Relaxing SIMD Control Flow Constraints Using Loop Transformations

Reinhard von Hanxleden
Ken Kennedy

CRPC-TR92207-S
April 1992

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

Appeared in Proceedings of the SIGPLAN '92 Conference
on Programming Language Design and Implementation,
SIGPLAN Notices, 27(7), July 1992.

Relaxing SIMD Control Flow Constraints using Loop Transformations*†

Reinhard v. Hanxleden Ken Kennedy

Department of Computer Science
Rice University
Houston, TX 77251-1892

Abstract

Many loop nests in scientific codes contain a parallelizable outer loop but have an inner loop for which the number of iterations varies between different iterations of the outer loop. When running this kind of loop nest on a SIMD machine, the SIMD-inherent restriction to single program counter common to all processors will cause a performance degradation relative to comparable MIMD implementations. This problem is not due to limited parallelism or bad load balance, it is merely a problem of control flow.

This paper presents a loop transformation, which we call *loop flattening*, that overcomes this limitation by letting each processor advance to the next loop iteration containing useful computation, if there is such an iteration for the given processor. We study a concrete example derived from a molecular dynamics code and compare performance results for flattened and unflattened versions of this kernel on two SIMD machines, the CM-2 and the DECmpp 12000. We then evaluate loop flattening from the compiler's perspective in terms of applicability, cost, profitability, and safety. We conclude with arguing that loop flattening, whether performed by the programmer or by the compiler, introduces negligible overhead and can significantly improve the performance of scientific codes for solving irregular problems.

*This research was supported by the Center for Research on Parallel Computation, a National Science Foundation Science and Technology Center. Use of the DECmpp 12000 was provided by the Center for Research on Parallel Computation under NSF Cooperative Agreement No. CCR-8809615 with support from Digital Equipment Corporation.

†From the *Proceedings of the ACM SIGPLAN '92 Conference on Program Language Design and Implementation*, San Francisco, CA, 1992.

1 Introduction

In the process of parallelizing scientific programs, it is common to find loop nests in which the outer loop can run in parallel but the amount of computation in the inner loop varies for different iterations of the outer loop. This causes a load balancing problem because the outer loop iterations have to be partitioned among the processors in such a manner that each processor has a roughly equal amount of work to do. Load balancing is a difficult problem in itself which has been frequently addressed in the literature [1, 9, 11]. Once this problem is solved, we can usually expect good performance when running such a loop nest on a shared-memory or distributed-memory MIMD (Multiple Instruction, Multiple Data) machine.

However, this kind of loop nest causes special problems for SIMD (Single Instruction, Multiple Data) architectures because of the restricted control flow on these machines [4]. If the number of iterations of the inner loops varies from one outer loop iteration to the next, then the restriction to a common program counter makes a naïve SIMD implementation inefficient. As observed in a case study implementing an image processing algorithm on the Massively Parallel Processor [23, page 143]: "... the complexity of each iteration in the SIMD environment is dominated by the largest region in the image. This is due to the fact that the synchronous execution of instructions forces each processor to either perform the operation or wait in an idle state until all processors have completed the operation." To overcome this limitation, we propose a new technique, which we call *loop flattening*, that, roughly speaking, amounts to lifting the innermost loop body up into the outer, parallel loop by merging the control of the inner loops with the control of the outer loop.

This paper is organized as follows. Section 2 describes the different variants of pseudo Fortran which we will use throughout this document. Section 3 presents a small example to illustrate the kind of problem we are interested in and gives a first glance at loop

flattening, which Section 4 elaborates at at a more general level. Section 5 examines the applicability of loop flattening for a nonbonded force kernel, taken from a typical molecular dynamics program, which we implemented on both the CM2 and the DECmpp. Section 6 evaluates loop flattening from the compiler perspective. We conclude with a discussion of related work in Section 7.

2 Languages

The concepts introduced here apply to a broad range of languages. We will give program examples in different variants of pseudo Fortran:

F77 - Strictly sequential Fortran 77 (possibly a "dusty deck" program).

F77D - F77 enhanced with distribution statements as proposed in Fortran D [8] and High Performance Fortran [12]. An important goal of F77D is to provide a basis for efficient compilation towards both MIMD and SIMD distributed memory machines, so it should not contain any constructs which are specific to either architecture.

F77$_{MIMD}$ - A Fortran 77 version to run on a MIMD machine, which assumes a separate name space for each processor.

F90$_{SIMD}$ - A Fortran 90 version to run on a SIMD machine, similar to Connection Machine Fortran [21] or MasPar Fortran [16]. There are two important differences to the F77 variants:

- By default, scalars of the F77 version will be *replicated* in the F90$_{SIMD}$ version; *i.e.*, they will be declared as vectors of size P, where processor p owns the p-th element.
- In keeping with Fortran 90 convention, omitted array indices refer to all elements of an array dimension, and an unsubscripted array reference refers to all array elements.

For enhancing readability of the F90$_{SIMD}$ examples, we extend the language constructs which are typically implemented by vendors in several ways:

- The FORALL construct cannot only be applied to single statements, but also to blocks. The general form of this extension can be interpreted differently depending on the semantics chosen for the case where different iterations modify the same set of data; our examples, however, will avoid these access interferences.
- DO-ENDDO's, DO-WHILE's, IF's, WHERE's, and FORALL's can be nested freely within each other.

```
C   P1 – sequential version
    DO i = 1, K
      DO j = 1, L(i)
        X(i,j) = i * j
      ENDDO
    ENDDO
```

Figure 1: Original loop nest **EXAMPLE**.

```
C   P2 – Fortran D version
    DECOMPOSITION XD(K,Lmax), LD(K)
    ALIGN X with XD, L with LD
    DISTRIBUTE XD(BLOCK,*), LD(BLOCK)

    DO i = 1, K
      DO j = 1, L(i)
        X(i,j) = i * j
      ENDDO
    ENDDO
```

Figure 2: **EXAMPLE** in F77D.

- WHILE loops can be controlled by an array of booleans (instead of just a scalar boolean), if the different array elements are guaranteed to have identical values.

3 Example of Loop Flattening

Consider the contrived F77 loop nest in Figure 1, henceforth called **EXAMPLE**. This clearly is a dependence free, parallelizable loop, where the number of inner loop iterations depends on the current iteration of the outer loop. Let K be 8 and let $L(1{:}8)$[1] have the values 4,1,2,1,1,3,1,3, respectively. Assuming $P = 2$ processors and the *owner computes rule*, where in all assignment statements the right hand side expression is computed by the processor which "owns" the left hand side variable, we can in this case just distribute L and the rows of X blockwise to achieve perfect load balance. This is illustrated in the F77D program in Figure 2, which assigns $L(1{:}4)$, $X(1{:}4,1{:}4)$ to the first processor and $L(5{:}8)$, $X(5{:}8,1{:}4)$ to the second processor. The owner computes rule results in partitioning the iteration space among the two processors, so each processor executes only some iterations of the outer loop.

For a MIMD machine, the Fortran D compiler would derive the F77$_{MIMD}$ program shown in Figure 3. Each processor executes the loop nest independently, needing a total of

$$TIME_{MIMD} = \max_{p=1,2} \sum_{i=1}^{4} L(i + 4(p-1)) = 8 \qquad (1)$$

[1] Fortran 90 notation for the array elements $L(1) \ldots L(8)$.

```
C    P3 – MIMD version
     DO i = 1, 4
       DO j = 1, L'(i)
         X'(i,j) = i * j
       ENDDO
     ENDDO
```

Figure 3: **EXAMPLE** in $F77_{MIMD}$. X and L are renamed to X' and L' to reflect that there is no common name space any more. On processor p, $p = 1, 2$, $L'(i)$ corresponds to $L(i + 4(p - 1))$, and $X'(i, j)$ corresponds to $X(i + 4(p - 1), j)$.

Time	1	2	3	4	5	6	7	8
i_1	1	1	1	1	2	3	3	4
j_1	1	2	3	4	1	1	2	1
i_2	1	2	2	2	3	4	4	4
j_2	1	1	2	3	1	1	2	3

Figure 4: MIMD execution trace for **EXAMPLE** loop; i_p and j_p denote i and j on processor p.

inner loop iterations. This is illustrated in the trace in Figure 4.

A $F90_{SIMD}$ version could be derived from the F77D program by just changing the outer DO loop to a FORALL loop. This would result in a partitioning of the iteration space, similar to the F77D version. For expository reasons, we will give a slightly different but equivalent $F90_{SIMD}$ version which takes the data decomposition and the number of processors already into account and thus directly reflects the control flow for $K = 8$ and $P = 2$. As in the $F77_{MIMD}$ version, we change the upper bound of the outer loop from $K = 8$ to $K/P = 4$ and let each processor execute all iterations of the loop. We continue to use the loop index i in control flow related statements; to allow the different processors to operate on different data, we introduce an auxiliary induction variable, i', which replaces i in non-control flow statements. The result is shown in Figure 5.

Note how we had to transform the inner DO loop

```
C    P4 – naive SIMD version
     DO i = 1, 4
       i' = i + [0,4]
       DO j = 1, max(L(i'))
         WHERE (j ≤ L(i'))  X(i',j) = i' * j
       ENDDO
     ENDDO
```

Figure 5: **EXAMPLE** in $F90_{SIMD}$. [0,4] denotes the two-element vector containing 0 and 4.

```
C    P5 – flattened SIMD version
     i = [1,5]
     K = [4,8]
     j = 1
     WHILE ANY(i ≤ K)
       WHERE (i ≤ K)
         X(i,j) = i * j
         WHERE (j = L(i))
           i = i + 1
           j = 1
         ELSEWHERE
           j = j + 1
         ENDWHERE
       ENDWHERE
     ENDWHILE
```

Figure 7: **EXAMPLE** in flattened $F90_{SIMD}$.

due to the single SIMD control flow. To make sure that each processor can perform all of its iterations, the upper bound $L(i')$ had to be changed into the maximum of $L(i')$ over all processors. This in turn necessitated a guard for the loop body which tests whether this processor is still involved in the current inner loop iteration or whether it is masked out and sits idle, possibly to participate again in later iterations.

We will refer to this transformation, which can be applied to other loop types as well, as *SIMDizing* a loop. It is a straightforward consequence of the SIMD restricted control flow, yet it is the crucial motivation for the concepts introduced in this paper. The outer loop does not have to be SIMDized in this particular case because we know that each processor works on exactly four rows of X and therefore has to execute the outer loop the same number of times. Loop SIMDizing has the effect that our $F90_{SIMD}$ program has to execute

$$TIME_{SIMD} = \sum_{i=1}^{4} \max_{p=1,2} L(i + 4(p - 1)) = 12 \qquad (2)$$

iterations. Roughly speaking, our time bound has increased from a maximum over sums to a sum over maxima. This becomes apparent when considering the execution trace shown in Figure 6.

Since the equivalent MIMD implementation performs significantly better, this bad running time can not be explained with lack of parallelism or bad load balance. To overcome this purely control flow related problem, we apply *loop flattening*, which will be introduced at a more general level in the next section. The result is shown in Figure 7. This version achieves the same time bound as in the MIMD implementation, needing only eight steps as shown in the trace in Figure 4.

The reader might have noticed that the loop body shown in Figure 7 is now always executed at least once for each outer loop iteration, which is equivalent to

Time	1	2	3	4	5	6	7	8	9	10	11	12
i_1	1	1	1	1	2			3	3	4		
j_1	1	2	3	4	1			1	2	1		
i_2	1				2	2	2	3		4	4	4
j_2	1				1	2	3	1		1	2	3

Figure 6: Execution trace for unflattened example loop; i_p, j_p denote the actual iteration counts of processor p, no entry means "idle."

init_1	i = 1
WHILE test_1	**WHILE** (i ≤ K)
init_2	j = 1
WHILE test_2	**WHILE** (j ≤ L(i))
$\mathcal{BODY}$	X(i,j) = i * j
increment_2	j = j + 1
ENDWHILE	**ENDWHILE**
increment_1	i = i + 1
ENDWHILE	**ENDWHILE**

Figure 8: Generic loop nest `GENNEST` (left) and corresponding `EXAMPLE` (right); original version after normalization.

assuming $L(i) \geq 1$ for all i. Even though this is correct in our example, a more general loop flattening does not rely on this assumption, as we will see in the next section.

4 General Loop Flattening

Assume that we are given two fully parallelizable nested loops like in the previous section; an extension of the following to deeper loop nests is straightforward. Each of the loops might be structured as a WHILE loop, a DO-WHILE loop, a simple DO or FORALL loop, or it might use conditional GOTO's. The transformation described here can be done either at the F77/ F77D level or at the F90_{SIMD} level. For simplicity and generality, we will present it here on the F77 level. A corresponding F90_{SIMD} version can always be directly derived by SIMDizing loops and replacing IF's with WHERE's.

As a first step, we *normalize* both loops by breaking their control pattern into three *phases* for each nesting level l; an initialization phase $init_l$, a guard $test_l$, and an incrementing step $increment_l$. For example, a control pattern like `DO var = lo, hi, stride` would be broken into $init_l \equiv$ `var = lo`, $test_l \equiv$ `(var ≤ hi)`, and $increment_l \equiv$ `var = var + stride`. The resulting loop nest `GENNEST` is shown in Figure 8, along with the corresponding version of the `EXAMPLE` from the previous section (of course, we usually expect $\mathcal{BODY}$ to contain more computational work than in `EXAMPLE`).

Since `GENNEST` conservatively tests for loop completion before entering the loop body, all loops can be brought into this normal form. To estimate the running time of the above code on P processors, for processor p let K_p be the number of outer loop iterations and L_p^i be the number of inner loop iterations for the i-th outer loop iteration. A straightforward MIMD version would then finish after

$$TIME_{MIMD} = \max_{p=1\ldots P} \sum_{i=1}^{K_p} L_p^i \qquad (1')$$

iterations.

A F90_{SIMD} version could be derived by SIMDizing both WHILE loops and would execute

$$TIME_{SIMD} = \sum_{i=1}^{\max_{p=1}^{P} K_p} \max_{p=1\ldots P} L_p^i \qquad (2')$$

iterations. Again, if the number of iterations of the inner loop varies from one outer loop iteration to the next, then the restriction to a common program counter makes this SIMD implementation inefficient.

Since we do not know whether the evaluation of $test_l$ has any side effects, we introduce flags t_l to store the results of evaluating the conditions $test_l$ before we make any other transformations, see Figure 9. So far, control flow is still unchanged.

The key idea of loop flattening is to make sure that each processor has a chance to advance to the next loop iteration where it participates in the execution of $\mathcal{BODY}$ before the control flow actually reaches $\mathcal{BODY}$. One requirement which follows immediately is that control variables (iteration counts etc.) are replicated to allow individual processors to advance independently to the next outer loop iteration whenever they are done with the current inner loop. Furthermore, we have to take $\mathcal{BODY}$ out of the part of the loop nest which handles the transition between different iterations of the inner and outer loop. Each processor should be able to execute $\mathcal{BODY}$ whenever it has still work left to do in this loop nest and the control flow reaches $\mathcal{BODY}$. In other words, $\mathcal{BODY}$ should be executed whenever t_1 is true, independent of t_2. The

```
init1                          i = 1
t1 = test1                     t1 = (i ≤ K)
WHILE t1                       WHILE t1
  init2                          j = 1
  t2 = test2                     t2 = (j ≤ L(i))
  WHILE t2                       WHILE t2
    BODY                           X(i,j) = i * j
    increment2                     j = j + 1
    t2 = test2                     t2 = (j ≤ L(i))
  ENDWHILE                       ENDWHILE
  increment1                     i = i + 1
  t1 = test1                     t1 = (i ≤ K)
ENDWHILE                       ENDWHILE
```

Figure 9: `GENNEST/EXAMPLE`, with guard variables.

```
init1                          i = 1
t1 = test1                     t1 = (i ≤ K)
IF t1 THEN init2               IF t1 then j = 1
WHILE t1                       WHILE t1
  t2 = test2                     t2 = (j ≤ L(i))
  WHILE (t1 ∧¬ t2)               WHILE (t1 ∧¬ t2)
    increment1                     i = i + 1
    t1 = test1                     t1 = (i ≤ K)
    IF t1 THEN                     IF t1 THEN
      init2                          j = 1
      t2 = test2                     t2 = (j ≤ L(i))
    ENDIF                          ENDIF
  ENDWHILE                       ENDWHILE
  IF t1 THEN                     IF t1 THEN
    BODY                           X(i,j) = i * j
    increment2                     j = j + 1
  ENDIF                          ENDIF
ENDWHILE                       ENDWHILE
```

Figure 10: `GENNEST/EXAMPLE`, after flattening.

flattened loop version meeting these goals is shown in Figure 10.

As the reader might verify, we still execute exactly the same instructions in the same order and the same number of times as we did in the original loop nest. We also still have two nested loops. However, $\mathcal{BODY}$ is lifted out of the inner loop. The inner loop now contains just the control structure to let each processor advance to the next iteration in which it actually executes $\mathcal{BODY}$. In other words, *the processors still have to run through $\mathcal{BODY}$ and the rest of the loop nest in lockstep, but now they may be executing effectively different loop iterations.*

The above transformation is the most general, conservative one. It can be optimized for several special cases; one common case is that

```
init1                          i = 1
init2                          j = 1
WHILE test1                    WHILE (i ≤ K)
  BODY                           X(i,j) = i * j
  increment2                     j = j + 1
  IF NOT test2 THEN              IF NOT(j ≤ L(i))
    increment1                     i = i + 1
    init2                          j = 1
  ENDIF                          ENDIF
ENDWHILE                       ENDWHILE
```

Figure 11: `GENNEST/EXAMPLE`, flattened and optimized.

```
init1                          i = 1
init2                          j = 1
WHILE test1                    WHILE (i ≤ K)
  BODY                           X(i,j) = i * j
  IF done2 THEN                  IF (j = L(i))
    increment1                     i = i + 1
    init2                          j = 1
  ELSE                           ELSE
    increment2                     j = j + 1
  ENDIF                          ENDIF
ENDWHILE                       ENDWHILE
```

Figure 12: `GENNEST/EXAMPLE` after further optimization.

1. $test_1$, $test_2$ and $init_2$ have no side effects, and that
2. for each outer loop iteration, the inner loop is executed at least once.

Then we can safely transform the code into the simpler version shown in Figure 11.

If it further is the case that

3. we can replace the guard $test_2$ with a test whether we are in the last inner iteration, $done_2$ (for example, in `DO var = lo, hi, stride`, we can replace $test \equiv$ `(var ≤ hi)` with $done \equiv$ `(var = hi)`),

then we can save the last execution of $increment_2$, as shown in Figure 12. The SIMDized equivalent `EXAMPLE` of this version was shown in Figure 7.

5 Case Study with Molecular Dynamics

The transformation described in the previous section should be profitable whenever some processors sit idle in an inner loop and still have work to do in later iterations of the outer loop. This seems to be a situation potentially occurring in many scientific programs solving

```
DO At1 = 1, N
  F(At1) = 0
  DO pr = 1, pCnt(At1)
    At2 = partners (At1, pr)
    F(At1) = F(At1) + Force (At1, At2)
  ENDDO
ENDDO
```

Figure 13: F90$_{SIMD}$ version of the nonbonded force calculation `NBFORCE`.

irregular problems [2, 19, 22, 23]. One example is the GROMOS molecular dynamics program, which contains several interesting kernels of this kind [6, 7, 10]. Here we want to focus on the calculation of the nonbonded forces between individual pairs of atoms.

5.1 The application

Since the nonbonded forces between pairs of atoms quickly decrease as the distances between them increase, they are usually approximated by considering only pairs of atoms which are closer together than a predefined *cutoff* radius; typical values are in the order of 10 Å. Still, in the GROMOS code this kernel typically accounts for about 90% of the overall simulation cost. For atom i, the atoms close enough to i are precomputed into an array $partners(i, 1{:}pCnt(i))$. This precomputation can be quite expensive in itself and is usually done only every k simulation steps, where $k = 10$ is one common value [20].

Figure 13 shows a F77 version `NBFORCE` for calculating the nonbonded forces between N atoms. This code can be parallelized by partitioning the set of all atoms into P disjoint subsets and assigning one subset to each processor p. To achieve load balancing, the sum over the number of the partners of the atoms in a processor's subset should be roughly equal across the processors. Furthermore, to achieve locality and scalability, the atoms within each subset should be closely together in space.

Figure 14 shows a F90$_{SIMD}$ program which lays out the data in a cyclic fashion. If we assume for simplicity that P divides N, then each processor computes the nonbonded forces for N/P atoms. The uneven atom density results in varying values of $pCnt$; therefore, the inner loop with the (relatively expensive) force calculation often has to be executed with processors masked out even though they still have work to do in later iterations, just as it was the case in the `EXAMPLE` in Section 3. All processors have to go through

$$TIME_{SIMD} = \sum_{i=1}^{N/P} \max_{p=1\ldots P} pCnt(Atom_p^i) \qquad (2'')$$

iterations, where $Atom_p^i$ is the i-th Atom of processor p.

```
F = 0
At1 = [1 : P]
lastAt = [N−P+1 : N]
WHILE ANY (At1 ≤ lastAt)
  WHERE (At1 ≤ lastAt)
    DO pr = 1, max(pCnt(At1))
      WHERE (pr ≤ pCnt(At1))
        At2 = partners (At1, pr)
        F(At1) = F(At1) + Force (At1, At2)
      ENDWHERE
    ENDDO
    At1 = At1 + P
  ENDWHERE
ENDWHILE
```

Figure 14: F90$_{SIMD}$ version of `NBFORCE`.

```
F = 0
At1 = [1 : P]
lastAt = [N−P+1 : N]
pr = 1
WHILE ANY (At1 ≤ lastAt)
  WHERE (At1 ≤ lastAt)
    At2 = partners (At1, pr)
    F(At1) = F(At1) + Force (At1, At2)
    WHERE (pr = pCnt(At1))
      At1 = At1 + P
      pr = 1
    ELSEWHERE
      pr = pr + 1
    ENDWHERE
  ENDWHERE
ENDWHILE
```

Figure 15: Flattened F90$_{SIMD}$ version of `NBFORCE`. We take into account that $pCnt(i) \geq 1$ for all i.

This can be improved on by applying loop flattening, where we take into account that each atom has at least one interaction partner. The result is shown in Figure 15. Now each processor can loop through its atoms individually, so this code achieves the same time bound as a MIMD implementation:

$$TIME_{SIMD}^{flat} = \max_{p=1\ldots P} \sum_{i=1}^{N/P} pCnt(Atom_p^i), \qquad (1'')$$

which is only limited by the quality of our workload distribution.

5.2 The hardware used

We implemented the nonbonded force kernel taken from the GROMOS program suite on two SIMD machines and one workstation. Our implementation models the behavior of the actual GROMOS routine by reading in the arrays $pCnt$ and $partners$ as produced by GROMOS and then generating the calls to a force

routine for each interaction pair. To exclude communication time from our measurements, we assume that the *pCnt* and *partners* arrays and the molecular configuration data (including the coordinates of atoms we are interacting with) are already locally available when calling the force routines.

The **DECmpp 12000 model 8B** (Digital Equipment Corporation), which is identical to the MasPar MP-1200 series model, consists of 8192 processors (up to 16384 available), which are arranged in a mesh topology. It has 64 Kbytes main memory per processor, which gives 512 Mbytes total. Based on clock cycle counts, the individual processors are rated at 1.8 Mips. They are joined by an array control unit rated at 14 Mips. The MPFortran version we had on site (1.0) did not allow the use of indirect array addressing in FORALL statements, so the timing results presented here are achieved using an α-version of the 2.0 compiler at MasPar which does not have this restriction.

The **CM-2** (Thinking Machines Corporation) consists of 8192 one-bit processors (up to 65536 available), arranged in a hypercube topology. These are enhanced with 128 64-bit vector Floating Point Accelerators (FPA's) which use vector registers of length four. Each FPA is shared by two processor nodes of 32 processors each. The processors have 256 Kbits memory per processor, yielding a total of 268 Mbytes. The performance measured for a BYTE ADD is 500 Mips. We compiled our codes using the Slicewise 1.1 CMFortran compiler which lays out the data "slicewise" across the one-bit processors and uses the FPA's directly.

We also implemented the kernel on the **Sparc 2** (Sun Microsystems, Inc.), which is rated at 28 Mips and whose 16 Mbytes memory allowed us to run the smaller test cases. We compiled our program with the Sun f77 compiler.

One additional interesting machine parameter is the *data granularity* which measures how small an array can be if we want to distribute it across *all* processors. This granularity, *Gran*, is particularly important on SIMD machines since whenever a certain array has to be manipulated by some processors, all processors have to step through the operation and they will be merely masked out if they do not actually own part of the array. Furthermore, this potential waste of processing time can not only occur for small arrays, but it is encountered whenever array sizes are not exact multiples of *Gran* [15]. On the CM-2, using the slicewise compiler results in $Gran = P * 4/32 = P/8$ (32 processors per FPA, vector length 4); *i.e.*, we can economically use arrays whose total sizes are arbitrary multiples of $P/8$. This is a major advantage of the Slicewise model over the Paris model, which allocates data per one-bit processor. The corresponding data granularity on the DECmpp is simply $Gran = P$, and on the Sparc it is obviously $Gran = 1$.

Furthermore, the SIMD machines differ in the way they distribute data across the processors, which is significant if a dimension larger than *Gran* is distributed. The difference can be summarized as a cyclic ("cut-and-stack") data layout on the DECmpp and a block-wise layout on the CM-2.

5.3 Implementation experience

The DECmpp program and the CM-2 program used a single source, annotated with two sets of compiler directives, one for each machine. This worked relatively well; the only exception in our code was the `reshape` intrinsic. (The CMFortran convention for the argument order of this function is `mold` argument first, `source` argument second; MPFortran calls the `mold` argument `shape`, and has the order reversed. This combination of incompatibilities necessitated separate include files when using `reshape`; another option we tried was to replace the `reshape`'s with explicit `forall` statements, which caused a slight performance degradation on both machines.) The Sparc implementation shared the code for performing I/O and gathering timing statistics.

On the DECmpp, a compiler switch is used to recompile for different machine sizes. No compiler switch is needed for CM-2 since it uses a *virtual processor* model which adjusts automatically to the actual machine size. However, we can still obtain significant performance improvements if compile time constants are used to adjust array dimensions to actual machine configurations.

The indirect addressing used in the flattened loop version frequently required resorting to FORALL's in the source code. For example, the statement

```
forall(i=1:P) at2(i) = partners(i,l(i),pr(i))
```

cannot be expressed with indirection vectors as

```
at2 = partner(:,l,pr)
```

since this expression would yield a three-dimensional array with `at2(i,j,k) = partners(i,l(j),pr(k))` instead of the desired one-dimensional array computed in the `forall` statement. However, implementing the flattend $F90_{SIMD}$ version from Figure 15 was still relatively straightforward. The derived code, L_f, ran well on both machines without further tuning; it is shown in Figure 16. `Lrs` is the number of *memory layers* (or *virtual processor slices*) which are in actual use; it is `Lrs` $= \lfloor 1 + (N-1)/Gran \rfloor$. The dimensions indexed with `1:Lrs` are of size `maxLrs` $= \lfloor 1 + (N_{max} - 1)/P \rfloor$; for our implementation, the maximal number of atoms simulated is $N_{max} = 8192$. For example, for $Gran = 128$ and the $N = 6968$ atom test case described in Subsection 5.4, it is `Lrs` $= 55$ and `maxLrs` $= 64$; for $Gran = 8192$, we have `Lrs` $=$ `maxLrs` $= 1$.

```
      subroutine AllFFlat()
C     Formal parameters omitted here;
C     F, pCnt, partners are distributed
C     in first dimension

      integer at1(P),at2(P),l(P),pr(P),m(P)
      real Force(P)
cmf$  layout Force,at1,at2,l,pr,m
cmpf  ondpu Force,at1,at2,l,pr,m

      F = 0
      l = 1
      pr = 1
      at1 = [1:P]
      do while(any(l.le.Lrs))
        forall(i=1:P) at2(i) =
     .       partners(i,l(i),pr(i))
        call OneFFlat(Force, at1, at2)
        forall(i=1:P, l(i).le.Lrs)
     .       F(i,l(i)) = F(i,l(i)) + Force(i)
        forall(i=1:P) m(i) =
     .       (pCnt(i,l(i)).ge.pr(i))
        where (m)
          pr = pr + 1
        elsewhere
          pr = 1
          l = l + 1
          at1 = at1 + P
        endwhere
      enddo
```

Figure 16: CMFortran/MPFortran version of flattened `NBFORCE`.

```
      subroutine AllF()
C     Formal parameters omitted here;
C     F, pCnt, partners are distributed
C     in first dimension

      integer at1(P,maxLrs),at2(P,maxLrs)
      real Force(P,maxLrs)
cmf$  layout Force(:news,:serial)
cmf$  layout at1(:news,:serial)
cmf$  layout at2(:news,:serial)
cmpf  ondpu Force,at1,at2
cmpf  map Force(allbits,memory)
cmpf  map at1(allbits,memory)
cmpf  map at2(allbits,memory)
      integer pr

      F = 0
      at1 = reshape(shape = [P,maxLrs],
     .     source = [1:Nmax])
      maxPCnt = maxval(pCnt)
      do pr = 1, maxPCnt
        at2(:,1:Lrs) = partners(:,1:Lrs,pr)
        call OneF(Force,at1,at2)
        where (pCnt.ge.pr)
          F(:,1:Lrs) = F(:,1:Lrs) +
     .         Force(:,1:Lrs)
        endwhere
      enddo
```

Figure 17: CMFortran/MPFortran version of unflattened `NBFORCE`.

Our experience with the implementation of the unflattened loop version was very different. The initial implementation of the pseudocode in Figure 14 was trivial to write, but its performance was roughly an order of magnitude worse than the flattened version on both machines and required significant performance debugging. We tried several different implementations using interface blocks, layout directives, inlining, different compiler switches, etc.; parameter arrays were automatic, fixed size, or passed in `COMMON` blocks; the dimension corresponding to different atom numbers was either left as a single dimension (as in `Force(1:Nmax)`), or split up into physical processor number and memory layer (as in `Force(1:P,1:maxLayers)`); the `:serial`-ized dimensions were rightmost or leftmost (the latter version recommended by the CMFortran manuals); we tried DO-WHILE loops (as in `do while(any(pr.le.pCnt))`) and DO-ENDDO loops with precomputed loop bounds (`do pr = 1, maxPCnt`); we also tried vectorizing the code in the dimension indexed by `pr`, but this was unfeasible due to the size of `partners`.

We here present timing results for two different unflattened versions; the first version, L_u^1, is shown in Figure 17. The other version, L_u^2, differs from L_u^1 in that all explicit "`1:Lrs`" indices are replaced with just a ":" referring to the whole dimension. Note that the dimension indexed with `1:Lrs` is laid out serially into local memory. Theoretically the machine front end could take advantage of the explicit subscripts of the L_u^1 version by pruning the number of processed memory layers. However, in practice it turns out that, at least on the CM-2, the processors will always cycle through *all* layers of memory. Doubling N_{max} (and therefore doubling `maxLrs`) and leaving all other parameters fixed results therefore not only in doubling execution time of the L_u^2 version on both machines, but on the CM-2, it also doubles running time of the L_u^1 version; on the DECmpp, the L_u^1 time increases by about 5%. The running time of L_f is independent of N_{max} on both machines, which is a nice side effect of loop flattening. Therefore, using the L_u^1 loops does not automatically

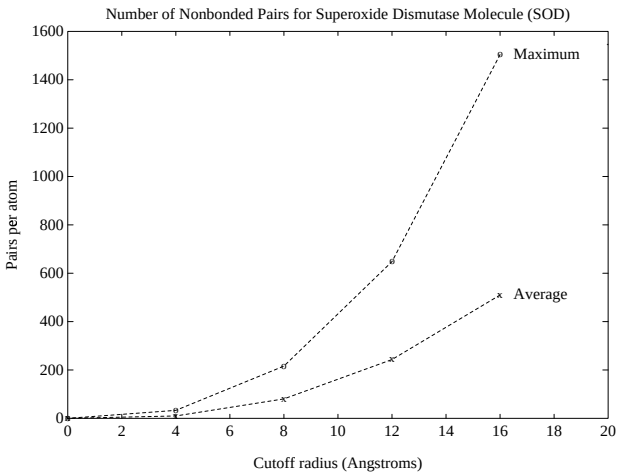

Figure 18: Maximum and average number of nonbonded force interaction partners per atom for the superoxide dismutase molecule, using different cutoff radii.

result in savings by reducing the number of processed layers; however, we have to pay the additional overhead of checking on each layer whether it is active [5]. This overhead is saved in the L_u^2 version.

5.4 The input data

We ran our test case for the bovine superoxide dismutase molecule (SOD), which has $N = 6968$ atoms. SOD is a catalytic enzyme composed of two identical subunits, each with 151 amino-acid residues and two metal atoms [20].

Figure 18 shows maximal and average numbers of interaction partners, $pCnt_{max}$ and $pCnt_{ave}$, which indicate the computational workloads for different cutoff radii. As expected, both values increase cubicly with the cutoff radius. As indicated in Equations $1''$ and $2''$, the difference between maximum and average number of partners gives us an upper bound on how much improvements we can expect from loop flattening.

5.5 The results

Table 1 gives performance results for the CM-2 and the DECmpp 12000, which are also displayed in Figure 19. For comparison, the running times on the Sparc were 3.86 seconds for the 4 Å case and 31.43 seconds for the 8 Å case. All runs were done several times, the differences in running times were usually less than 0.01%.

All loop versions were also timed with inlined calls to the force routine. On the CM-2, the effect was marginal; on the DECmpp, fluctuations were within 5%, roughly evenly distributed in both directions.

Table 2 gives the number of calls to Force routine for the flattened and the unflattened loop versions (the latter number scaled up by `Lrs` to account for the different argument sizes of `OneF()` and `OneFFlat()`) for different data granularities, along with their ratios. Note that the counts given in the last row are actually the maxima of $pCnt$ for the corresponding cutoff radii, as given in Figure 18. The given L_u/L_f ratios are bounded by the $pCnt_{max}/pCnt_{ave}$ ratios, which are 3.347, 2.689, 2.665, and 2.949 for cutoffs 4 Å, 8 Å, 12 Å, and 16 Å, respectively.

5.6 Interpretation

Considering the different computing powers per individual processor, the overall speedups of the parallel codes over the Sparc code version were satisfactory. However, we have to take into account that we excluded communication costs from our study. Due to the irregular nature of the problem, these communication costs might be relatively high; but as indicated earlier, the communication requirements are not changed by our transformation.

When comparing Tables 1 and 2, loop flattening fulfills the expectations given by Equations $1''$ and $2''$. Despite the significant effort on speeding up the unflattened loop versions (as described in Subsection 5.3), the improvements of the flattened version often went beyond what we predicted from the $pCnt_{max}/pCnt_{ave}$ ratios, in particular on the DECmpp. We assume that this is largely due to the side effect mentioned in Subsection 5.3, namely that loop flattening makes actual running times less dependent on array sizes if we do not access all parts of the array; *i.e.*, we can increase N_{max} without automatically making L_f slower (unlike for L_u^2 and even L_u^1). This we consider a significant advantage in practice, since it allows compiling programs with provision for maximal problem sizes without paying a penalty on smaller sizes.

Moreover, it turned out that the effort of expressing array bounds in terms of actual machine sizes improved the unflattened loop versions as well. This was particularly beneficial for the virtual processor model of the CM-2.

The differences between the two unflattened loop versions L_u^1 and L_u^2 were larger on the CM-2 than on the DECmpp, as mentioned in Subsection 5.3. However, even on the DECmpp L_u^2 performed better than L_u^1 when `Lrs` approached `maxLrs`.

It is important to keep in mind that the slicewise compiler for the CM-2 actually generates code with a data granularity of $Gran = P/8$, as discussed in Subsection 5.2. This coarser granularity results in more atoms per processor and therefore better applicability of loop flattening. As the table and the graph indicate for the CM-2, several cases could be run with the L_f

	4Å			8Å			12Å			16Å		
$P/Gran$	L_u^1	L_u^2	L_f	L_u^1	L_u^2	L_f	L_u^1	L_u^2	L_f	L_u^1	L_u^2	L_f
1024/128			3.89			27.03						
2048/256	6.57	3.86	2.13	42.91	25.13	14.72						
4096/512	3.22	1.83	1.11	21.02	11.95	7.65			24.78			
8192/1024	1.72	0.99	0.64	11.19	6.46	4.57			13.31			27.17
1024/1024	0.910	0.934	0.390	5.36	5.85	2.81	15.91	17.45	8.19	36.86	40.45	16.84
2048/2048	0.638	0.481	0.266	3.35	3.00	1.69	9.96	8.95	4.98	23.07	20.71	10.68
4096/4096	0.352	0.269	0.157	1.86	1.55	1.05	5.18	4.59	3.14	11.96	10.58	6.51
8192/8192	0.145	0.129	0.104	0.683	0.715	0.671	1.92	2.09	2.00	4.42	4.82	4.66

Table 1: Performance results for the CM-2 (upper half) and the DECmpp (lower half). Running times (in seconds) are listed for different cutoff radii and different loop versions. L_u^1: unflattened loop selecting memory layers, L_u^2: unflattened loop using all memory layers, L_f: flattened loop.

	4Å			8Å			12Å			16Å		
$Gran$	L_u	L_f	L_u/L_f	L_u	L_f	L_u/L_f	L_u	L_f	L_u/L_f	L_u	L_f	L_u/L_f
128		722			5076							
256	924	397	2.327	6048	2754	2.196						
512	462	224	2.063	3024	1559	1.940		4649				
1024	231	125	1.848	1512	906	1.669	4536	2642	1.717	10528	5436	1.937
2048	132	86	1.535	864	545	1.585	2592	1606	1.614	6016	3434	1.752
4096	66	51	1.210	432	357	1.210	1296	1069	1.212	3008	2222	1.354
8192	33	33	1	216	216	1	648	648	1	1504	1504	1

Table 2: Number of calls to Force routine, flattened/unflattened version. The data granularity, *Gran*, is equal to P for the DECmpp and $P/8$ for the CM-2. L_u counts are multiplied with `Lrs`.

version with reasonable performance while they could not be run at all in L_u^1 or L_u^2 because of stack overflows; large temporary arrays were needed in L_u^1 and L_u^2 even in loop versions which forward substituted intermediate results.

6 Loop Flattening from the Compiler's Perspective

The discussion so far seems to advocate a certain style of SIMD programming for applications which can benefit from loop flattening, just as a certain style of programming emerged when vector machines became popular. However, this would be contrary to existing efforts to make programming independent from machine idiosyncrasies, as for example the development of the Fortran D language. For non-SIMD machines, it still seems natural and efficient to have the inner loop bodies contained in the inner loops, even though flattened loops should run well on these machines also. Therefore, we suggest to make loop flattening part of the optimizing repertoire of SIMD compilers.

Applicability is ensured whenever there are multiple loops fully contained in each other, *i.e.*, there are not several loops on the same nesting level. This can be easily derived from the abstract syntax tree. Furthermore, the normalized version always tests the loop guard $test_l$ before executing $\mathcal{BODY}$, so we cover all loop constructs. The transformation itself is relatively straightforward; for example, there are no parameters to adjust, unlike in loop skewing. The first step of the transformation is to identify the three phases *init*, *test*, and *increment*.

WHILE/DO-WHILE loops: The relevant phases can be identified from their position between the WHILE/ENDWHILE keywords. Since $increment_2$ and $\mathcal{BODY}$ stay together throughout the transformation, we actually do not need to separate these two phases.

DO/FORALL loops: The phases can be derived directly from the loop header, as exemplified earlier.

GOTO loops: Similarly to WHILE loops, we can identify the phases by their position between labels and jumps.

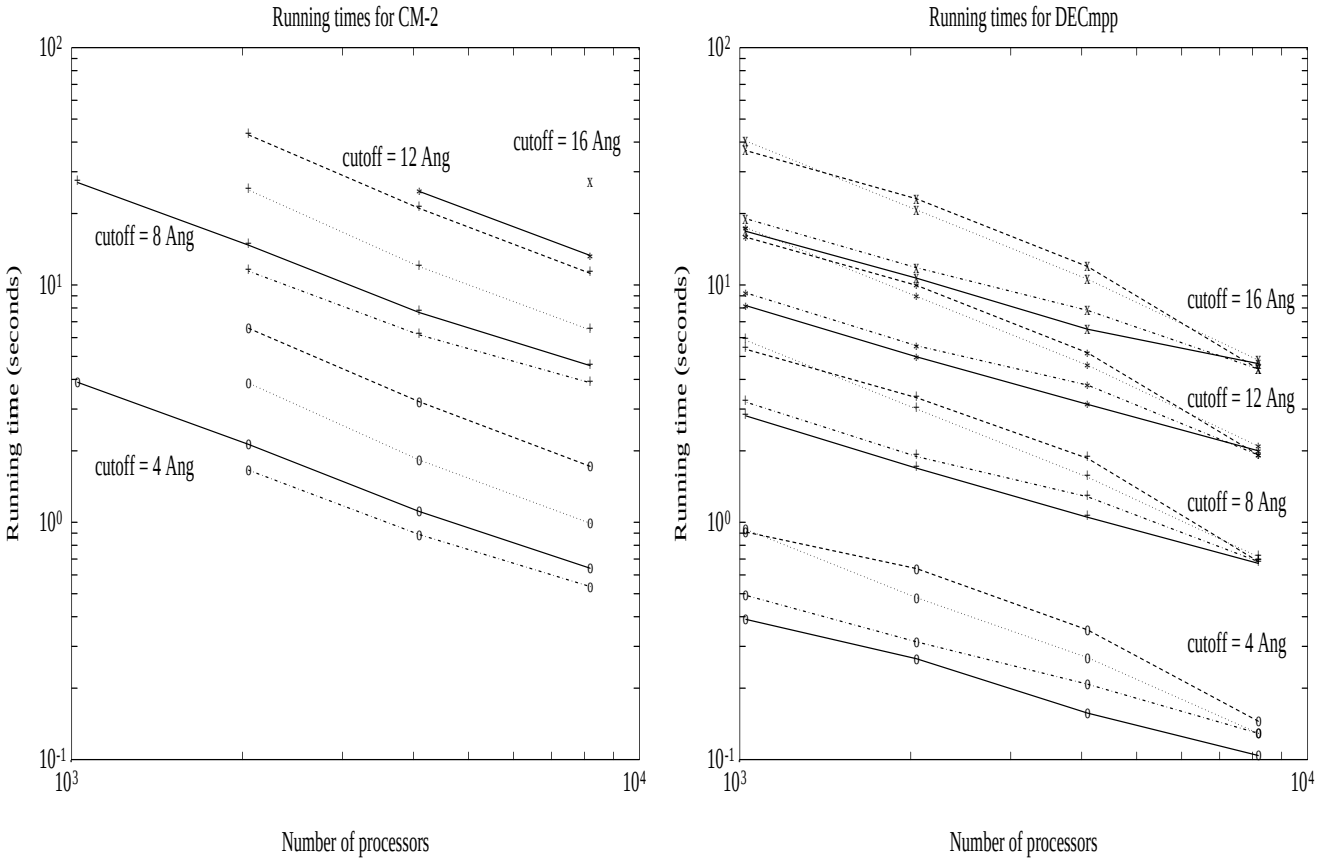

Figure 19: Performance results for the CM-2 and the DECmpp 12000. Different loop versions vary in line style; dashes: unflattened loop selecting memory layers; dots: unflattened loop using all memory layers; solid lines: flattened loop. Different cutoff radii are indicated by point styles; circles: 4 Å, plusses: 8 Å; stars: 12 Å; crosses: 16 Å. For judging speedups, note the log-log scale and the aspect ratio.

After normalization, the introduction of flags t_l and the actual code rearrangement follow straightforwardly. As described in Section 4, we also can often detect opportunities for further optimizations, for example when we are transforming simple DO/FORALL loops.

In evaluating *profitability*, we note that the additional overhead caused by loop flattening is, in the worst case, to manipulate two flags and to perform two conditional jumps. So we can relatively safely assume profitability whenever the inner loop bounds may vary across the processors.

As with many code transformations, the hardest problem in automating loop flattening is to determine its *safety*. A sufficient condition is that the loop into which we lift an inner loop body can be parallelized, which might be hard to detect, especially if indirect addressing occurs. However, this is already a necessary condition for parallelizing loops in general, and therewith a standard problem for parallelizing compilers [13]. The same technology developed there can be applied here.

When safety is ensured, either by user information (like a FORALL loop header) or by "heroic dependence analysis," we expect that the systematic loop flattening transformation, as described in Section 4, can be implemented efficiently into compilers like the Fortran D compiler in the ParaScope programming environment [14].

7 Related Work

The restricted control flow of pure SIMD programming has been addressed by several researchers. Philippsen and Tichy introduce two variants of a FORALL statement, a synchronous version and an asynchronous one [17]. The *asynchronous FORALL* allows multiple threads of control to coexist. This can either be emulated using stacks of MASK bits, or it can be implemented directly in an MSIMD machine which contains multiple program counters. In either case, their proposal is mainly concerned with allowing the concurrent execution of both branches in IF-THEN-ELSE

constructs; it does not directly apply to inner loops with varying bounds.

Loop flattening can also be used to process multiple array *segments* of different lengths per processor, as introduced in Blelloch's *V-RAM model* [3]. Thus it can be viewed as a generalization of substituting direct addressing with *indirect addressing* as Tomboulian and Pappas did for computing the Mandelbrot set [22].

Loop flattening bears similarities to *loop coalescing* in that it also manipulates loop control flow, but it is very different in its motivation and final outcome [18]. Loop coalescing merges iteration variables to achieve a higher degree of parallelism and to allow a more flexible distribution of inner loop iterations among the processors. Although loop flattening can also simplify load balancing, the transformation per se does not change *which* loop iterations a processor executes. Instead, it gives it more freedom as to *when* it executes them.

To conclude, the relative performance difference between conventional and flattened $F90_{SIMD}$ programs will depend on the variance of the cost of the inner loops for different outer loop iterations. We expect the difference to be significant in many cases, as it was for the application described in this paper. Furthermore, we believe that current compiler technology can automate this transformation effectively. If this were done, it would represent another significant step toward compiler support of architecture independent parallel programming for a large class of irregular scientific problems.

Acknowledgements

We are very grateful to Terry Clark and Ridgway Scott for providing detailed knowledge and experience about parallelizing molecular dynamics codes. Terry also supplied us with the pairlist data for the SOD molecule simulated in our experiments. Special thanks to Jim Armstrong from MasPar, who was an invaluable source of information and very helpful with the DECmpp implementation. Mark Mazina assisted with both the DECmpp and the CM-2 programming. We also wish to thank Chuck Koelbel and Jerry Roth for many profitable discussions about solving irregular problems on parallel computers and about SIMD related problems. Matthias Felleisen gave helpful comments on the submitted abstract.

References

[1] T. Bemmerl, A. Bode, O. Hansen, and T. Ludwig. A testbed for dynamic loadbalancing on distributed memory multiprocessors. PUMA Working Paper 14, Technical University Munich, München, Germany, August 1990.

[2] H. Berryman, J. Saltz, W. Gropp, and R. Mirchandaney. Krylov methods preconditioned with incompletely factored matrices on the CM-2. *Journal of Parallel and Distributed Computing*, 8:186–190, 1990.

[3] G. E. Blelloch. *Vector Models for Data-Parallel Computing.* The MIT Press, 1990.

[4] T. Bräunl. Structured SIMD programing in Parallaxis. *Structured Programming*, 10(3):121–132, 1989.

[5] P. Christy. Virtual processors considered harmful. In *Proceedings of the 6th Distributed Memory Computing Conference*, Portland, OR, April 1991.

[6] T. W. Clark, R. v. Hanxleden, K. Kennedy, C. Koelbel, and L. R. Scott. Evaluating parallel languages for molecular dynamics computations. In *Scalable High Performance Computing Conference*, Williamsburg, VA, April 1992.

[7] T. W. Clark, R. v. Hanxleden, J. A. McCammon, and L. R. Scott. Parallelization strategies for a molecular dynamics program. In *Intel Supercomputer University Partners Conference*, Timberline Lodge, Mt. Hood, OR, April 1992.

[8] G. C. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D language specification. Technical Report TR90-141, Dept. of Computer Science, Rice University, December 1990. Revised April, 1991.

[9] G. C. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker. *Solving Problems on Concurrent Multiprocessors.* Prentice-Hall, 1988.

[10] W. F. van Gunsteren and H. J. C. Berendsen. GROMOS: GROningen MOlecular Simulation software. Technical report, Laboratory of Physical Chemistry, University of Groningen, Nijenborgh, The Netherlands, 1988.

[11] R. v. Hanxleden and L. R. Scott. Load balancing on message passing architectures. *Journal of Parallel and Distributed Computing*, 13:312–324, 1991.

[12] *Proceedings of the High Performance Fortran Forum*, Houston, TX, January 1992.

[13] S. Hiranandani, K. Kennedy, and C. Tseng. Compiler support for machine-independent parallel programming in Fortran D. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines.* North-Holland, Amsterdam, The Netherlands, 1992.

[14] K. Kennedy, K. S. McKinley, and C. Tseng. Analysis and transformation in the ParaScope Editor. In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[15] K. Knobe, J. Lukas, and G. Steele, Jr. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *Journal of Parallel and Distributed Computing*, 8(2):102–118, February 1990.

[16] MasPar Computer Corporation, Sunnyvale, CA. *MasPar Fortran Reference Manual*, 1991.

[17] M. Philippsen and W. F. Tichy. Modula-2* and its compilation. In *First International Conference of the Austrian Center for Parallel Computation*, Salzburg, Austria, September 1991.

[18] C. D. Polychronopoulos. Loop coalescing: A compiler transformation for parallel machines. In S. Sahni, editor, *Proceedings of the 1987 International Conference on Parallel Processing*, St. Charles, IL, August 1987. Pennsylvania State University Press.

[19] J. Saltz, S. Petiton, H. Berryman, and A. Rifkin. Performance effects of irregular communication patterns on massively parallel multicomputers. ICASE Report 91-12, Institute for Computer Application in Science and Engineering, Hampton, VA, January 1991.

[20] J. Shen and J. A. McCammon. Molecular dynamics simulation of Superoxide interacting with Superoxide Dismutase.

Chemical Physics, 158:191–198, 1991.

[21] Thinking Machines Corporation, Cambridge, MA. *CM Fortran Reference Manual*, 1991.

[22] S. Tomboulian and M. Pappas. Indirect addressing and load balancing for faster solutions to the Mandelbrot set on SIMD architectures. In *Frontiers90: The 3rd Symposium on the Frontiers of Massively Parallel Computation*, pages 443–450, College Park, MD, October 1990.

[23] M. Willebeek-LeMair and A. P. Reeves. Solving nonuniform problems on SIMD computers: Case study on region growing. *Journal of Parallel and Distributed Computing*, 8:135–149, 1990.