i_{σ_n} has the most, the algorithm builds up a legal permutation in $\mathcal{P}$ by first testing to see if the loop i_{σ_1} is legal in the outermost position. If it is legal, it is added to $\mathcal{P}$ and removed from $\mathcal{L}$. If it is not legal, the next loop in $\mathcal{L}$ is tested. Once a loop l is positioned, the process is repeated starting from the beginning of $\mathcal{L} - \{l\}$ until $\mathcal{L}$ is empty. The following theorem holds for the *NearbyPermutation* algorithm.

> **Theorem:** *If there exists a legal permutation where σ_n is the innermost loop, then* NearbyPermutation *will find a permutation where σ_n is innermost.*

[2] Determining memory order does not depend on a perfect nest. Methods exist for permuting imperfect nests [49], but we only permute perfect nests or nests that fusion or distribution make perfect (see Section 3.3.1).

Figure 6: **NearbyPermutation**: NearbyPermutation Algorithm

INPUT:
$\mathcal{O}$ $= \{i_1, i_2, ..., i_n\}$, the original loop ordering
$\mathcal{L}$ = $\{i_{\sigma_1}, i_{\sigma_2}, \ldots, i_{\sigma_n}\}$, a permutation of $\mathcal{O}$

OUTPUT:
$\mathcal{P} = \{p_1, \ldots, p_n\}$ a nearby permutation of $\mathcal{O}$

ALGORITHM:
procedure NearbyPermutation ($\mathcal{O}$, $\mathcal{L}$)
$\mathcal{P} = \emptyset$; $k = 0$; $m = n$
while $\mathcal{L} \neq \emptyset$
for $j = 1, m$
$l = i_{\sigma_j} \in \mathcal{L}$ { l is the j^{th} loop in $\mathcal{L}$}
if direction vectors for $\{p_1, \ldots, p_k, l\}$ are legal
$\mathcal{P} = \{p_1, \ldots, p_k, l\}$
$\mathcal{L} = \mathcal{L} - \{l\}$; $k = k + 1$; $m = m - 1$
break for
endif
endfor
endwhile

The proof by contradiction of the theorem proceeds as follows. Given an original set of legal direction vectors, each step of the "for" is guaranteed to find a loop which results in a legal (lexicographically positive) direction vector, otherwise the original was not legal [3, 9]. In addition, if any loop σ_1 through σ_{n-1} may be legally positioned prior to σ_n, it will be.

NearbyPermutation therefore places the loops carrying the most reuse as innermost as possible. If the desired inner loop cannot be obtained, it places the next most desirable inner loop in the innermost position if possible, and so on. This characteristic is important because most data reuse occurs on the innermost loop(s), so positioning it correctly yields the best data locality.

3.2.5 Tiling for Parallelism

This step introduces a single level of outer loop parallelism, which is all the outer loop parallelism that typical bus-based shared memory parallel processors can effectively exploit. At this point in the algorithm, the nest is structured such that it accesses the fewest cache lines and accesses to the same cache line occur close together in time. In addition, if a loop carries temporal invariant or spatial locality, it has been identified. The two goals during the introduction of parallelism are:

1. To place a parallel loop in the outermost legal position, maximizing the granularity of parallelism.
2. If the parallel loop carries reuse, to tile it such that cache line reuse will fall locally on a processor, reducing or eliminating communication between processors.

The algorithm therefore selects a loop for parallelization which is either already parallelizable in the outermost position or if not, can be legally permuted and parallelized into an outermost position. If this loop carries either temporal or spatial reuse, the algorithm strip-mines it by *tile size*. Strip-mining is always safe and it produces two loops, a parallel outer *iterator* and an inner contiguous *strip*. If the iterator is not in the outermost position, the optimizer permutes it to the outermost legal position.[3] The algorithm strip-mines by the number of processors,

[3] Additional register and cache tiling for the individual processors should also be performed when applicable, but is beyond the scope of this paper.

assigning one iteration to each processor with the largest strips possible. We discuss this choice in more detail after an example.

Example. Consider *dmxpy* again. Note that only the i loop is parallel. It can be safely interchanged and parallelized in the outermost position. Since it carries temporal invariant and spatial locality, we tile the nest. The optimizer strip-mines the i loop by the number of processors, permutes the iterating loop to the outermost position, and parallelizes it. Figure 1(c) illustrates the result. Because this loop structure maximizes data locality, it reduces communication of data between iterations and therefore between processors. In experiments on the Sequent, this version of *dmxpy* results in speed-ups of up to 16.4 on 19 processors. This algorithm also attains linear speed-ups for kernels such as matrix multiply [24].

Previous work also experiments with different versions of *dmxpy*, and shows the version in Figure 1(c) is the best [24]. For example, parallelizing i in the outermost position and assigning adjacent i iterations to different processors instead causes the cache line for $y(i)$ to be shared among multiple processors. When compared with the version our algorithm produces, this sharing results in additional bus traffic when the line is sent to multiple processors instead of one processor, and additional coherence traffic since every write or an invalidate must go to all the caches which contain the line. Previous work demonstrates that these costs degrade performance significantly [24].

Constants and Tile Sizes. In the experiments reported in Section 4, the optimizer gives the parallel iterator one iteration for each processor, producing strips as large as possible. If the parallel loop carries spatial reuse and has enough iterations, reuse is attained and this strategy works well [24]. If the parallel loop has spatial locality and fewer iterations than the number of processors times the number of items on a cache line, then previous work shows that to achieve the best performance, we should actually reduce the number of parallel iterations such that the cache lines are not shared between processors [24]. A runtime test could differentiate these cases. In this work for unknown loop bounds of parallel loops, we assume the number of iterations is greater than the number processors times the cache line size.[4]

3.2.6 Summary and Discussion

To review, the complete *Optimize* algorithm appears in Figure 3. We first compute the *RefGroup* for all the references in the loop. Next we find memory order, and apply *NearbyPermutation* to achieve memory order when possible. The final step parallelizes the outermost parallel loop, and if necessary permutes it to the outermost position.

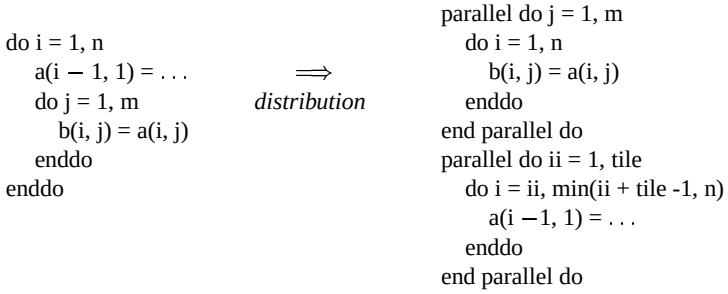
In our experiments, memory order is usually a legal permutation of the nest [34]. The complexity of the entire algorithm in this case is dominated by the time to sort the loops in the nest and the corresponding dependence vectors. The algorithm is thus $O(n\ log(n))$ in time to sort and linear in space, where n is the depth of the nest. In the worst case, when the desired outermost loop must be innermost, *NearbyPermutation*'s complexity dominates, $O(n^2)$ time. The parallelization step of the algorithm is linear. We have previously shown the data locality algorithm effective for uniprocessors [34]. We have also demonstrated that the parallelization algorithm effective for kernels [24], and in Section 4 we show that this algorithm is effective for application programs on shared-memory multiprocessors.

3.3 Fuser: Improving the Granularity of Parallelism

This section describes an approach for incorporating fusion and distribution into the *Optimize* algorithm. Loop fusion and distribution have several purposes in our algorithm. The foremost purpose is fusing parallel loops together to increase the granularity of parallelism and to reduce communication of shared data. Fusion and distribution may also create perfect nests which *Optimize* can improve.

[4] An alternative approach would give the runtime scheduler the flexibility to balance irregular work loads. For example, by making the strips the same size as the cache line, there would be more parallel iterations than processors, and thus the runtime scheduler could assign iterations to processors dynamically.

Figure 7: Loop Distribution and Parallelization Example



3.3.1 Loop Distribution

If a loop nest cannot be parallelized effectively using *Optimize*, then dividing the statements in the nest using distribution may enable parallelization of some subset of the statements by either creating perfect nests or isolating dependences that prevent loop permutation. For example, in the left loop nest in Figure 7, there is a loop-carried dependence between the two assignment statements that prevents the nest from being performed correctly in parallel. Distribution exposes parallelism, and results in the two parallel nests on the right in Figure 7. *Optimize* then interchanges the resulting doubly nested loop to achieve good inner loop locality and parallelizes the outer loop. *Optimize* also tiles the singly nested loop to exploit spatial locality and parallelism. Both nests now execute efficiently and correctly in parallel.

Distribution algorithm. Beginning with the innermost loop l_n in a nest $\{l_1, \ldots, l_n\}$, the algorithm *Distribute* divides the statements into strongly connected regions *scrs* based on the dependences. Each *scr* is then placed in a loop by itself which divides the statements up into the finest granularity possible. In the style of Allen *et al.* [2], the process is repeated for the next outermost loop, until some loop cannot be distributed over the statements (this loop may of course be l_n). If new nests are created as in Figure 7, these become candidates for parallelization by *Optimize*. This algorithm is not optimal because combining distribution with loop permutation may uncover deeper distributions that in turn may be more effectively parallelized [4, 33]. This flexibility was not required in our experiments and is not explored further here.

After distribution and parallelization, there may be a sequence of parallel and sequential nests, some of which may be fused back together. We showed that the problem of fusing a set of loops is the same, regardless if they resulted from distribution or were written that way [25]. Fusion is desirable between parallel loops because it may reduce communication of shared data and it reduces the number of barrier synchronization. Barrier synchronization is often costly on multiprocessors.

3.3.2 Loop Fusion

Loop fusion merges multiple loops with *conformable headers* into a single loop. Two loop headers are conformable if they have the same number of iterations and are both either sequential or parallel loops. Two loop nests are conformable at level k, if each is perfectly nested and the headers at level 1 through k are conformable. For example, in Figure 8(a) all four nests are conformable at level 2. Fusion eliminates unnecessary barrier synchronization and reduces communication of shared data between loops. It is safe if it does not reverse any dependences between candidate loops. We only perform safe fusions. Our goal is to maximize parallelism. Subject to this constraint, we then minimize the number of parallel loops. Fusion does not combine two parallel loops when dependences would force the resulting loop to execute sequentially.

Fusion algorithm. When there is a group of adjacent loop nests with conformable headers which are differentiated only by their parallel and sequential status and n candidate nests, we have an $O(n^2)$ time and space algorithm that minimizes the number of parallel loops [25]. (More general fusion problems are NP-hard [25, 26].)

Figure 8: Fusion Example from Subroutine tridvpi in Erlebacher

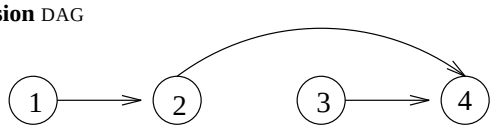
(a) original program

```
do k = 1, kmaxd
   do j = 1, jmaxd
(1)   f(1,j,k) = f(1,j,k) * b(1)
do k = 1, kmaxd
   do j = 1, jmaxd
      do i = 2, n-1
(2)      f(i,j,k) = (f(i,j,k) - a(i) * f(i-1,j,k)) * b(i)
do k = 1, kmaxd
   do j = 1, jmaxd
(3)   tot(j,k) = 0
do k = 1, kmaxd
   do j = 1, jmaxd
      do i = 2, n-1
(4)      tot(j,k) = tot(j,k) + d(i) * f(i,j,k)
```

(b) fused program

```
parallel do k = 1, kmaxd
   do j = 1, jmaxd
      f(1,j,k) = f(1,j,k) * b(1)
      do i = 2, n-1
         f(i,j,k) = (f(i,j,k) - a(i) * f(i-1,j,k)) * b(i)
      tot(j,k) = 0
      do i = 2, n-1
         tot(j,k) = tot(j,k) + d(i) * f(i,j,k)
```

(c) fusion DAG



This restricted case however arises frequently in practice. It occurs when the fusion candidates result from distribution. Programmers also write these types of adjacent and fusible nests and several occur in programs in our test suite.

The fusion algorithm works by building a fusion DAG where nodes are nests, edges represent dependences between nests, and fusion-preventing edges are specially marked. A dependence is fusion preventing if fusion is not safe or inhibits parallel execution. The algorithm first greedily merges nodes representing parallel nests such that the graph remains a DAG and no nodes connected by a fusion-preventing edge are merged. It then greedily merges sequential nodes. This merge respects the original constraints, any constraints introduced by the fusion of parallel nests, and insures that a DAG results [25]. The original order of the nests may change as long as no dependence constraints are violated.

Example. Figure 8(c) illustrates the DAG for the code in Figure 8(a). Since all the outer k loops are parallel and the number of iterations are the same up to level 2, the loops are conformable at level 2. There are no fusion-preventing dependences and thus all the nodes can be fused into one parallel doubly nested loop. The resulting code appears in Figure 8(b). Notice that statement reordering would enable an additional fusion of the inner loops and thus further improve locality. This step is beyond the scope of this paper [34, 25].

3.4 Intra and Inter-Nest Parallelization

We combine *Optimize* and *Fuser* in Figure 9 to optimize loop nests within a single procedure. We call this algorithm *Parallelize*. It combines fusion and distribution with *Optimize* to introduce effective parallelism and to improve the granularity of parallelism achieved. It takes as input a set of adjacent loop nests, $\mathcal{R} = \{l_1, \ldots, l_n\}$ in a procedure and produces an optimized version of the nests. For each nest l_j, it begins by applying *Optimize*. *Optimize* first improves locality and then detects and introduces parallelism at the outermost possible level as described above. If *Optimize* introduces parallelism, *Parallelize* goes on to the next loop nest l_{j+1}. Otherwise, *Parallelize* tries to fuse the inner loops of l_j, in order to enable permutation and tiling, and tries *Optimize* again. If it is still unable to introduce parallelism, the algorithm distributes to the finest granularity. If distribution is able to form new loop nests ln_i, they may be parallel at some inner level or the outermost level. ln_i may be inner nests or outer nests in the *else* of Figure 9. The algorithm applies *Optimize* to each ln_i. These resultant nests are candidates for fusion. Similarly, after it optimizes each outer loop l_j, the algorithm fuses the resultant nests when safe and profitable.

Figure 9: **Parallelize**: An Intraprocedural Parallelization Algorithm

INPUT: $\mathcal{R} = \{l_1, \ldots, l_n\}$, l_j adjacent nests in a procedure

OUTPUT: an optimized version of $\mathcal{R}$

ALGORITHM:

procedure Parallelize ($\mathcal{R}$**)**
  forall l_j
    if Optimize(l_j) introduces parallelism **continue**
    elseif Fuse all inner loops of l_j forming $\mathcal{F}$ and **Optimize**($\mathcal{F}$) introduces parallelism
      continue
    else
      $\{ln_1, ln_2, \ldots, ln_m\}$ = **Distribute**(l_j)
      if m = 1 **continue**
      forall ln_i **Optimize** (ln_i)
      Fuse ($ln_1, ln_2, \ldots, ln_m$)
    endif
  endfor
  Fuse($L_1, \ldots, L_n$)

3.5 Enabler: Interprocedural Analysis and Transformation

Striving for a large granularity of parallelism has a natural consequence: the compiler must look for parallelism in regions of the program that span multiple procedures. Our approach to interprocedural optimization differs fundamentally from previous research that uses inlining. Inlining is typically performed instead of interprocedural analysis and without knowing if it yields any optimization opportunities. Our approach adds to the complexity of the individual loop optimizations, but avoids performing unnecessary inlining. We restrict the application of interprocedural transformations to cases where it enables loop optimizations and is therefore expected to be profitable. This strategy is called *goal-directed* interprocedural optimization. We introduce two new interprocedural transformations: (1) *Loop embedding* – which pushes a loop header into a procedure called within the loop, and (2) *loop extraction* which extracts an outermost loop from a procedure body into the calling procedure. We also use *procedure cloning* to make specialized versions of procedures. The following subsections first review the interprocedural analysis we need, and then describe the extensions to the loop transformations, and our use of interprocedural transformations.

3.5.1 Interprocedural Analysis

This section describes the interprocedural array section analysis that enables interprocedural optimization. This analysis is part of dependence testing in ParaScope, and is computed before optimization [22]. We include this description as technical background.

We use *section* analysis to analyze interprocedural side effects to arrays [8, 20, 21, 22]. Sections represent the most commonly occurring array access patterns; single elements, rows, columns, grids, and their higher dimensional analogs. The various approaches to interprocedural array side-effect analysis must make tradeoffs between precision and efficiency [8, 11, 22, 30, 45]. Section analysis loses precision because it only represents a selection of array structures and it merges sections for all references to a variable in a procedure into a single section. However, these properties make it efficient. It often works as well as more precise techniques [22, 30].

Sections reduce the dependence problem on loops containing procedure calls to the problem on ordinary statements. For example, Figure 10 illustrates the two sections for array a that result from each of the two calls to Q. The sections are in terms of constants and parameters passed at the call. The superscript indicates a *R*ead and/or *W*rite access and their relative order. The sections contain all the information necessary to perform dependence testing in the calling procedure without further inspecting the called procedure [22]. Since we also

Figure 10: Interprocedural Parallelization Example

(a) before optimization

```
subroutine P(a)
   real a(n,n)
   integer i

   do i = 1, 7
      call Q(a,i)      S_a^RW : a[i,j=1:100]
      call Q(a,i+1)    S_a^RW : a[i+1,j=1:100]
   enddo

subroutine Q(f,i)
   real f(n,n)
   integer i,j
   do j = 1,100
      f(i,j) = f(i,j) + . . .
   enddo
```

(b) loop extraction

```
subroutine P(a)
   real a(n,n)
   integer i,j
   do i = 1, 7
      do j = 1, 100
         call Q(a,i,j)
      enddo
      do j = 1, 100
         call Q(a,i+1,j)
      enddo
   enddo

subroutine Q(f,i,j)
   real f(n,n)
   integer i,j

   f(i,j) = f(i,j) + . . .
```

(c) fusion, interchange, & parallelization

```
subroutine P(a)
   real a(n,n)
   integer i,j

   parallel do j = 1, 100
      do i = 1, 7
         call Q(a,i,j)
         call Q(a,i+1,j)
      enddo
   end parallel do

subroutine Q(f,i,j)
   real f(n,n)
   integer i,j

   f(i,j) = f(i,j) + . . .
```

test for loop interchange and fusion between loops in the caller and the call, we require sections that are slightly more precise than *data access descriptors* [8]. We need to know if the sections are precise. For example, when section analysis merges information for two read accesses it may lose precision. At a section merge, we record whether the new section is still precise or if it becomes imprecise. We also use the augmented call graph G_{ac} to reveal the call and the loop nesting structure as described in Section 2.

The next paragraph uses an example to demonstrate dependence testing and motivate testing across calls for intraprocedural loop fusion and interchange. We then discuss the transformation tests in more detail, and finally we show how to move loops across calls to effect these transformations.

Example. Consider Figure 10(a) where the calls to Q are annotated by $\mathcal{S}_a$, precise sections of array *a*. In this example, the first call reads and modifies row i, and the second call reads and modifies row $i+1$ of array *a*. Using the sections, ParaScope's dependence testing reveals the dependence between the two calls, $\vec{\delta} = \{1, 0\}$, carried by the i loop in P, and a that the j loop in subroutine Q is parallel. Notice we have the distance for the j loop, even though it results from code in subroutine Q that we have not inspected. If the loops were in the same procedure, *Parallelize* would fuse the j loops and then interchange the i and j loops. Fusing the j loops would create a perfect nest. The interchange would place the i loop in the innermost position yielding the best locality and the j at the outermost position yielding the largest granularity of parallelism, as in Figure 10(c). To perform the required tests, *Parallelize*, *Optimize*, and *Fuse* must use the G_{ac} to look across procedure calls and deal with sections as well as references. This process is very similar to dependence testing with sections.

3.5.2 Extending Loop Optimizations across Procedures Boundaries

Parallelize. Consider again the *Parallelize* algorithm in Figure 9. *Parallelize* begins with a set of adjacent nests $\mathcal{R} = \{l_1, \ldots, l_n\}$ and simply passes loop nests to *Optimize*, *Fuse*, and *Distribute*. Figure 11 contains *Enabler*, the modified, interprocedural version of *Parallelize*. To avoid the barriers of procedure calls, we generalize $\mathcal{R}$ to sets of adjacent nests and/or calls. Consider for example optimizing procedure C in Figure 12. Since the call to Q and the i loop are adjacent, we pass the call node and the loop node to *Parallelize*. Similarly, we extend *Optimize* and *Fuse* as described below. The *Driver* procedure also needs slight modifications: the partitioning step now creates sets of adjacent nests and calls, and instead of calling *Parallelize*, *Driver* calls *Enabler*.

Figure 11: **Enabler**: An Interprocedural Parallelization Algorithm

INPUT: $\mathcal{R} = \{l_1, \ldots, l_n\}$, l_j adjacent nests and calls in a procedure

OUTPUT: an optimized version of $\mathcal{R}$

ALGORITHM:

```
procedure Enabler (R)
    forall l_j
      if l_j is a call to procedure p Driver(p)
      elseif l_j nests around calls c_i to p_k and sections for p_k are not exact
        forall p_k Driver(p_k)
      elseif Optimize(l_j) introduces parallelism continue
      elseif l_j contains no calls
        Fuse all inner loops of l_j forming F and Optimize(F) introduces parallelism
          continue
      elseif l_j contains only adjacent calls c_i to p_k with outer enclosing nests
        if Fuse (c_i) forming F and Optimize(F) introduces parallelism
          continue
      elseif l_j does not contain calls
        {ln_1, ln_2, ..., ln_m} = Distribute(l_j)
        if m = 1 continue
        forall ln_i Optimize (ln_i)
        Fuse (ln_1, ln_2, ..., ln_m)
      endif
    endfor
    Fuse(l_1, ..., l_n)
```

Testing for distribution into a procedure requires more information than interprocedural sections provide, therefore we do not generalize the parameters to *Distribute*. *Enabler* ensures arguments to *Distribute* are nests.

Optimize. *Enabler* only calls *Optimize* with a loop nest that contains no calls, or that contains calls whose actions are represented by exact sections. The original version of *Optimize*, of course, works for the first case. For a nest containing one or more calls, *Optimize* uses any array references in the nest and the sections at calls to determine loop order of the loops in the calling procedure. This case only looks at the nests in the calling procedure, and the only change is to use sections in addition to references upon encountering the call statement.

If the nest contains a single procedure call, and an outer nest encloses the entire body of this procedure, *Optimize* computes memory order for each loop in the caller, and each loop in the outer enclosing nest of the called procedure as well. The sections identify these loops and all array references. As we showed in the example above, dependence testing on the exact sections results in a direction vector that includes the loops in the called procedure. *Optimize* simply uses this direction vector to determine if the loop order it wants is legal. *Optimize* does not have to compute any additional dependence information. If *Optimize* specifies an interchange of nests that cross procedure boundaries, it clones the procedure and moves the nest out of the callee and into the caller (see Section 3.5.3).

Fuser. *Enabler* (Figure 11) calls *Fuse* in 3 places. In the first call, it passes outer loop nests that do not contain calls. In the second, it passes a group of adjacent calls with exact sections and an enclosing loop nest. In this case, *Fuse* applies the fusion test to the sections for a candidate call. *Fuse* uses the exact sections to determine if the loop nest headers are conformable, to test for dependences between the sets of conformable headers, and then builds and partitions the fusion graph in the usual way. If *Fuse* finds a fusion, it extracts the loop nests from the calls and clones the called procedure (see Section 3.5.3). *Optimize* then tries again on the resulting loop structure. In the final call to *Fuse*, *Fuse* may see inexact sections, nests and/or calls, and it must check these

parameters.

Example. For Figure 10, *Driver* calls *Enabler* with the i loop from procedure P. *Enabler* determines the sections for all calls in the i loop are exact, and calls *Optimize* the first time. *Optimize* determines the i loop is not parallel, and since there is more than one call in the nest, it returns, failing to introduce parallelism. *Enabler* then determines that all calls are adjacent, and that the called procedures contain an outer loop nest, and calls *Fuse* on these calls. *Fuse* tests the sections for fusion, finds they can be fused, extracts the nests, and fuses them. *Enabler* sends the result to *Optimize* which now can interchange the i loop to exploit spatial locality, and parallelize the j loop in the outermost position.

3.5.3 Interprocedural Code Motion: Loop Embedding and Loop Extraction

Optimize and *Fuse* thus may specify that two loops in different procedures should be interchanged or fused. We use loop embedding and loop extraction to place the loops in the same procedure and enable the loop transformation. *Loop embedding* pushes a loop header into a procedure called within the loop, and *loop extraction* extracts an outermost loop from a procedure body into the calling procedure. These transformations expose the loop structure to optimization without incurring all the costs of inlining. Just as inlining is always safe, these transformations are always safe. Note, a similar analysis could decide when to perform inlining.

The choice between embedding and extraction is made based on the desired optimizing transformation. All things being equal, embedding loop nests into the called procedure is preferable because it reduces procedure call overhead by the number of iterations in the nest. If a loop nest optimization needs loops that originate inside a call site, extraction is required, as illustrated in Figures 1(d) and 10(b). An implementation could handle this in two ways. (1) When *Optimize* decides to do an interprocedural interchange, it would always perform embedding. If *Fuse* later detected a fusion involving the same nest, the loops would be extracted and fused. The disadvantage of this option is that the compiler would need to incrementally update the G_{ac}. (2) Alternatively, *Optimize* and *Fuse* could just record their desired loop transformations and a transformation phase could decide between embedding and extraction as it performed the loop transformations. This method separates mechanisms from policy and is consistent with good software engineering practices.

3.5.4 Procedure Cloning

Procedure cloning generates multiple copies of a procedure each tailored to its calling environment [13]. Even without embedding or extraction, cloning is necessary for interprocedural parallelization because multiple versions of a procedure are required if a procedure is called in two or more settings that require different parallelizing optimizations. For instance, there are two calls to Q in Figure 12(a); one is surrounded by a loop and one is not. Both the i and j loops are parallel, but we only want to introduce one level of parallelism. We therefore produce a version tailored to each call site, as illustrated in Figure 12(b).

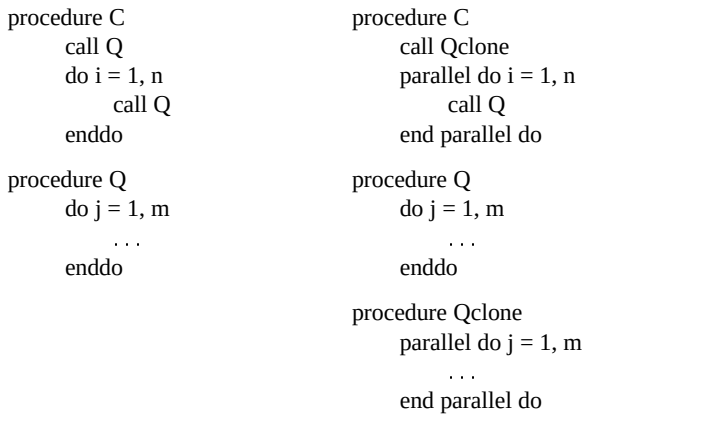
3.5.5 Summary

The judicious application of interprocedural optimizations does not change the basic structure of the kernel parallelization algorithm, but it complicates testing the safety and profitability of the individual loop transformations, as described above. Our strategy separates legality and profitability tests from the mechanics of the transformations [33]. The safety tests depend on the precision of the dependence information and section analysis. For permutation, the dependences must be precise enough in the caller to determine if they would be reversed after permutation. Since fusion requires additional dependence testing, the sections must be precise. If they are not precise, the algorithms conservatively assume that transformation is unsafe.

4 Experiment

For our experimental validation, we measure our algorithm's ability to match or exceed performance on parallel programs written by programmers who thought and cared about parallel performance, not dusty deck sequential programs. Our baseline measurement is thus a hand-coded parallelized program. We assembled programs

Figure 12: Cloning Example



(a) original (b) parallelized with cloning

written for a variety of parallel machines. We eliminated all the parallel loops and synchronization to create sequential versions of each program. We then applied our algorithm to these sequential versions. The compiler was required to use its analysis and algorithms to parallelize the program. We do not recommend that users eliminate their directives, but we use use this version to measure the compiler's ability to find and further optimize parallelism we know exists. Since the focus of this paper is the optimization algorithm, in a few cases we assume more advanced analysis than was implemented. We note all these exceptions. We executed and compared the original hand-parallelized version, the sequential version, and the hand-optimized parallel version on a 20 processor Sequent Symmetry S81. Our results are applicable to other symmetric multiprocessors. Our results are very encouraging. Our algorithm exceeds or matches hand-coded parallel programs for seven of nine programs in our suite. Based on our successes and failures, we comment on a parallel programming style from which compilers are more likely to achieve or improve hand-tuned performance for shared-memory, bus-based parallel machines.

5 Methodology

In this section, we describe the experimental setup of the program versions, the implementation status, the execution environment, and the program test suite.

5.1 Creating Program Versions

For each Fortran program we obtained, we measured three program versions, *the original parallel version*, *the sequential version*, and *the hand-optimized parallel version*. Each Fortran program was then compiled and executed on the Sequent. The original parallel version is parallelized according to the user's original intent. For each of the programs that were originally written for the Sequent, the original program version is simply the user's parallel program. For the programs written for other architectures, we modified all the parallelization directives to reflect the equivalent Sequent directives. The programs used parallel loop directives which include declarations for private variables, and critical sections. In *Erlebacher*, the parallelism is not explicit. Here, we performed a naive parallelization of outer loops to create the hand parallelized version.

We created the sequential version of each program simply by ignoring all the parallel directives. We optimize the sequential version using the advanced analysis and transformations available in our interactive parallel programming tool, the ParaScope Editor (PED) [12, 27], and also use PED to hand-apply our parallelization

algorithm. PED is a source-to-source transformation tool which provides dependence and section analysis, loop parallelization, and loop transformations. Although the individual loop transformations were automated, the parallelization algorithm and interprocedural transformations were not.[5] We discuss the implementation of the analyses and transformations below.

5.2 Parallelization

Analysis. PED uses a range of dependence tests that start with simple, quick tests and then, if necessary, uses more powerful and expensive tests [18]. If a dependence cannot be disproved, PED produces distance and direction vectors. It also performs analysis to determine scalar variables that can be made private in parallel loops. To improve the precision of dependence testing, it uses advanced symbolic dependence tests, interprocedural constants, interprocedural symbolics, and interprocedural MOD and REF array sections [22]. All of this testing is implemented.

Transformations. PED's selection of source-to-source transformations includes loop parallelization with private variable declarations, loop interchange, fusion, distribution, and tiling. It does not include loop embedding, extraction, or procedure cloning. PED produces a variety of parallel Fortran outputs, one of which is Sequent Parallel Fortran. We used the transformations available in PED to apply our parallelization algorithm. In PED transformations have two phases. The mechanics of a transformation are separated from its test for correctness. Users select a transformation and in response, PED determines the safety of the transformation using dependence analysis. If it is safe, the user decides to apply it or not. If a transformation is applied, PED carries out the mechanics of changing the program and incrementally updating the dependence information to reflect the new source. We did not implement our algorithm in PED. We instead performed the transformations as specified by the algorithm in PED, and applied them only when PED assured their correctness. We kept optimization diaries for each program [33].

5.3 Execution Environment

We ran and compared all three versions on a Sequent Symmetry S81 with 20 processors. We validated each program using its output. For each of the programs, all the versions produced the same correct output. The Sequent has a simple parallel architecture, allowing our experiments to focus solely upon medium and large grain parallelism. Each processor has its own 64Kbyte two-way set-associative cache with a cache line size of 4 words. Each processor and one shared, main memory is connected to the bus. The Sequent compiler introduces parallelism based on parallel loop and fork directives [39].

We used the parallel loop directives with private variable declarations to introduce parallelism. We compiled with version 2.1 of Sequent's Fortran ATS compiler using the compiler options that specify multiprocessing, the Weitek 1167 floating-point accelerator, and optimization at its highest level (O3). In a few programs, Sequent compiler bugs prevented the highest level of optimization and use of the Weitek chip at the same time. In these programs, we used the Weitek 1167 floating-point accelerator since it achieves better performance.

5.4 The Programs

To our knowledge, no test suites of explicitly parallel Fortran programs currently exist. To obtain parallel programs, we solicited scientists at Argonne National Laboratory and users of the Sequent and Intel iPSC/860 at Rice. We present all the programs that users submitted. The first 9 applications programs in Table 1 were volunteered and were written to run on the following parallel machines: the Sequent Symmetry S81 with 20 processors, the Alliant FX/8 with 8 and 16 processors, and the Intel iPSC/860 with 32 processors. Table 1 enumerates the programs, their total number of non-comment lines, their authors and affiliations. 9 programs out of all 10 are dense matrix codes. *Interior* is a sparse matrix code. The authors are all numerical scientists and 6 of the 9 programs are state-of-the-art parallel versions. Papers have been published about them and a lot of

[5] Our algorithm is automatable, and much of the algorithm has been implemented since these experiments were performed.

Table 1: Program Test Suite

Name	*Description*	*lines*	*Authors*	*Affiliation*
1. Seismic	1-D Seismic Inversion	606	Michael Lewis	Rice
2. BTN	BTN Unconstrained Optimization	1506	Stephen Nash, Ariela Sofer [36, 37]	George Mason
3. Erlebacher	ADI Integration	615	Thomas Eidson	ICASE
4. Interior	Interior Point Method	3555	Guangye Li, Irv Lustig [31]	Cray Research, Princeton
5. Control	Optimal Control	1878	Stephen Wright	Argonne
6. Direct	Direct Search Methods	344	Virginia Torczon [14]	Rice
7. ODE	Two-Point Boundary Problems	3614	Stephen Wright [51]	Argonne
8. Multi	Multidirectional Search Methods	1025	Virginia Torczon [14]	Rice
9. Banded	Banded Linear Systems	1281	Stephen Wright [50]	Argonne
10. Linpackd	Linpackd benchmark	772	Jack Dongarra [15]	Tennessee

attention was paid to their performance. It is therefore unlikely that large amounts of additional parallelism are available without significant algorithm restructuring. The programs are described in more detail elsewhere [33].

The discussion will focus on the first 8 programs. We included *Linpackd* since it is well known and it contains parallelism, but we did not use a hand-parallelized version. The ninth code, *Banded*, did not execute correctly on the Sequent. *Banded* was written for an Alliant FX/8 and converting three parallel loops to the equivalent Sequent parallel loop directives resulted in a runtime error. Because of this error, we do not present results for this program, but we did examine and try to parallelize it. Our techniques could not discover any of the parallelism in *Banded*. The parallel loops contained procedure calls that explicitly divided a linearized array on to 8 processors. The program used offsets into a logical row of a linearized array at a call site and then subscripted it with negative indices. This practice is not legal Fortran, will thwart even advanced dependence analysis, and is most likely responsible for the runtime error on the Sequent. Our inability to analyze or parallelize this program was due to two poor programming styles: linearization of logical arrays, and using a fixed number of processors to divide the work. These practices illustrate a programming style that is not portable to a different machine or even to different numbers of processors. Since *Banded* did not execute, we exclude it from the rest of the discussion.

By collecting programs rather than writing them ourselves we avoided the pitfall of writing a test suite to match the abilities of our techniques and architecture. However, many of the problems inherent to any program test suite also arise here. Maybe only authors of easy to parallelize, well structured codes volunteered. Maybe the authors of poorly structured ones did not want to expose their codes to a critical eye.

6 Results

We measured execution times for:

seq: the sequential version of the program,

hand: the hand-coded, user parallelized program, and

opt: the version obtained using our optimization algorithm.

The elapsed times for the entire applications were measured in seconds using the system call *secnds*. From these times, we computed speed-ups for the parallel programs. We also measured subparts of a program if there were differences between the hand-optimized parallel version and the user parallelized version. We separate those differences into the following categories.

The Entire Application: execution time of the application.

Optimization: execution times in subparts of the program where our optimization algorithm generated a different parallelization strategy than the hand-coded version.

Analysis: execution times in subparts where the optimized version could not detect parallelism specified by the hand-coded version.

Table 2: Speed-ups over Sequential Program Versions

Speed-ups over the sequential version on a 19 Processor Sequent							
	Optimization		*Analysis*		*Entire Application*		
Name	*hand*	*opt*	*hand*	*opt*	*hand*	*opt*	Δ
Seismic	3.0	7.9			9.1	12.3	35%
BTN	2.0	3.9	-6.1	1.0	3.2	4.1	28%
Erlebacher	13.8	15.0			13.2	14.2	7%
Interior	6.9	10.4	6.9	5.2	6.9	6.9	0%
Control†					3.8	3.8	0%
Direct					2.4	2.4	0%
ODE					3.4	3.4	0%
Multi			15.1	1.0	5.3	1.0	-530%
Linpackd		16.5				9.2	*NA*

Table 3: Program Execution Times

Execution Times in seconds									
	Optimization			*Analysis*			*Entire Application*		
	seq	*hand*	*opt*	*seq*	*hand*	*opt*	*seq*	*hand*	*opt*
Seismic	21.14	7.14	2.69				155.97	17.05	12.59
BTN	13.97	7.045	3.57	0.14	0.85	0.14	44.01	13.93	10.73
Erlebacher	87.83	6.36	5.86				88.22	6.67	6.20
Interior	19.50	2.00	1.87	24.12	3.47	4.64	1044.16	151.16	151.53
Control†							17.44	4.61	4.61
Direct							151.28	63.65	63.65
ODE							41.96	12.22	12.22
Multi				75.45	4.98	75.45	87.60	16.32	87.60
Linpackd	517.87		31.43				547.59		59.43

† : 8 processors

We used the microsecond clock, *getusclk*, to measure execution times for the differing program subparts. For differences on inner loops, we measured the performance of the outermost enclosing loop in order to disrupt execution as little as possible. The speed-ups of these optimized subparts are under reported.

Table 2 contains speed-ups over the sequential program version for the entire application and subparts. The execution times in seconds of all the program and program subpart versions appear in Table 3. In both tables, a blank entry means that no program or program subpart fell in that category. Since we did not use a hand parallelized version of *Linpackd*, those columns are empty in Tables 2 and 3. In *Control*, *Direct*, and *ODE*, the optimized version and the user parallelized version did not differ and therefore we did not measure any subparts.

6.1 Interpretation and Analysis of Results

As can be seen in the percent change column (Δ) in Table 2, the optimized programs either performed as well or better than the hand-coded parallel versions except for *Multi*. These programs are complete applications that contain I/O and computation. The speed-ups were therefore not linear and ranged from 2.4 to 14.2 on 19 processors. Consider the *Optimization* category. Every time our algorithms chose an optimization strategy different from the user's, it was an improvement. The improvement was at least a factor of 1.9 and at best a factor of 4.9.

In the following sections, we discuss the program analysis and optimizations that achieved our improvements in more detail. We include analysis because good optimization is intimately tied to good analysis. Since our focus is optimization, we assumed better analysis than was implemented in a few cases. These caveats are detailed as well.

6.1.1 Program Analysis

The 8 programs contained 923 loops. There were 445 nests of depth 1 or deeper. Dependence analysis detected 551 parallel loops at all levels of nesting out of the 923 loops (60%) and 271 out of 445 (61%) parallel loops in the outermost position of a nest.[6] Compared with the programmers, dependence analysis failed to detect user parallelism in about 3% of the loops and found parallelism users had missed in about 2% of the loops. When users introduced parallelism, the compiler was usually able to find it. Compilers are generally more thorough and meticulous than the average user, but users often have high-level knowledge the compiler cannot ascertain. The improvements experienced by the optimized versions were not due to analysis, but were due to our optimization strategy which differed from the user's strategy (see Section 6.1.2). All the degradations in Tables 2 and 3 result from analysis failing to find parallelism.

In three programs, *BTN*, *Interior*, and *Multi*, users found more parallelism than our analysis did (the *Analysis* column in Tables 2 and 3). For *Interior*, these degradations did not have much effect on overall execution time. If we look at the execution times in Table 3, it is apparent that this program subpart only affects the overall execution time by less than 3%. Each of *BTN* and *Multi* contain parallel loops with critical sections that update shared variables. Analysis techniques exist that can properly identify the parallelism [44], but since it was not part of our algorithm, we did not use them. In *BTN*, the benefit of parallelism was actually overwhelmed by the overhead of the critical section, resulting in better performance when the loop executed sequentially. In *Multi*, there was a single outer loop with a critical section. This parallel loop accounted for 86% of the sequential running time and 30% of the parallel running time. The algorithm did not parallelize this loop because of the critical section, and thus *Multi*'s performance degraded.

Interprocedural Sections. Interprocedural section analysis proved to be a very important. Only one program, *Erlebacher*, did not have one or more parallel loops containing a call. Out of a total of 246 procedure calls made by the first 8 programs in the tables, 119 (48%) of these calls are nested inside loops and 48 (20%) of these loops were parallel. Section analysis detected parallel loops with calls as well as programmers. Parallelizing *BTN* and *ODE* required flow-sensitive section analysis. In *BTN* and *ODE*, we determined the array kill by hand since it is a very simple case that a reasonable implementation would catch. To effectively analyze and optimize the modular parallel programming style found in these programs requires both flow-sensitive and flow-insensitive interprocedural section analysis.

Index Arrays. Five of the 10 programs use index arrays that are permutations of the index set. Several of these are monotonic non-decreasing with a regular, well defined pattern. In *Interior*, *Control*, and *Direct*, parallelization would not have been possible without using user assertions and the testing techniques developed in our earlier research [32]. The other two programs used them in a way that did not affect parallelization. When the user asserts that an index variable used in a subscript is a monotonic non-decreasing permutation array, dependence testing can then eliminate dependences and detect parallel loops. We used this information to parallelize loops in *Interior*, *Control*, and *Direct*.

Linearized Arrays. *ODE* and *Banded* contain linearized arrays and use symbolics to index them in order to simulate multiply dimensioned arrays. A symbolic test is needed when the symbolic term is unknown, but loop invariant. This feature would enable precise dependence analysis of many symbolic references into linearized arrays. However, a better solution is to reward well structured multidimensional array references with excellent

[6] The statistics in this section do not include *Banded* since it did not execute on the Sequent, nor *Linpackd* since we did not use a hand parallelized version.

Figure 13: Extracting, Fusing, and Cloning in Seismic

```
subroutine setvel
      call setvz(..)
      call ftau(..)
      parallel do i = 1, np
            call chgvar(..)
      end parallel do
      call fzeta(..)
```

(a) original

```
subroutine setvel
      parallel do i = 1, np
            call setvzExt(..)
            call ftauExt(..)
            call chgvar(..)
            call fzetaExt(..)
      end parallel do
```

(b) parallelization, extraction, & fusion

performance. Programmers will then have an incentive to use multiply dimensioned arrays when appropriate. If array linearization improves performance, as it often does on the Cray YMP, then the compiler should perform it.

Summary. For the most part, we used the existing interprocedural and intraprocedural dependence and symbolic analysis. Since our focus is on the optimization strategy, we assumed better analysis than was implemented in a few straight-forward cases: flow-sensitive analysis for *BTN* (6 loop nests) and *ODE* (2 loop nests), and index variable analysis for *Interior*, *Control*, and *Direct.*

6.1.2 Program Optimization

Three programs, *Seismic*, *BTN*, and *Erlebacher*, experience significant improvements due to our optimization strategy. In *Seismic*, the majority of the improvement comes from fusing 4 loops. In the original program, part of which appears in Figure 13(a), each of the subroutines *setvz*, *ftau*, and *fzeta* contains an outer, enclosing parallel loop with *np* iterations. Our parallelization algorithm, using the augmented call graph, detects that these parallel loops are candidates for fusion. The fusion is safe, so it extracts them and performs the fusion in the subroutine *setvel*, as illustrated in Figure 13(b). The optimized version actually has more procedure call overhead, but the benefits of reduced synchronization and communication far out weigh this cost. None of the other optimized programs use interprocedural transformations.

BTN's improvements are due to improved parallelization of 3 important nests that accounted for 50% of the hand-coded parallel execution time. The *Optimize* portion of our algorithm improves the locality of the nests with permutation and then tiles to introduce outer loop parallelism. In this case, tiling uses permutation to move the parallel loop out and leaves a strip in place to exploit locality. This optimization cuts the execution time of the 3 nests in half and improves overall performance by 28%. These nests need to balance locality and parallelism. The users successfully parallelized 24 outer loops in which locality and parallelism did not conflict, but failed to achieve parallelism and data locality on the 3 nests when they conflicted. This result implies users are capable of detecting parallelism and locality, but are less proficient at combining them.

Similarly, most of the improvement to *Erlebacher* results from the use of permutation and tiling by *Optimize* to balance locality and parallelism. *Erlebacher* also benefits from the application of fusion to 8 groups of nests. The number of nests fused in a group varied from 2 to 5 nests, with an average of 3 nests fused. *Interior* also benefits from fusion.[7]

Except for distribution and embedding, the programs exercised all of the transformations in the parallelization algorithm. Every time our algorithms chose an optimization strategy that differed from the users, it was an improvement.

[7] Fusion of sequential loops in *Control* also improves its performance, but scalar improvements are beyond the scope of this work.

7 Related Work

Not many studies of parallelizing optimizers have been published. Many commercial parallelizing compilers do not reveal their optimization strategies to maintain a market advantage. The IBM PTRAN project, an industrial research compiler, has published parallelization algorithms that use control and data dependence, and a wide selection of transformations, but without results [1, 40, 41]. Below, we compare this study with those of parallelizing compilers from Illinois and Stanford [10, 17, 19, 43].

The Illinois studies are traditional; they evaluate their techniques on dusty deck programs [10, 16, 28, 17]. They extend Kap, an automatic parallelizer, and then use it to parallelize the Perfect Benchmarks. Their target architecture is Cedar, a shared-memory parallel machine with cluster memory and vector processors. Their work focuses on detecting parallelism via array and scalar analysis, rather than improving locality. Their interprocedural analysis results de facto from inlining or is performed by hand. In some cases, they do not measure program performance, but rather number of parallel loops found. Our results demonstrate that performance is dependent on locality and granularity, not just parallelism. The algorithms Kap uses are unpublished, which limits what can be learned from these papers. The resulting programs were then further improved manually by 'automatable' transformations. It is not clear that even if each individual transformation they propose is automatable, that a practical decision procedure exists that could correctly apply them. The most recent work on Polaris [17] demonstrates that they have come closer to finding such a decision procedure, but they still do not specify it. In contrast, our study uses a more clearly defined algorithm. Both studies however would benefit greatly from complete implementations.

Singh & Hennessy used the Alliant FX/8, the Encore Multimax, and their Fortran compilers [42, 43]. The compiler algorithms are again unpublished. The FX/8 has cluster memory instead of local caches, which means all data accesses are slow. Since caches are not available to improve performance, the parallelization algorithm is simplified. On the Encore, the slow processors minimized the impact of its small local caches. Singh & Hennessy considered dusty deck programs. By inspection, they found interprocedural analysis, user assertions, and symbolic analysis to be useful. Our results offer a significant step towards providing these analyses, as well as going a step further to optimize for a more complex architecture. The main result in these papers is that successful parallelization requires many programs to be rewritten. We start with this premise. However, the ability of our techniques to find further improvements reveals that even after users perform algorithm restructuring for parallelism, there is performance to be gained.

Our core technique, *Optimize*, bears the most similarity to Wolf & Lam's research [47, 48]. Their algorithm is potentially more precise and uses skewing and reversal. Our algorithm can take advantage of known loop bounds to more precisely compute locality and granularity of parallelism, and is more efficient. When a nest of depth n is fully permutable our algorithm experiences it's best case $O(n\ log(n))$ time complexity while Wolf & Lam's algorithm experiences exponential behavior in the depth of the loop nest n. The most expensive step in both algorithms is determining the reuse. Our algorithm performs this step only once, and then chooses an optimization to achieve the reuse. Their algorithm evaluates reuse for every legal permutation. Their work includes very few experimental results for the parallelization algorithm, and they do not perform fusion, distribution, or any interprocedural analysis and transformations. More recent work on the SUIF system includes extensive interprocedural analysis and data and control restructuring between nests to further improve locality and parallelism [5, 19]. This work demonstrates important improvements over our approach, and some success on dusty deck programs. However, they do not perform fusion or distribution, and our approach is effective in many cases.

Instead of using fusion to eliminate barrier synchronization, recent work has focused on replacing barrier synchronization between nests with explicit data placement and finer grain communication [6, 7, 38, 46]. The data placement yields locality on the processor and the finer grain communication enables the processors to overlap more computation and communication rather than all waiting at a barrier. When fusion is legal, fusion can be more effective because references to the same location occur more closely together in time, making the

cache more likely to exploit the locality. For example, consider two adjacent loops that access the same array. If the working set of the first loop exceeds the cache, fusion yields reuse. However when fusion is not legal, these techniques can be used instead to improve performance.

Jeremiassen and Eggers [23] improve locality in explicitly parallel programs by restructuring the data layout. We instead focus on restructuring the program control flow. We also transform the program into a sequential equivalent and therefore do not analyze parallel programs. Data transformations [23] and combining data transformations with control transformations [5] can successfully parallelize programs that the techniques presented in this paper can not.

8 Conclusions

This paper presents a new parallelization algorithm that balances parallelism and data locality. We use an effective strategy to introduce locality, exploit parallelism, and maximize the granularity of parallelism. Interprocedural section analysis is an important component of our successes. We evaluated the parallelization algorithm against hand-parallelized programs with promising results. The algorithm improves performance over hand-parallelized programs whenever it applied optimizations, significantly improving performance in 3 of the 9 programs. It matches or improves parallel performance for programs written in Fortran 77 with a clean, modular parallel programming style. The successes and failures indicate that many parallel programmers are using a portable programming style and an advanced compiler can analyze and optimize these programs. The compiler improvements come from balancing locality and parallelism, and increasing the granularity of parallelism. The compiler also improves on user-parallelized codes because it is inherently more methodical than a user. Most importantly, these results suggest that we need both parallel algorithms and compiler optimizations to effectively utilize parallel machines.

9 Acknowledgments

This research was supported by NSF grant CCR-9525767 and a DARPA/NASA Research Assistantship in Parallel Processing. Use of the Sequent Symmetry S81 was provided under NSF Cooperative Agreement No. CDA-8619393. Any opinions, findings, and conclusions expressed in this paper are the author's and do not necessarily reflect those of the sponsor. The author did the majority of this work while she was a graduate student at Rice University.

I especially want to thank Ken Kennedy, who provided impetus and guidance for much of this research. I am grateful to Mary Hall, Chau-Wen Tseng, Preston Briggs, Paul Havlak, Nat McIntosh, and the anonymous reviewers for their comments which improved this work and its presentation. Paul Havlak's implementation of regular sections proved invaluable. To all of these people go my thanks.

References

[1] F. Allen, M. Burke, P. Charles, J. Ferrante, W. Hsieh, and V. Sarkar. A framework for detecting useful parallelism. In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[2] J. R. Allen, D. Callahan, and K. Kennedy. Automatic decomposition of scientific programs for parallel execution. In *Proceedings of the Fourteenth Annual ACM Symposium on the Principles of Programming Languages*, Munich, Germany, January 1987.

[3] J. R. Allen and K. Kennedy. Automatic loop interchange. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, Montreal, Canada, June 1984.

[4] J. R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[5] J. Anderson, S. P. Amarasinghe, and M. Lam. Data and computation transformations for multiprocessors. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Santa Barbara, CA, July 1995.

[6] B. Appelbe, S. Doddapaneni, and C. Hardnett. A new algorithm for global optimization for parallelism and locality. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[7] B. Appelbe, C. Hardnett, and S. Doddapaneni. Program transformation for locality using affinity regions. In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[8] V. Balasundaram and K. Kennedy. A technique for summarizing data access and its use in parallelism enhancing transformations. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 41–53, Portland, OR, June 1989.

[9] U. Banerjee. A theory of loop permutations. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*. The MIT Press, 1990.

[10] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Effective automatic parallelization with Polaris. *International Journal of Parallel Programming*, May 1995.

[11] M. Burke and R. Cytron. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 162–175, Palo Alto, CA, June 1986.

[12] K. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. The ParaScope parallel programming environment. *Proceedings of the IEEE*, 81(2):244–263, February 1993.

[13] K. Cooper, M. W. Hall, and K. Kennedy. Procedure cloning. In *Proceedings of the 1992 IEEE International Conference on Computer Language*, Oakland, CA, April 1992.

[14] J. E. Dennis, Jr. and V. Torczon. Direct search methods on parallel machines. *SIAM Journal of Optimization*, 1(4):448–474, November 1991.

[15] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK User's Guide*. SIAM Publications, Philadelphia, PA, 1979.

[16] R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, and D. Padua. Restructuring Fortran programs for Cedar. *Concurrency: Practice and Experience*, 5(7):553–574, October 1993.

[17] R. Eigenmann, J. Hoeflinger, and D. Padua. On the automatic parallelization of the Perfect Benchmarks. *IEEE Transactions on Parallel and Distributed Systems*, 9(1):5–23, January 1998.

[18] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 15–29, Toronto, Canada, June 1991.

[19] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S. Liao, and M. Lam. Detecting coarse-grain parallelism using an interprocedural parallelizing compiler. In *Proceedings of Supercomputing '95*, San Diego, CA, December 1995.

[20] M. W. Hall, K. Kennedy, and K. S. McKinley. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91*, pages 424–434, Albuquerque, NM, November 1991.

[21] M. W. Hall, B. R. Murphy, S. P. Amarasinghe, S. Liao, and M. Lam. Data and computation transformations for multiprocessors. In *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing*, Columbus, OH, August 1995.

[22] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[23] T. E. Jeremiassen and S. J. Eggers. Reducing false sharing on shared memory multiprocessors through compile time data transformations. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–188, Santa Barbara, CA, July 1995.

[24] K. Kennedy and K. S. McKinley. Optimizing for parallelism and data locality. In *Proceedings of the 1992 ACM International Conference on Supercomputing*, pages 323–334, Washington, DC, July 1992.

[25] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pages 301–321, Portland, OR, August 1993.

[26] K. Kennedy and K. S. McKinley. Typed fusion with applications to parallel and sequential code generation. Technical Report TR93-208, Dept. of Computer Science, Rice University, August 1993.

[27] K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in an interactive parallel programming tool. *Concurrency: Practice and Experience*, 5(7):575–602, October 1993.

[28] D. Kuck, E. Davidson, D. Lawrie, A. Sameh, C.-Q. Zhu, A. Veidenbaum, J. Konicek, P. Yew, K. Gallivan, W. Jalby, H. Wijshoff, R. Bramley, U.M. Yang, P. Emrath, D. Padua, R. Eigenmann, J. Hoeflinger, G. Jaxon, Z. Li, T. Murphy, J. Andrews, and S. Turner. The Cedar system and an initial performance study. In *Proceedings of the 20th International Symposium on Computer Architecture*, San Diego, CA, May 1993.

[29] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. J. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, January 1981.

[30] Z. Li and P. Yew. Efficient interprocedural analysis for program restructuring for parallel programs. In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems (PPEALS)*, New Haven, CT, July 1988.

[31] I. J. Lustig and G. Li. An implementation of a parallel primal-dual interior point method for multicommodity flow problems. Technical Report CRPC-TR92194, Center for Research on Parallel Computation, Rice University, January 1992.

[32] K. S. McKinley. Dependence analysis of arrays subscripted by index arrays. Technical Report TR91-162, Dept. of Computer Science, Rice University, December 1990.

[33] K. S. McKinley. *Automatic and Interactive Parallelization*. PhD thesis, Dept. of Computer Science, Rice University, April 1992.

[34] K. S. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[35] K. S. McKinley and O. Temam. A quantitative analysis of loop nest locality. In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 94–104, Boston, MA, October 1996.

[36] S. G. Nash and A. Sofer. A general-purpose parallel algorithm for unconstrained optimization. *SIAM Journal of Optimization*, 1(4):530–547, November 1991.

[37] S. G. Nash and A. Sofer. BTN: software for parallel unconstrained optimization. *ACM TOMS*, 1992.

[38] M. O'Boyle and F. Bodin. Compiler reduction of synchronization in shared memory virtual memory systems. In *Proceedings of the 1995 ACM International Conference on Supercomputing*, pages 318–327, Barcelona, Spain, July 1995.

[39] A. Osterhaug, editor. *Guide to Parallel Programming on Sequent Computer Systems*. Sequent Technical Publications, San Diego, CA, 1989.

[40] V. Sarkar. Automatic partitioning of a program dependence graph into parallel tasks. *IBM Journal of Research and Development*, 35(6):779–804, November 1991.

[41] V. Sarkar and R. Thekkath. A general framework for iteration-reordering loop transformations (technical summary). In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 175–187, San Francisco, CA, June 1992.

[42] J. Singh and J. Hennessy. An empirical investigation of the effectiveness of and limitations of automatic parallelization. In *Proceedings of the International Symposium on Shared Memory Multiprocessors*, Tokyo, Japan, April 1991.

[43] J. Singh and J. Hennessy. Finding and exploiting parallelism in an ocean simulation program: Experiences, results, and implications. *Journal of Parallel and Distributed Computing*, 15(1):27–48, May 1992.

[44] J. Subhlok. *Analysis of Synchronization in a Parallel Programming Environment*. PhD thesis, Dept. of Computer Science, Rice University, August 1990.

[45] R. Triolet, F. Irigoin, and P. Feautrier. Direct parallelization of CALL statements. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 176–185, Palo Alto, CA, June 1986.

[46] C. Tseng. Compiler optimizations for eliminating barrier synchronization. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 144–155, Santa Barbara, CA, July 1995.

[47] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Dept. of Computer Science, Stanford University, August 1992.

[48] M. E. Wolf and M. Lam. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[49] M. J. Wolfe. Advanced loop interchanging. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 536–543, St. Charles, IL, August 1986.

[50] S. J. Wright. Parallel algorithms for banded linear systems. *SIAM Journal of Scientific and Statistical Computation*, 12(4):824–842, July 1991.

[51] S. J. Wright. Stable parallel algorithms for two-point boundary value problems. *SIAM Journal of Scientific and Statistical Computation*, 1992.