# Evaluating Automatic Parallelization for Efficient Execution on Shared-Memory Multiprocessors

Kathryn S. McKinley
*Department of Computer Science, University of Massachusetts, Amherst, MA 01003-4610*
*mckinley@cs.umass.edu*

## Abstract

We present a parallel code generation algorithm for complete applications and a new experimental methodology that tests the efficacy of our approach. The algorithm optimizes for data locality and parallelism, reducing or eliminating false sharing. It also uses interprocedural analysis and transformations to improve the granularity of parallelism. Although the individual components of the algorithm have been published previously, their coordination is unique to this paper. For experimental validation, we do not attempt to parallelize 'dusty deck' programs where many have tried and failed. Instead, we collect programs where the users tried to achieve excellent parallel performance. We apply our optimizations to *sequential* versions of these programs, *i.e.,* the compiler was required to use its analysis and algorithms to parallelize the program and could not rely on user *assertions* that for example, a loop is parallel. With this metric, our algorithm improves or matches hand-coded parallel programs on shared-memory, bus-based parallel machines for eight of the nine programs in our test suite.

## 1 Introduction

A lesson to be learned from vectorization is that users rewrote programs based on feedback from vectorizing compilers. The rewritten programs were independent of any particular vector hardware, but 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 vectorization was not successful for dusty deck programs, that when programmers have never considered medium to large grain parallelism, automatic parallelization is doomed to failure. Indeed, finding medium to large grain parallelism is more difficult than single statement parallelism and compilers have had few successes on dusty deck programs [8, 15, 20, 21].

Since it is unknown how much parallelism a dusty deck program contains, measuring the success of a compiler on one is at best tenuous. The programs may actually be completely sequential, parallel, or somewhere in between. However, only linear speed-up can be declared a success and linear speed-up is rare, even for parallel applications. In practice, parallel programs often require algorithms and data structures that differ from equivalent sequential and vector programs. The intellectual and programming costs required for good parallel performance need to be paid. Our hope is that by providing sophisticated compilers which map parallel programs to modestly and massively parallel machines that the programming cost will only have to be paid once. Users will concentrate on parallel algorithms at a high level and the compiler will be responsible for machine dependent details such as exploiting the memory hierarchy. In this paper, we test this thesis for Fortran on shared-memory, bus-based parallel machines with local caches.

We first develop an advanced parallelizing algorithm for complete applications that exploits and balances data locality, parallelism, and the granularity of parallelism. These optimization techniques minimize false sharing between processors. Since we have found that large and medium grain parallelism often requires crossing procedure boundaries, it also uses interprocedural analysis and optimization. The algorithms are described in Section 3. Using this optimizer, we designed an experimental study which took as input parallel programs. These programs were written for a variety of parallel machines. We applied the optimizer to sequential versions of the programs. Our algorithm was therefore required to use its analysis and optimizations to exploit and introduce parallelism. It could not rely on user assertions, *e.g.,* a loop is parallel or a variable privatizable.

Most of the programs in our test suite are published versions of state of the art parallel algorithms[18]. It is therefore unlikely that large amounts of additional parallelism are available without more algorithm restructuring. Nor are these programs obviously parallel. Many require interprocedural and symbolic analysis to find parallel loops. Using the original parallelization as a standard gives us a measure of success for the abilities our optimizer, unlike dusty deck studies where the goal in terms of the amount of parallelism is unknown. By examining programming styles in light of the compiler's successes and failures, we also explore whether a machine-independent parallel programming style exists. We found that for the most part, these parallel programmers use a clean, modular style that is amenable to compiler analysis and optimization, allowing the more machine-dependent optimizations to be left to the compiler. In cases where data locality and parallelism intertwined, our compiler improved hand-coded performance by eliminating false sharing.

We present a brief technical background and then describe our compilation strategy. We spend the remainder of the paper detailing and interpreting our experiments.

## 2 Technical Background

**Data Dependence.** We assume the reader is familiar with *data dependence*. Throughout the paper, $\vec{\delta} = \{\delta_1 \ldots \delta_k\}$ represents a hybrid direction/distance vector for a data dependence between two array references, corresponding from left to right to the outermost loop to innermost loop enclosing the references. Data dependences are *loop-independent* if the references to the same memory loca-

The majority of this work was done while the author was at Rice University and supported by a DARPA/NASA Research Assistantship in Parallel Processing. Use of the Sequent Symmetry S81 was provided under NSF Cooperative Agreement No. CDA-8619393.

***The 8th ACM International Conference on Supercomputing*, Manchester, England, July 1994.**

tion 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 generally, we assume Fortran's column-major storage.

## 3 Automatic Parallel Code Generation

This section contains a new algorithm for parallel code generation of complete applications. 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. For convenience, we name the components as follows.

**Optimize** - uses loop permutation and tiling to exploit data locality and parallelism, minimizing or eliminating false sharing [11, 5].

**Fuser** - performs loop fusion and distribution to enable *Optimize* and increase the granularity of parallelism [12, 13]. The combination is an effective *kernel* (intraprocedural) parallelization algorithm.

**Enabler** - uses interprocedural analysis and transformations to enable the kernel algorithm to be applied across procedure calls. In particular, loops containing calls can be parallelized and nests spanning calls optimized. The interprocedural transformations, loop embedding, loop extraction, and procedure cloning are used only when they enable loop transformations [9, 18].

These components appeared previously in the literature and for the algorithmic details the reader should refer to the appropriate articles [9, 11, 12, 18]. Section 3.4 however extends and integrates them for the first time into a single code generation algorithm. To illuminate the algorithm and experimental results, we summarize its components below.

### 3.1 Optimize: Data Locality and Parallelism

The most effective and essential component of our parallel code generation algorithm uses a simple memory model to drive optimizations for data locality and parallelism [5, 11]. We employ loop permutation and tiling to introduce and exploit data locality and 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, eliminating false sharing. The algorithm performs the following five steps.

1. It builds *reference groups* for array references that exhibit group-temporal and group-spatial locality.
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 (*i.e.*,the fewest cache lines accessed).
4. It achieves memory order or a *nearby* loop order through loop permutation.
5. It introduces outer loop parallelism by *tiling* the nest to eliminate false sharing (*i.e.*, strip-mining and permuting the nest).

Because we take advantage of the following observation about reuse, our analysis is greatly simplified.

> *If a loop $l$ causes more reuse than loop $l'$ at the innermost loop position, $l$ will also cause more reuse than $l'$ at any outer loop position.*

The first four steps of the algorithm therefore 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.

#### 3.1.1 Reference Groups

This step builds a set of *reference groups* for each loop in a nest based on data dependences. For every loop $l$ in the nest, it considers $l$ at the innermost position. For $l$, the algorithm places two array references in the same group if there is a dependence $\vec{\delta}$ between them that indicates *group-reuse* as follows:

1. $\exists$ *Ref*$_1$ $\vec{\delta}$ *Ref*$_2$ , and
2. a) $\vec{\delta}$ is a loop-independent dependence, or
   b) $\delta_l$ is a small constant $d$ and all other entries are zero, the $l$ loop will thus carry the dependence, or
   c) $\delta_f$, the distance in the first subscript dimension is a small constant $d$ and all other entries are zero.

Conditions a) and b) detect group-temporal reuse. Condition c) detects many forms of group-spatial reuse.

#### 3.1.2 Loop Costs

To determine the cost in cache lines of a reference group, we select a representative array reference from each group. For each candidate inner loop $l$, an array reference is classified and assigned a cost as follows:

**Loop-invariant** - if the subscripts of the reference do not vary with $l$, then it requires only 1 cache line for all iterations of $l$ (these references should end up in registers).

**Consecutive** - if only the first subscript dimension (the column) varies with $l$, then it requires a new cache line every $cls$ iterations, resulting in $trip/cls$ cache line accesses, where $l$ performs $trip$ iterations and $cls$ is the cache line size in array elements. Adjustments are made for nonunit strides less than the cache line size [5].

**Non-consecutive** – 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 $trip$ cache line accesses.

To determine the *reference cost* over the entire nest when loop $l$ is innermost, we multiply the above cost by the trip counts of the remaining loops. These loops would enclose $l$ if $l$ is innermost. The *loop cost* is simply the sum over all the reference groups for a candidate inner loop $l$.

Figure 1: Subroutine *dmxpy* from *Linpackd*

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

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 | n1 ∗ n2 |
| loop cost | 1/2 n1 ∗ n2 + n2 | 5/4 n1 ∗ n2 + n1 |

**Example.** Consider the subroutine *dmxpy* from *Linpackd* in Figure 1. 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 $m$, we 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. As illustrated in the table, the reference $y(i)$ is consecutive in the $i$ loop and invariant in the $j$ loop, the reference $x(j)$ is consecutive in the $j$ loop and invariant in the $i$ loop, and the reference $m(i, j)$ is consecutive in the $i$ loop and non-consecutive in the $j$ loop. The cost of array $m$ dominates since it accesses the largest amount of data.

#### 3.1.3 Memory Order

To determine the loop permutation which accesses the fewest cache lines, we rely on our observation; if a loop $l$ causes more reuse than loop $l'$ at the innermost loop position, $l$ will also cause more reuse than $l'$ at any outer loop position. Therefore, we 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 ordering *memory order*. Consider *dmxpy* again. Assuming $n1, n2 > 1$, loop $i$ accesses fewer cache lines than $j$ and should therefore be placed innermost, so memory order is $\{l_j, l_i\}$.

#### 3.1.4 Achieving Memory Order

Memory order specifies the permutation of the nest with the least cost. To determine if the order is a legal one, we permute the corresponding entries in the distance/direction vector. If the result is lexicographically positive (the majority of the time it is [5]), the permutation is legal and we transform the nest. If not, we use an algorithm called *NearbyPermutation* [11].

Without violating the dependence constraints and based on the loops' relative amounts of reuse, *NearbyPermutation* will place the loops carrying the most reuse as innermost as possible; *i.e.*, it is guaranteed to find a legal permutation that positions the innermost loop correctly if one exists, and 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.

#### 3.1.5 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. If additional levels of parallelism are desirable, this step may be applied repeatedly.

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 invariant or consecutive reuse, 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, and
2. if the parallel loop carries reuse, to tile it such that no false-sharing of cache lines between processors results. Reuse will thus fall locally on a processor, minimizing communication.

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 in this case produces two loops, a parallel outer *iterator* and an inner contiguous *strip*. If the loop carries only spatial reuse and the array layout in memory is known, we base the tile size on the cache line size and the alignment of the array. Otherwise, tile size is based on the size of the cache, the cache line size, and the run-time scheduler. In the experiments reported in Section 4, the optimizer makes the strips as large as possible by matching the number of tiles to the number of processors. When necessary, the optimizer permutes the iterator to an outermost position.

**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 invariant and cache line reuse, we tile the nest, *i.e.*, the optimizer strip-mines the $i$ loop by the number of processors, permutes the iterating loop to the outermost position, and parallelizes it. Figure 2 illustrates the result. Because this loop structure maximizes data locality, it minimizes communication of data between iterations and therefore between processors. In experiments on the Sequent, this structure results in speed-ups of up to 16.4 on 19 processors. The algorithm obtains linear speed-ups for kernels such as matrix multiply [11].

Figure 2: Parallelizing *dmxpy*

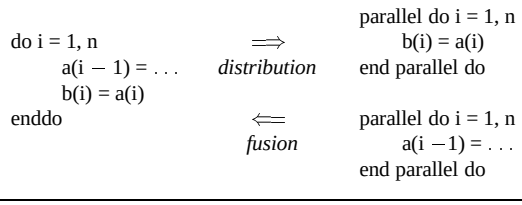
```
do j = 1, n2
    do i = 1, n1
        y(i) = y(i) + x(j) * m(i,j)
        ⇓
parallel do ii = 1, n1, tile
    do j = 1, n2
        do i = ii, min(ii + tile - 1,n1)
            y(i) = y(i) + x(j) * m(i,j)
```

**Discussion.** In our experiments, memory order is usually a legal permutation of the nest [5]. 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. These algorithms have proven effective in practice for uniprocessors and shared-memory multiprocessors [5, 11]. We define the subroutine *Optimize* to perform the above algorithms on an arbitrary loop nest.

### 3.2 Fuser: Improving the Granularity of Parallelism

Loop fusion and distribution have several purposes in our parallel code generation algorithm [5, 12]. The foremost is fusing parallel

Figure 3: Loop Distribution and Parallelization




loops together to increase the granularity of parallelism and minimize communication of shared data. In addition, loop distribution functions as an enabling transformation for effective parallelization by *Optimize*. This section describes a simple approach to incorporating fusion and distribution into the *Optimize* algorithm.

#### 3.2.1 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. For example in the left loop nest in Figure 3, there is a loop carried dependence between the two assignment statements that prevents the nest from being performed correctly in parallel. However, after distribution the two loops on the right result and both may execute 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.* [1], the process is repeated for $l_{n-1}$ 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 3, these become candidates for parallelization by *Optimize*. This algorithm is not optimal because combining distribution with loop permutation may result in a deeper distribution that in turn may be more effectively parallelized [1, 18]. This flexibility was not required in our experiments, so for simplicity it 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. Fusion is desirable between parallel loops because it may reduce communication of shared data and it reduces the amount of barrier synchronization between processors which is typically nontrivial on parallel bus-based hardware. We showed that the problem of fusing a set loops is the same, regardless if they resulted from distribution or were written that way [12].

#### 3.2.2 Loop Fusion

Loop fusion merges multiple loops with *conformable headers* into a single loop. It eliminates unnecessary barrier synchronization and reduces communication of shared data between loops. Two loop headers are conformable if they have the same number of iterations and are both either sequential or parallel loops [13]. Loop fusion is safe if it does not reverse any dependences between candidate loops. We only perform safe fusions [13]. 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 the result must be executed sequentially, as illustrated in Figure 3.

**Fusion algorithm.** Given a collection of loop nests that are candidates for fusion and do not all have conformable headers, we divide them into $k$ sets based on their *type*. Two or more loop nests with conformable headers have the same type. *Typed fusion* seeks to find the minimal number of loops resulting from a fusion in which nodes of a different type cannot be fused. We proved this general form of fusion is NP-hard [13]. When there are only two types which are differentiated 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 [12]. This restricted case arises frequently in practice. In particular, 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.

### 3.3 Kernel Parallelization

Our kernel parallelization algorithm appears in Figure 4. 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 in a procedure and produces an optimized version of the nests. For each nest, it begins by applying *Optimize*. If this step is not successful, enabling transformations are attempted. It first tries fusing all inner loops, which may enable permutation and tiling, and then tries *Optimize* again. If it is still unsuccessful, the algorithm distributes to the finest granularity. If distribution is able to form new loop nests, they may be at some inner level or the outermost level, *i.e.*, the outermost loop(s) are not necessarily included in the nests $ln_i$ resulting from the distribution in the *else* of Figure 4. *Optimize* is applied to each $ln_i$. These resultant nests are candidates for fusion. Similarly, after each outer loop $l_j$ has been optimized, the algorithm fuses the resultant nests when safe and profitable.

### 3.4 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 is fundamentally different from previous research that uses inlining. Inlining is typically performed instead of interprocedural analysis and without knowing if it will yield any optimization opportunities. Instead, we restrict the application of interprocedural transformations to cases where it enables other optimizations and therefore is expected to be profitable. This strategy, called *goal-directed* interprocedural optimization, uses interprocedural analysis to determine an optimization strategy and the in-

Figure 4: Kernel Parallelization Algorithm

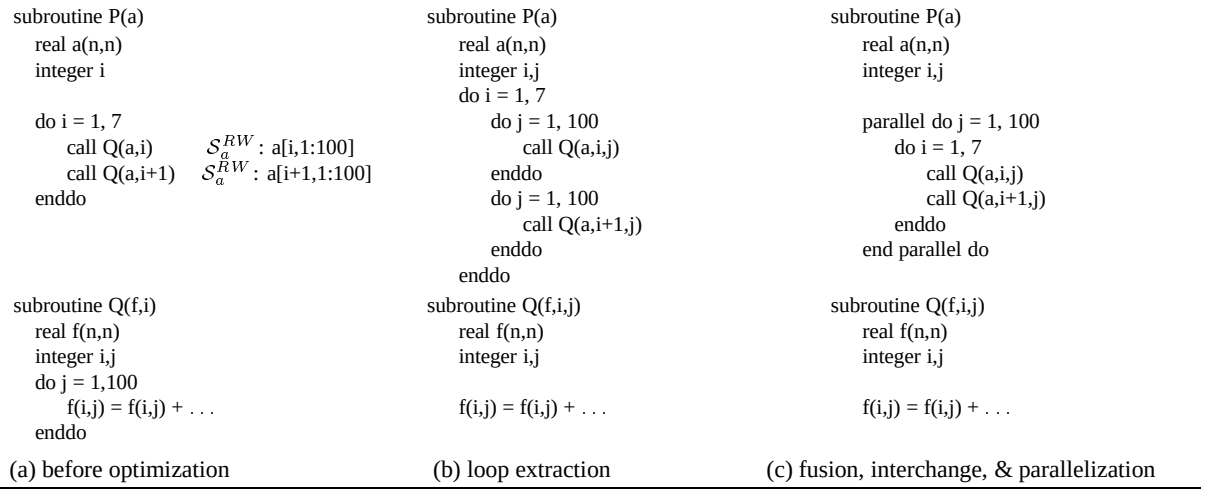
INPUT: $\mathcal{L} = \{l_1, \ldots, l_n\}$, $l_j$ adjacent nests in a procedure

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

ALGORITHM:

```
for j = 1, n
  if Optimize(l_j) succeeds
  elseif Fuse all inner loops of l_j and
    Optimize the result succeeds
  else
    {ln_1, ln_2, ..., ln_m} = Distribute(l_j)
    if m = 1 continue
    for i = 1, m
      Optimize (ln_i)
    endfor
    Fuse (ln_1, ln_2, ..., ln_m)
  endif
endfor
Fuse(l_1, ..., l_n)
```

Figure 5: Interprocedural Parallel Code Generation




terprocedural optimizations, *loop embedding*, *loop extraction*, and *procedure cloning*, to effect it.

### 3.4.1 Interprocedural Analysis

We use *section* analysis to analyze interprocedural side effects to arrays [3, 9, 10]. Our sections [18] are slightly more precise than *data access descriptors* [3]. Sections represent a restricted set of 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 [3, 4, 10, 16, 23]. 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; in practice, it often works as well as more precise techniques [10, 16].

Sections reduce the dependence problem on loops containing procedure calls to the problem on ordinary statements [10]. To increase the precision of our representation, we include access order and the precision of sections [18]. We also create an *augmented call graph* [9]. It represents procedure calls and the loop nesting structure. These extensions enable the profitability and safety of *intraprocedural* transformations (*e.g.*, fusion, permutation, and parallelization) to be determined when the nests span procedure boundaries [18]. In addition, these determinations need only inspect the results in the calling procedure.

### 3.4.2 Loop Embedding and Loop Extraction

*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. They expose the loop structure to optimization without incurring all the costs of inlining. Just as inlining is always safe, these transformations are always safe. They require an outer loop which encompasses all the other statements in the called procedure. Using extended section analysis, the kernel optimizer and thus *Optimize* can test for permutation, fusion, distribution, tiling, and parallelization for loop nests that span procedures [18]. If an optimization is applicable across procedure boundaries, then we use embedding or extraction to enable it. The same analysis could decide when to perform inlining.

**Example.** Consider Figure 5(a) where the calls to $Q$ are annotated by $\mathcal{S}_a$, the sections of array *a*. In this example, the same section of *a* is both modified and read at each call. Using the sections, dependence testing in $P$ reveals the dependence between the two calls, $\vec{\delta} = \{1, 0\}$, carried by the $i$ loop. Notice we have the distance for the $j$ loop, even though it results from code in subroutine $Q$. We call the kernel algorithm on the nest rooted at loop $i$ whose scope includes subroutine $Q$. *Optimize* tries to interchange the $i$ and $j$ loops to put the unit stride access on the inner loop and the parallel $j$ loop at the outermost position. It fails because the loop structure, as revealed by the augmented call graph, prevents interchange. The algorithm next determines that fusion of the inner loops is safe and that fusion enables interchange and parallelization by *Optimize*. To perform the optimizations, the loops are placed in the same procedure via loop extraction. See Figure 5(b) and (c).

**Embedding versus Extraction.** 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. However, if loop nests originating from more than one call site are needed to perform an optimization, extraction is required, as illustrated in Figure 5(b).

### 3.4.3 Procedure Cloning

Procedure cloning generates multiple copies of a procedure each tailored to its calling environment [7]. Even without embedding or extraction, cloning is necessary for interprocedural parallel code generation 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 6(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 6(b).

## 3.5 Whole Program Parallelization

The judicious application of interprocedural optimizations does not change the basic structure of the kernel parallelization algorithm. However, testing the safety and profitability of each of the transformations is complicated somewhat. Our strategy separates legality and profitability tests from the mechanics of the transformations [18]. The safety tests depend on the precision of the dependence information and the sections analysis. For permutation, the dependences must be precise enough in the caller to determine if

Table 1:

| Program Test Suite | | | | |
|---|---|---|---|---|
| *Name* | *Description* | *lines* | *authors* | *affiliation* |
| Seismic | 1-D Seismic Inversion | 606 | Michael Lewis | Rice |
| BTN | BTN Unconstrained Optimization | 1506 | Stephen Nash & Ariela Sofer | George Mason |
| Erlebacher | ADI Integration | 615 | Thomas Eidson | ICASE |
| Interior | Interior Point Method | 3555 | Guangye Li & Irv Lustig | Cray Research, Princeton |
| Control | Optimal Control | 1878 | Stephen Wright | Argonne |
| Direct | Direct Search Methods | 344 | Virginia Torczon | Rice |
| ODE | Two-Point Boundary Problems | 3614 | Stephen Wright | Argonne |
| Multi | Multidirectional Search Methods | 1025 | Virginia Torczon | Rice |
| Banded | Banded Linear Systems | 1281 | Stephen Wright | Argonne |
| Linpackd | Linpackd benchmark | 772 | Jack Dongarra | Tennessee |

Figure 6: Cloning for Correct Parallel Code Generation

```
procedure C                    procedure C
    call Q                         call Qclone
    do i = 1, n                    parallel do i = 1, n
        call Q                         call Q
    enddo                          end parallel do

procedure Q                    procedure Q
    do j = 1, n                    do j = 1, n
        . . .                          . . .
    enddo                          enddo

                               procedure Qclone
                                   parallel do j = 1, n
                                       . . .
                                   end parallel do
```

(a) original (b) parallelized with cloning

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 transformation is unsafe. Testing distribution of a loop into a call is more difficult since the dependence information in the caller is not of fine enough granularity. Additional dependence testing in the call would need to be performed to determine its safety. Therefore, we only use fusion, permutation, and tiling in combination with interprocedural transformations.

## 4 Experiment

For our experimental validation, we do *not* apply our parallel code generation algorithm to 'dusty deck' programs whose authors never considered parallel execution. Although, it will be successful in some instances. Instead, we measure the ability of our optimizations to detect and exploit parallelism that is known to exist, *i.e.*, the compiler's ability to match or exceed performance of parallel programs written by programmers who thought and cared about their parallel performance. Based on our successes and failures, we also determine a parallel programming style from which compilers are more likely to achieve or improve hand-tuned performance for shared-memory, bus-based parallel machines.

We designed the following experiment to measure the efficacy of our automatic parallel code generator. We assembled programs written for a variety parallel machines. We applied our algorithm to *sequential* versions of these programs. The compiler was required to use its analysis and algorithms to parallelize the program and could not rely on user *assertions* that for example, a loop is parallel. We executed and compared the original hand-parallelized version, the sequential version, and the automatically parallelized version on a 20 processor Sequent Symmetry S81. Our results are very encouraging. Our algorithm exceeds or matches hand-coded parallel programs for eight of the nine programs in our suite.

## 5 Methodology

### 5.1 The Programs

We solicited programs from scientists at Argonne National Laboratory and from users of the Sequent and Intel iPSC/860 at Rice. No screening process was performed; we used all the programs that were submitted. The 9 applications programs that were volunteered had been 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. There are 9 programs on which we will focus, plus we added *Linpackd* since it is well known to contain parallelism. 8 programs out of 9 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 attention was paid to their performance. They are described in more detail elsewhere [18].

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. Of interest here is that it may be only well structured codes were volunteered. Maybe the authors of poorly structured ones did not want to expose their codes to a critical eye. Fortunately, this furthers our arguments for a modular machine-independent programming style, rather than thwarting the experiment.

### 5.2 Creating Program Versions

For each of the programs that were originally written for the Sequent, this version is the original parallel version. For the programs written for other architectures, we modified any parallelization directives to reflect the equivalent Sequent directives. In *Erlebacher*, the parallelism is not explicit. Here, we performed a naive parallelization of outer loops to create the parallel version.

We created the sequential version of each program simply by ig-

noring all the parallel directives. Directives included parallel loops, variable privatization, and critical sections. On the sequential version, we then used the advanced analysis and transformations available in our interactive parallel programming tool, the ParaScope Editor (PED) [6, 14], to perform our parallel code generation algorithm. Although the individual transformations were automated, the code generation algorithm was not.

### 5.3 Automatic Parallel Code Generation

**Analysis.** To overcome gaps in the current implementation of program analysis in PED, we imported dependence information from PFC. PFC is the Rice system for automatic vectorization [2]. PFC's analysis is more mature and includes important features which were not yet implemented in PED. It performs advanced symbolic dependence tests. It also computes interprocedural constants, interprocedural symbolics and interprocedural MOD and REF information for simple array sections [10]. PFC produces a file of dependence information that is converted into PED's internal representations.

**Transformation.** Our implementation was not complete when these experiments were performed. We used the augmented call graph, program analysis, and the transformations available in PED, to apply our parallel code generation algorithm. In PED, transformations have two phases, *i.e.*, the mechanics of a transformation are separated from its test for correctness. Users select a transformation and in response, PED determines the safety and profitability of the transformation. If it 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. Except for the interprocedural transformations, the individual transformations used by our parallel code generation algorithm are implemented in PED, but the driver was not. Using PED, we attempted the transformations as specified by the algorithm and applied them only when PED assured their correctness. We kept optimization diaries for each program.

### 5.4 Execution Environment

We ran and compared all three versions on a Sequent Symmetry S81 with 20 processors. The Sequent has a simple parallel architecture which does not include vector hardware, allowing our experiments to focus solely upon medium and large grain parallelism. Each processor has its own 64Kbyte two-way set-associative cache and is connected to the bus. The cache line size is 4 words. The Sequent has a flexible compiler that allows the program to completely specify parallelism, [19].

To introduce parallelism into the programs, we used the parallel loop compiler directives. 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, the Weitek 1167 floating-point accelerator was used and optimization was suppressed.

## 6 Results

We measured execution times for:

***seq:*** the sequential version of the program,

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

***auto:*** the version obtained using our optimization algorithm.

We also measured subparts of a program if there were differences between the automatically parallelized version and the user parallelized version. For example, if the automatic version parallelized a nest and the hand-coded version did not, the execution time for that nest is measured in all versions. These measurements reveal the magnitude of the particular success and failures of our algorithm. They are labeled in the tables as follows.

***The Entire Application:*** execution time of the application.

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

***Degradations:*** execution times in regions where the automatically generated version could not detect parallelism specified by the hand-coded version.

The elapsed times for the entire applications were measured in seconds using the system call *secnds*. Execution times for program subparts that differed were measured using the microsecond clock, *getusclk*. From these times we computed speed-ups for the parallel programs. Some of the differences between program versions occurred on at inner loops. In these cases, we measured the performance of the outermost enclosing loop in order to disrupt the execution as little as possible. The speed-ups of these optimized versions are under reported. Table 2 contains the speed-ups over the sequential version of the parallel versions. The execution times in seconds of all the program and program subpart versions appear in Table 3.

In Table 2 and 3, a blank entry means that no program or program subpart fell in that category. For example, the automatically parallelized version and the user parallelized version did not differ for *Control*, *Direct*, and *ODE* and therefore we did not measure any subparts.

### 6.1 Interpretation and Analysis of Results

As can be seen in the percent change column ($\Delta$) in Table 2, except for *Multi*, the automatically generated programs either performed as well or better than the hand-coded parallel versions. 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 improvements 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 three programs, *Interior*, *BTN* and *Multi*, users found more parallelism than our automatic techniques. For *Interior*, these degradations did not have much effect on overall execution time. If we

Table 2:

Speed-ups over the sequential version on a 19 Processor Sequent

| | *Improvements* | | *Degradations* | | *Entire Application* | | |
|---|---|---|---|---|---|---|---|
| *Name* | *hand* | *auto* | *hand* | *auto* | *hand* | *auto* | $\Delta$ |
| 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% |
| Banded† | | | | | $\star$ | 1.0 | $\star$ |
| Linpackd | | 16.5 | | | | 9.2 | *NA* |

$\star$ : result not obtainable; † : 8 processors

Table 3:

| Execution Times in seconds | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | *Improvements* | | | *Degradations* | | | *Entire Application* | | |
| | *seq* | *hand* | *auto* | *seq* | *hand* | *auto* | *seq* | *hand* | *auto* |
| 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 |
| Banded† | | | | | | | ⋆ | ⋆ | ⋆ |
| Linpackd | 517.87 | | 31.43 | | | | 547.59 | | 59.43 |

⋆ : result not obtainable; † : 8 processors

look at the corresponding execution times in Table 3, it is apparent that both the degradations and improvements only affect the overall execution time by 3% or less. Each of *BTN* and *Multi* contain parallel loops with critical sections that update shared variables. Analysis techniques exist that can properly identify the parallelism [22], 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*, it was the only outer parallel loop and accounted for 86% of the sequential running time and 30% of the parallel running time.

Our automatic techniques could not discover any of the parallelism in *Banded*. This program was written for an Alliant FX/8 and converting three parallel loops to the equivalent Sequent parallel loop directives resulted in a runtime error. 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 and will thwart even advanced dependence analysis. It is most likely responsible for the runtime error on the Sequent. The 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.

### 6.1.1 Program Analysis

Out of approximately 650 loops which made up 341 nests, dependence analysis detected approximately 400 parallel loops at all levels of nesting out of the 650 loops (62%) and 220 out of 340 (65%) parallel loops in the outermost position of a nest[1]. 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 generally able to find it. Compilers are however more thorough and meticulous than the average user. As illustrated in Tables 2 and 3, the loops with differences turned out not to impact performance significantly one way or the other. Instead, as we discuss in Section 6.1.2, the improvements experienced by the automatically parallelized versions were due to our optimization strategy.

**Interprocedural Sections.** The analysis of interprocedural sections proved to be a very important feature of the current system. All but one of the 9 programs contained one or more parallel loops with a call. Out of a total of 277 procedure calls made by these 9 programs, 123 (44%) of these calls were nested inside loops and 52 (19%) of these loops were parallel. Given our execution results, section analysis detected parallel loops with calls as well as programmers. However, flow-sensitive section analysis was needed to parallelize *BTN*, *ODE*, and *Banded*. In *ODE* and *BTN*, we determined the array kill by hand since it is a very simple case that a reasonable implementation would catch. These results indicate parallel programmers use procedures calls to manage complexity. In addition, both flow-sensitive and flow-insensitive interprocedural section analysis are needed to effectively analyze and optimize this modular parallel programming style.

**Index Arrays.** Five of the programs in this test suite used index arrays that were permutations of the index set. Several of these were monotonic non-decreasing with a regular, well defined pattern. In three programs, automatic parallelization would not have been possible without using user assertions and the testing techniques developed in our earlier research [17]. The other two programs used them in a way that did not affect parallelization.

**Linearized Arrays.** The programs *ODE* and *Banded* contained linearized arrays and used 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 performance. Programmers will then have an incentive to use multiply dimensioned arrays when appropriate. If array linearization improves performance, as on the Cray YMP, then the compiler should perform it.

### 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 7(a), each of the subroutines *setvz*, *ftau*, and *fzeta* contains an outer, enclosing paral-

[1] The statistics in this section do not include *ODE* or *Banded*.

Figure 7: Cloning for Correct Parallel Code Generation

```
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

lel 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*, see Figure 7(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. This tradeoff is difficult for programmers to make; the user successfully parallelized 24 outer loops in which the two do not conflict, but failed on the 3 where they did.

Similarly, some 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[2].

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

## 7 Related Work

Our core technique, *Optimize*, bears the most similarity to Wolf & Lam's research [24, 25]. Their algorithm is potentially more precise and uses skewing and reversal, but in their experiments on data locality, skewing was never useful [24]. Our algorithm is simpler, can take advantage of known loop bounds, 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. Their work includes very few experimental results for the parallelization algorithm, and they do not perform fusion, distribution, or any interprocedural analysis and transformations.

Not many studies of parallelizing optimizers have been published. Commercially available parallelizing compilers typically do not reveal their optimization strategies, much less publish them. The studies of which we are aware are from Illinois and Stanford [8, 15, 20, 21].

The Illinois studies are traditional [8, 15]; they extend Kap, an automatic parallelizer, and then use it to parallelize the Perfect Benchmarks, dusty deck programs. Their target architecture is Cedar, a shared-memory parallel machine with cluster memory and vector processors. 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. In contrast, our study uses a cleanly defined and efficient algorithm. Both studies however would benefit greatly from complete implementations. The Illinois papers also lack statistics about the effectiveness and applicability of optimizations and analysis. Their interprocedural analysis results de facto from inlining or is performed by hand.

Singh & Hennessy used the Alliant FX/8, the Encore Multimax and their Fortran compilers [20, 21]. The compiler algorithms are again unpublished. The FX/8 has cluster memory instead of local caches, so their parallelization problem was easier than the one considered here. On the Encore, the slow processors minimized the impact of its small local caches. Singh & Hennessy considered dusty deck programs and by inspection 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 success of our techniques reveals that even after users perform algorithm restructuring for parallelism, there is performance to be gained.

## 8 Conclusions

This paper presents a new parallelization algorithm that balances parallelism and communication, *i.e., data locality*. It uses a simple, yet effective strategy to introduce locality, exploit parallelism, and maximize the granularity of parallelism. Of particular importance to its effectiveness is interprocedural section analysis. We evaluated the algorithm using hand-parallelized programs and a compiler generated parallel version. We compare the two parallel versions. Our results are promising. The algorithm improves performance 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 the programming style many parallel programmers are using can be portable and can be analyzed and optimized by an advanced compiler. Further experimentation however is needed and we are in the process of completing our implementation.

## 9 Acknowledgements

I especially want to thank Ken Kennedy, who provided impetuous and guidance for much of this research. I thank Mary Hall, Chau-Wen Tseng, Preston Briggs, Paul Havlak, and Nat McIntosh for their comments and support throughout the development of this work. Paul Havlak's implementation of regular sections proved invaluable. To all of these people go my thanks.

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

## References


[1] 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.

[2] 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.

[3] 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 Program Language Design and Implementation*, Portland, OR, June 1989.

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

[5] S. Carr, K. S. McKinley, and C. Tseng. Compiler optimizations for improving data locality. In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 1994. *To appear*.

[6] 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.

[7] K. Cooper, K. Kennedy, and L. Torczon. The impact of interprocedural analysis and optimization in the $\mathbb{R}^n$ programming environment. *ACM Transactions on Programming Languages and Systems*, 8(4):491–523, October 1986.

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

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

[10] 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.

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

[12] K. Kennedy and K. S. McKinley. Maximizing loop parallelism and improving data locality via loop fusion and distribution. In *Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[13] 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.

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

[15] 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.

[16] 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.

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

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

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

[20] 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.

[21] 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.

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

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

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

[25] 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.