Restructuring Fortran Programs for Cedar *

Rudolf Eigenmann, Jay Hoeflinger, Greg Jaxon, Zhiyuan Li, David Padua
Center for Supercomputing Research & Development
University of Illinois at Urbana-Champaign
Urbana, Illinois 61801

Abstract

This paper reports on the status of the Fortran translator for the Cedar computer at the end of March, 1991. A brief description of the CEDAR FORTRAN language is followed by a discussion of the FORTRAN77 to CEDAR FORTRAN parallelizer that describes the techniques currently being implemented. A collection of experiments illustrate the effectiveness of the current implementation, and point toward new approaches to be incorporated into the system in the near future.

1 Introduction

The University of Illinois has been a pioneer in the development of program translation techniques for vector and parallel computers since the late 1960s, when Illiac IV was developed. It is therefore natural that automatic parallelization has become one of the major concerns of the Cedar project, the latest machine building effort of the University of Illinois.

The Cedar machine is a hierarchical multi-processor. It supports several levels of parallelism and provides data storage at the processor, cluster, and system levels. The Cedar architecture links multiple clusters together with global memory modules. Each cluster contains multiple processors (linked by a concurrency control bus), local memory, and a shared data cache. Each processor contains a private instruction cache, scalar and vector registers, plus special instructions to support concurrent execution at the cluster level. Refer to Figure 1 for more detail about the architecture.

*This work was supported by the U.S. Department of Energy under Grant No. DOE DE-FG02-85ER25001. This paper is a modified version of a paper presented at the International Conference on Parallel Processing, held in St. Charles, Illinois, August 12 - 16, 1991.

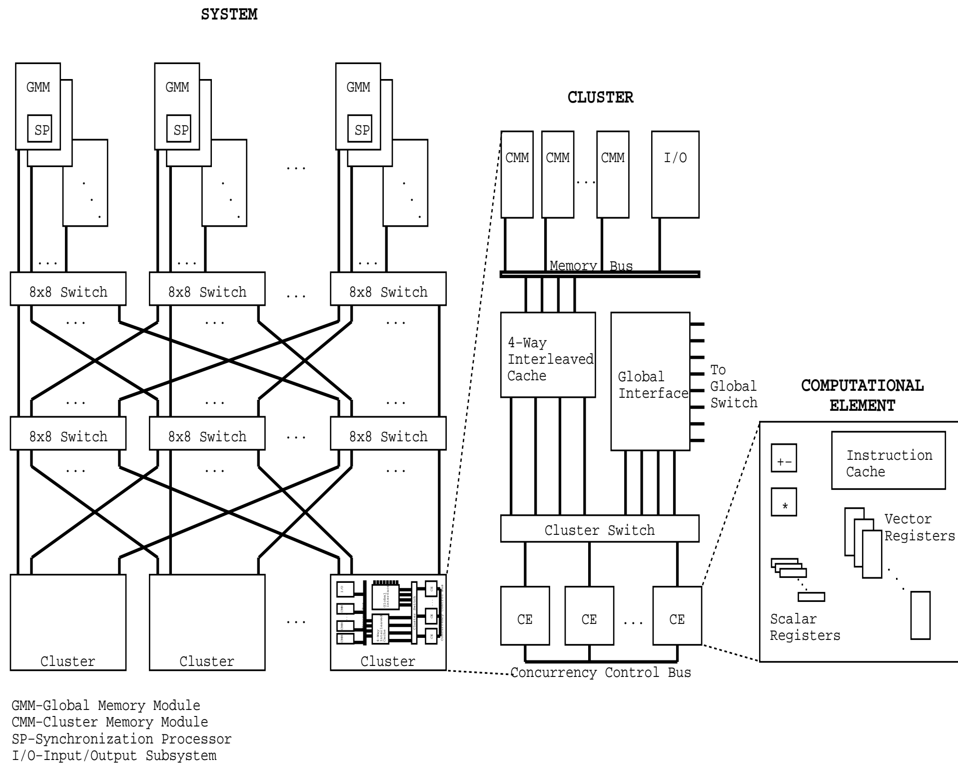

Figure 1: The Cedar Architecture

The Cedar machine existed in two, very similar configurations during the time when the experiments in this paper were run. Both consisted of four clusters of eight processors each, 512 kilobytes of shared data cache per cluster, and a two-stage interconnection network consisting of a forward and a backward path made of 8x4 and 4x8 crossbars. Configuration 1 had 64 megabytes of global memory and four clusters of 16 megabytes of local memory each. Configuration 2 used 64 megabytes of global memory and four clusters of 64 megabytes of local memory each.

The most frequent use of supercomputers today is in the execution of scientific programs that are dominated by numerical algorithms. Furthermore, parallel numerical algorithms have been studied extensively and are relatively well understood. For these reasons we have used, and plan to use in the near future, mostly numerical applications to study Cedar's behavior and performance. Because Fortran is the dominant language in numerical computing today, most of the compiler effort has been devoted to the design and implementation of Fortran translators.

Even though we emphasize numerical computing, it is not our only interest. We believe that Cedar is a general-purpose computer that should also perform effectively on non-numerical problems. Therefore, some effort has been devoted to the implementation and study of the behavior of parallel symbolic programs and more work in this area is planned. To support this effort we are developing parallelizing compilers for symbolic computing languages such as LISP [13] and PROLOG, as well as for C.

Our Fortran translation system, which is shown in Figure 2, consists of two components. The back-end compiler, a modified version of the Alliant Fortran compiler, generates machine code for Cedar from programs written in CEDAR FORTRAN, a parallel programming dialect described in Section 2. CEDAR FORTRAN gives the programmer access to the main architectural features of the machine, including all levels of parallel execution and memory hierarchy.

A programmer developing a new supercomputer application, who is concerned with performance and knowledgeable about parallelism and the target machine, could use CEDAR FORTRAN exclusively. In fact, many of the programs developed by the applications and algorithms researchers on the Cedar project have been written in CEDAR FORTRAN.

Some programmers, however, find it more desirable or even necessary to write in a conventional programming language such as FORTRAN77, because they are not interested in learning the machine details or investing the extra time required to develop a parallel program. Programmers also may want to use existing FORTRAN77 code (sometimes called "dusty decks") instead of writing a new program, or he or she may want to build a new program using a large fraction of existing sequential code. Another motivation for using FORTRAN77 is that parallel source programs are cumbersome to port, in part because of the lack of widely accepted standards. This is in contrast with FORTRAN77 programs, which can be ported relatively easily to most machines (especially if portability was taken into consideration when the program was written).

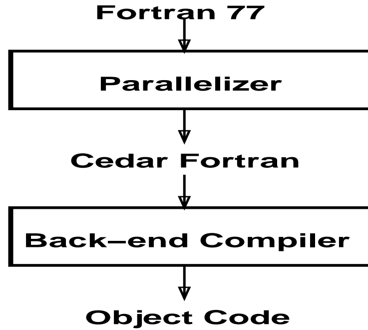

Figure 2: Fortran translation for Cedar

The parallelizer (also referred to in this text as the *restructurer*) in Figure 2 was developed for programmers who prefer or need to program Cedar using FORTRAN77. It is based on a 1988 version of the KAP restructurer, a product of Kuck and Associates (KAI). We modified KAP, as discussed in Section 3, to take into account those architectural characteristics of Cedar that distinguish it from other shared-memory multiprocessors. As discussed in Section 4, we are in the process of evaluating the present version of the parallelizer [2] and studying approaches to make it more effective. Some positive results have already been obtained, but we are still far from the goal of succeeding most of the time in automatically producing effective Cedar code from sequential FORTRAN77 programs.

2 Cedar Fortran

2.1 Language Description

CEDAR FORTRAN was designed with two purposes in mind: to be the output language for the Cedar restructurer and to be a programming language for expressing parallel programs. The result is a language with minor syntactic extensions to FORTRAN77, yet with the expressive power to make full use of the architectural features of the Cedar machine [12].

CEDAR FORTRAN has many features in common with the ANSI Technical Committee X3H5 standard for parallel Fortran (whose basis was PCF Fortran, developed by the Parallel Computing Forum), including parallel loops, loop-local data declarations, declarations for the visibility of data, and constructs to support post/wait synchronization.

Vector operations are provided in CEDAR FORTRAN. Assignment and all arithmetic operators work for vectors as well as for scalars. Some vector reduction intrinsics are also provided, such as `sum` and `dotproduct`. Also part of the language is the FORTRAN90 `WHERE` statement for masked vector assignments.

Two types of parallel loops exist in CEDAR FORTRAN: `DOACROSS` and `DOALL`. A `DOACROSS` loop is called an *ordered* parallel loop, because its iterations start in the same order as they would if the loop were sequential. This guarantees correct execution if *cascade synchronization* is used in the `DOACROSS` loop body. In cascade synchronization, signals are passed from earlier iterations to later ones, for the purpose of maintaining a sequential ordering of the execution of a particular portion of the loop body. An example of using cascade synchronization in a `DOACROSS` loop may be seen in Figure 4.

The `DOALL` loops are called *unordered* parallel loops, because no assumptions can be made about the order in which the iterations will be executed.

The syntactic form of all these loops is similar to that of a FORTRAN77 `DO` loop, but they have a few extra components as shown in Figure 3.

```
{ C }  { DOALL    }
{ S }  {          }   index = start, end [, incr]
{ X }  { DOACROSS }
            [local declarations]

            [ preamble ]
            [ LOOP     ]

            body

            [ ENDLOOP   ]
            [ postamble ]          (only SDO or XDO)

    { C }  { DOALL    }
END { S }  {          }
    { X }  { DOACROSS }
```

Figure 3: Concurrent loop syntax

There are three classes of loops in CEDAR FORTRAN: cluster loops using the prefix `C`, spread loops using the prefix `S`, and cross-cluster loops using the prefix `X`.

`CDOALL` and `CDOACROSS` loops cause all processors on a single cluster to join in the execution of the loop body. `SDOALL` loops cause a single processor on each Cedar cluster to begin execution of the loop body. `CDOALL` loops are often nested inside `SDOALL` loops to engage all processors on each cluster. An `XDOALL` loop causes all processors on all clusters to begin executing the loop body.

Data declared local to `CDO` and `XDO` loops (see Figure 3) is visible to only a single processor. Each processor has its own copy of this data. Data declared local to an `SDO` loop is visible to all processors of a single cluster. The *preamble* of the loop is executed once by each processor that joins the loop, prior to execution of

the loop body. The loop *postamble* is not currently available for CDO loops, but may be used for SDO and XDO loops. It is executed once by each processor after the processor finishes all its work on the loop.

```
CDOACROSS i=1,n

   c(i) = d(i) + e(i)
   g(i) = f(i) * h(i)

   call await(1,1)
   b(i) = a(i) + b(i-1)
   call advance(1)

END CDOACROSS
```

Figure 4: Cascade synchronization in DOACROSS

CEDAR FORTRAN provides statements for explicitly declaring data outside loops to be visible to

- all processors on all clusters, or
- all processors on a single cluster.

By default, data declared outside of loops in CEDAR FORTRAN programs is visible to all processors on a single cluster.

GLOBAL *var* [, *var*] ...
CLUSTER *var* [, *var*] ...
PROCESS COMMON / *name* / *var* [, *var*] ...
COMMON / *name* / *var* [, *var*] ...

Figure 5: CEDAR FORTRAN data declaration statements

The GLOBAL and PROCESS COMMON statements (see Figure 5) declare data that is visible to all processors on all clusters. A single copy of this data exists in global memory, and any processor with the address of one of these data items may access it.

The CLUSTER and COMMON statements declare data that is visible to all processors on a single cluster. A separate copy of this data exists in each cluster participating in the execution of the program.

The Cedar Fortran language is fully described in [16].

2.2 Implementing Cedar Fortran on Cedar

2.2.1 Parallel loops

The parallel loops of Cedar Fortran are all self-scheduled, by default, through a technique called microtasking [3]. `CDO` loops use microtasking supported by special concurrency hardware within the Alliant FX/8. This hardware is used for dispatching iterations of `CDO` loops and for synchronizing between iterations of `CDOACROSS` loops.

`SDOALL` and `XDOALL` loops use microtasking supported by the Cedar Fortran runtime library. The library starts a requested number of *helper tasks* ("implicit tasks" in IBM terminology [18]) which remain idle until an `SDOALL` or `XDOALL` loop starts. At that time, the helper tasks begin competing with the main task for iterations of the loop.

2.2.2 Tasking

Subroutine-level tasking is also supported by the Cedar Fortran runtime library. In subroutine-level tasking, a new execution thread is formed for running a subroutine. When the subroutine returns, the thread ends. Two ways of doing subroutine-level tasking are available: via a new cluster task built by the operating system at the time the thread is started (via a `ctskstart` call), or via an already-existing helper task (the thread is started via an `mtskstart` call).

The `ctskstart` mechanism involves much higher overhead, but it allows unrestricted forms of synchronization. On the other hand, synchronization instructions are not allowed in threads started with `mtskstart` because of the possibility of deadlock in our implementation of the microtasking approach. This deadlock potential arises from the fact that a helper task remains associated with a thread until it completes its execution. Because no context switching is allowed, when the number of helper tasks is smaller than the number of threads, waiting for threads that have not been scheduled on any helper task may produce deadlock. On the other hand, in the right situations, the `mtskstart` mechanism provides a low-overhead mechanism for subroutine-level tasking, making possible the use of a finer grain of parallelism.

2.2.3 Vector prefetch from global memory

The Cedar machine provides hardware to prefetch data from global memory. The back-end compiler generates instructions to trigger the prefetch mechanism prior to the vector fetching of global data. The data is prefetched into a special buffer attached to the processor that issued the prefetch request. Once data is in the prefetch buffer, it is available to the processor at cache speed. Ideally, the prefetch trigger instruction should be placed as early in the instruction stream as possible, so that the data is in the buffer when it is needed. The back-end compiler generates

a prefetch instruction for 32 elements before each vector register load instruction whose source is in global memory.

3 Automatic Parallelization

More than two decades of research in parallelization [21, 25] have produced many commercially available parallelizers. Some are embedded within machine-specific compilers whereas others, notably VAST from Pacific Sierra Research, and KAP from Kuck & Associates, Inc. [20], are machine independent and have been targeted for many different machines.

These parallelizers convert sequential programs into vector/concurrent code that in some cases runs significantly faster than the original version. However, there is still much room for improvement, and as a consequence there is a need for new analysis and restructuring techniques and for a more comprehensive evaluation of the capabilities and limitations of today's techniques.

New developments in computer architecture produce new problems in parallelization. Recent examples are multiple functional unit machines and distributed memory multiprocessor architectures. In the case of Cedar, the problem is to generate code for a hierarchical memory multiprocessor, where the processors are organized into clusters and there are three levels of parallelism (across clusters, inside clusters, and the vector pipelined parallelism of the processors).

Particularly important when generating code for Cedar is the existence of cluster memory which can be used in addition to the global memory to achieve good performance. Data references which are redirected from global memory to cluster memory make use of the cluster data cache, plus reduce global memory bank contention. Aggressive data privatization can be used to store data in cluster memory. Domain decomposition, whether applied automatically by the compiler or through user directives, also holds promise as an effective way to make use of cluster memory. Domain decomposition involves partitioning data structures and storing the parts in local memory modules accessible by only a subset of the processors. Domain decomposition techniques can also be used when compiling for distributed-memory computers.

Thanks to the global memory, domain decomposition is not as critical in Cedar as it is in distributed memory machines and, in at least some cases, simple data allocation strategies (e.g. the privatization algorithms discussed below) are sufficient to produce effective parallelization for Cedar.

3.1 The Cedar Fortran restructurer

CEDAR FORTRAN's restructurer is built on KAP. It accepts FORTRAN77 extended with a subset of the vector operations in the FORTRAN90 standard. Our modified version also accepts declarations indicating whether a variable or array is to be stored in global or cluster memory, and whether a `COMMON` block is to be visible

to all cluster tasks or to only one of them. The restructurer produces CEDAR FORTRAN source code as output.

The following subsections present the restructuring techniques we incorporated into KAP. Besides implementing these techniques, we further modified KAP by rewriting or extending several transformations including scalar expansion, stripmining, `DOACROSS` synchronization, `IF` to `WHERE` conversion, recurrence and reduction recognition, inline subroutine expansion, floating of loop bound-related calculations, and last-value assignments.

3.2 Stripmining, Globalizing, and Privatizing

The general restructuring scheme is illustrated by the following simple loop:

```
DO i=1,n
   a(i) = b(i)
END DO
```

After detecting that the loop iterations are independent, the restructurer generates a parallel version that can be executed on all processors in Cedar. In order to exploit all levels of parallelism in Cedar, the iteration space may be *stripmined* [22, 25] so that in each iteration a separate *strip* of data is processed in vector form. The loop can be restructured into the following form:

```
GLOBAL a,b,strip,n
XDOALL i=1,n,strip
   a(i:MIN(i+strip-1,n))= b(i:MIN(i+strip-1,n))
END XDOALL
```

For a given loop, the optimal strip length depends on the total number of iterations and the number of processors that participate. When these quantities are not known at compile time, we use default values. In general, `XDOALL` and stripmining are used when only one loop in a nest is parallelized. When several nested loops are parallelized, the outermost loop is transformed into a `SDOALL`, and the second to the outermost is transformed into a `CDOALL` loop. If there are only two nested parallel loops, the innermost is also stripmined to generate vector statements.

In the translated version above, the `GLOBAL` declaration makes `a,b,strip` and `n` visible to all processors on all clusters. This statement is generated by the *globalization* pass, which identifies the variables used in parallel loops involving processors from different clusters and then marks them as `GLOBAL`. Any variable used by the processors in a single cluster is marked as `CLUSTER` by the globalization pass.

The *privatization* pass looks for scalar variables whose value does not cross iteration boundaries, and marks them as local to the loop. It is worth noticing that some of the storage introduced by the compiler such as the bounds computed for

the inner loop after stripmining can be kept private. An example of privatization is shown next.

```
DO i=1,n
   t = b(i)
   a(i) = sqrt(t)
END DO

      ⇓

GLOBAL a, b, strip, n
XDOALL i=1,n,strip
   INTEGER upper, i3
   REAL t(strip)
   i3 = MIN(strip,n-i+1)
   upper = i + i3 - 1
   t(1:i3) = b(i:upper)
   a(i:upper) = sqrt(t(1:i3))
END XDOALL
```

Here, `upper`, `i3`, and `t` are declared to be *private* to the processors executing the parallel loop through the declaration statements within the loop.

Privatization is related to *scalar expansion* [28] which expands a scalar into an array if all references to the scalar in iteration i of the loop can be replaced by references to the ith element of the array. Privatizing expands the storage for the scalar to one cell per processor. The restructurer searches for the privatizable usage pattern at every level in a loop nest. It creates temporary storage using a combination of privatization and scalar expansion. In vector loops, scalar expansion is applied, while privatization is applied in concurrent loops. Sometimes the array bounds of expanded scalars need to be computed at runtime.

Both the globalization and privatization passes cooperate with the code that judges the best execution mode for each loop. However, an inherent difficulty of statically deciding the placement of a data item is that the decision affects the execution time of all parts of the program where the data item is used. Placing an array in global memory may benefit some parallel loops, but slow down some serial loops that cannot take advantage of the vector prefetch facilities. In most cases, these costs and benefits cannot even be calculated at compile time, yet placement must be done for every data item. Data placement choices are also complicated by `EQUIVALENCE` and `COMMON` block relations between variables.

Often the placement analysis must span procedure boundaries. The Cedar restructurer provides *inline expansion* of subroutine calls as an option to reduce the number of routine boundaries and meet some interprocedural analysis needs.

To simplify the static placement problem, there is a user-settable (global or cluster) default allocation for all data whose usage may cross a routine boundary,

which we call *interface data*; this includes `COMMON` blocks and all formal and actual parameters in subroutine and function calls. Where no single choice is satisfactory, the programmer can force the placement of particular variables using the `GLOBAL` or `CLUSTER` declarations mentioned above.

3.3 Reductions, Recurrences, and Synchronization

The Cedar restructurer recognizes loops that can be parallel if the order of their arithmetic or logical operations is allowed to change. Loops such as dot products, linear recurrences (e.g., `X(i) = X(i-1)*B(i) + C(i)`) and minimum/maximum searches are replaced by calls into a library of Cedar-optimized functions. For example, a dot product can be distributed to all Cedar processors, its partial results being summed up in two steps: within each cluster, then across the clusters. When a parallel `dotproduct` routine was used in the Conjugate Gradient algorithm [23], it cut the execution time of the whole program in half compared to the version of the program that used `dotproduct` vectorized on one processor only.

To make use of a library routine, the restructurer must often distribute an original loop to isolate those computations done by library code, which adds loop control overhead, reduces the average grain size of parallel activity, and reduces the effectiveness of the machine's registers. The payoff comes from the wealth of algebraic and programming insight that library authors use to reduce operation counts and memory references [5, 8].

Loops where different iterations may use the same storage cell can usually be concurrentized as `DOACROSS` loops. Uses of the shared location(s) are serialized by the `await` and `advance` functions in the concurrency control hardware, while the rest of the loop executes in parallel. The Cedar restructurer inserts the smallest set of synchronization instructions that will suffice [24].

When considering a `DOACROSS` loop version, the restructurer lowers its estimate of the benefit owing to parallel execution by a *synchronization delay* factor. Intuitively this is the size of the synchronized region (as a fraction of one iteration) divided by the number of processors that may be executing it concurrently.

3.4 Optimization Alternatives

Once parallelism has been recognized, there are still many ways that concurrent activity can be scheduled. Cedar's cluster architecture makes interprocessor communication cost less *within* a cluster than between clusters. For some loops it is not certain that a `DOALL` form could activate other clusters quickly enough to be of benefit. In others, the compiler must guess whether a `DOACROSS` could pass a synchronization signal through 8 or 32 processors fast enough to outperform the same loop distributed into serial `DO`s and parallel `DOALL`s. An understanding of the interaction of Cedar's many components and the overhead costs involved is still taking shape.

To find the right match between loop levels and hardware levels, the restructurer considers a whole loop nest at one time. A central coordinator tries out many potential transformations such as how loops in a nest might be interchanged, parallelized, or stripmined, and which data must then be placed in global memory. The many sources of parallelism and synchronization in Cedar can make the number of alternatives to consider become quite large.

Currently, the restructurer uses simple heuristics to identify transformed program versions worth further consideration. A user-settable hard limit (50 by default) keeps the number of candidate versions manageable. We believe that as the number of alternatives increases, so does the number of near-optimal ones; this should allow us to keep the heuristics simple and still be confident of finding a good translation of a loop.

4 Experiments

In this section we discuss some of the experience we have accumulated in the automatic parallelization of the Perfect Benchmarks® programs [26] and some linear algebra routines from *Numerical Recipes* [27]. The work reported below is part of an ongoing study whose goal is to learn about the limitations of the current version of the parallelizer as well as to develop new automatic techniques that, once incorporated in the parallelizer, would overcome these limitations.

This section is divided into two parts. In the first part we study the general ability of the restructurer to detect parallelism. This is a summary of work we have reported in [7], plus some new results. The transformations in this part are suitable for any parallel machine. In the second part we address some issues specific to compiling for the Cedar architecture.

4.1 Parallelism Detection

After the preliminary version of the parallelizer as described in Section 3 was completed, we started to study its effectiveness on small routines and synthetic loops. The initial results were encouraging. Table 1 shows the speedup results for a set of linear algebra routines.

The first routine is a *conjugate gradient* algorithm [23]; the other routines are from *Numerical Recipes* [27]. The data size shown in the second column in most cases represents the number of rows and columns of the input matrices. The speedup values refer to the increase in speed of the parallelized version run on Cedar versus the serial (scalar) form. In many cases satisfactory speedups are achieved. In fact, in all but two of the routines the compiler was able to parallelize all major loops.

The size of the input data set has a great influence on performance and speedup, because, as the amount of computation grows, it overcomes the negative effect due to the parallel loop overhead and due to the fetching of data from global memory.

Some of the routines exhibit very good speedups with relatively small data sets. Other routines start low, and their speedup is still improving when the size reaches 1000.

One particularly interesting case is that of routine *mprove*, which has a sharp increase in speedup when the size reaches 1000. The reason for this increase is the paging behavior of the serial version. The data for the serial version is all stored in the memory of a single cluster. For sizes greater than 800, the amount of data needed in the serial version exceeds the the size of physical memory, causing thrashing, whereas the data of the parallel version fits in the larger global memory. Similar effects contribute to the high speedups of the CG algorithm.

Routine	Data size	Speedup
CG	400	163
ludcmp	1000	9.2
lubksb	1000	6.8
sparse	800	29
gaussj	600	10
svbksb	200	32
svdcmp	200	7.2
mprove	1000	1079
toeplz	800	1.3
tridag	800	2.1

Table 1: Speedups of automatically restructured linear algebra routines on Configuration 1 of the 32-processor Cedar

As part of this study, we also ran several of the programs in the Perfect Benchmarks suite, which are complete applications ranging in size from a few hundred to a few thousand lines of code. The results obtained were less satisfactory, as shown in Table 2, where the speedups obtained on both Cedar and the Alliant FX/80 are presented. It must be noted that the "Automatically Compiled" results listed for Cedar in the table were run on Configuration 1 of the Cedar machine (as mentioned in the Introduction), while the "Manually improved" results for Cedar were run on Configuration 2 of Cedar. This difference in configuration should have had very little effect, limited to possibly reducing the speedup for the "Manually improved" programs by reducing page faults for the sequential version of the programs.

As can be seen from Table 2, in several cases practically no speedup was obtained automatically. Our experience with these codes corresponds to that reported by many computer vendors, who have obtained for these programs a performance far below their machine's theoretical peak. Analyzing the restructured codes by hand, we have found that many of the difficulties result from the general weakness of the existing restructuring technology and not from the target architecture or the algorithms used.

In our hand analysis [7], we examined the loops of the restructured program. If

program	Automatically compiled		Manually improved		$\frac{Manual\ speedup}{Automatic\ speedup}$	
	FX/80	Cedar	FX/80	Cedar	FX/80	Cedar
ARC2D	8.7	13.5	10.6	20.8	1.2	1.5
FLO52	9.0	5.5	14.6	15.3	1.6	2.8
BDNA	1.9	1.8	5.6	8.5	2.9	4.7
DYFESM	3.9	2.2	10.3	11.4	2.6	5.2
ADM	1.2	0.6	7.1	10.1	5.9	16.8
MDG	1.0	1.0	7.3	20.6	7.3	20.6
MG3D	1.5	0.9	13.3	48.8	8.9	54.2
OCEAN	1.4	0.7	8.9	16.7	6.4	23.9
TRACK	1.0	0.4	4.0	5.2	4.0	13.0
TRFD	2.2	0.8	16.0	43.2	7.3	54.0
QCD	1.1	0.5	2.0	1.8[1]	1.8	3.8
SPEC77	2.4	2.4	10.2	15.7	4.3	6.5
			Average manual improvement:		4.5	17.2

Table 2: Speedups versus serial for Perfect Benchmarks programs on Alliant FX/80 and Cedar

a loop was not parallelized by the restructurer, we studied the reason. If the problem resulted from limitations of the parallelizer, we tried to use more aggressive strategies and hand-parallelize the loop when possible. Throughout this process, we limited ourselves to automatable analyses and transformations rather than pursuing a complete analysis of the application problems and their numerical solutions. We restricted our work to automatable techniques because our goal is to improve the parallelizer. Those automatable techniques that we found successful will be incorporated into later versions of the parallelizer.

The methodology we used, in general, was to present the original serial programs to the restructurer, then hand-modify the resulting parallelized form. We built many tools to assist us in this very time-consuming and tedious work. Some of them are described in [6, 14].

Preliminary results from our experiment are encouraging. For the twelve programs we show in Table 2, by calculating the ratio of the speedup for the manually transformed code to the speedup for the automatically transformed version and averaging, the codes targetted at the Alliant FX/80 perform an average of 4.5 times

[1] A random number generator produces a dependence cycle in QCD which serializes half of the computation. The speedup value from the table (1.8) is the result when both halves of the cycle are serialized. If only the lexically forward dependence is serialized with a critical section, then a speedup of 4.5 is obtained. If the dependence is not serialized at all, (for instance, if the random number is replaced with a parallel random number generator), then a speedup of 20.8 is obtained. Only when the cycle is completely serialized does the code pass the Perfect Benchmarks validation test.

better than the automatically restructured codes. Applying the same calculation to the codes targetted for Cedar, the manually-transformed codes perform an average of more than 17 times better than the automatically restructured codes. When reading Table 2, keep in mind that these are speedups of the vector-concurrentized code versus the serial/scalar code, therefore speedup numbers greater than the number of processors involved are possible.

The rest of this section discusses some of the techniques we applied by hand to obtain the improved performance shown in Table 2. We believe that most of these techniques can be automated.

4.1.1 Compiling in the presence of interprocedural information

Our compiler currently relies on *inlining* [17] for interprocedural analysis. Inlining replaces call statements with the text of the called subroutine. However, in many of the Perfect programs, inlining fails. Sometimes subroutine calls are so deeply nested that inlining causes the compiler to run out of memory for its data structures. In other cases, array reshaping across subroutine boundaries causes the subscripts of the arrays in a loop to become too complex for the dependence analyzer to analyze.

For our hand-analysis, we quite often did the analysis *in the presence of* interprocedural information. This means that whenever we needed information that could not be found in the subroutine itself, we crossed procedure boundaries to get it, keeping in mind the control flow of the program. Sometimes, our techniques required that constants or relations between variables be propagated interprocedurally. Rather than attempt to propagate all constants, and all possible relations in a separate pass, we would proceed with a transformation technique until some constant or relation was needed, then do the propagation for just the object needed. This was particularly important for relations, for it allowed us to put the relations in precisely the form in which they were needed.

Interprocedural summary information was also very useful. It involved simply keeping track of which interface variables were used and defined by a particular routine and all of the routines which it called. This helped us to focus on the dependences within a subroutine which prevented it from being called from a DOALL loop, and to find techniques to deal with them.

4.1.2 Array privatization

One of the most important techniques which we employed by hand was *array privatization*. This transformation is closely related to scalar privatization, discussed in Section 3.2. The pattern of definition and use for a privatizable array is the same as it is for a privatizable scalar. Any element used must have first been defined.

The privatization of arrays is important for two reasons. First, it enables parallelization by removing dependences from the loop. Second, it allows the data in the array to reside local to the processor executing a particular iteration of the loop, reducing memory latency. Both of these effects are useful for all parallel machines.

We encountered many privatizable arrays in the Perfect codes. Most were very easy to recognize. Some were more difficult, requiring the propagation of relational information, sometimes across procedure boundaries. Array privatization was important for **all** of the Perfect programs.

4.1.3 Parallel reductions

Statements of the type `sum = sum + a(i)` form a cycle in the data-dependence graph, which usually serializes the loop. The fact that the sum operation is commutative permits a parallel execution that accumulates partial sums on each processor. The partial sums can be accumulated after the loop or added inside the loop in a (possibly unordered) critical section.

Quite often in the Perfect Benchmark codes we found loops which contained multiple accumulation statements, e.g.

```
       DO 100 i=1,n
          DO 100 j=1,m
                . . .
             a(j) = a(j) + <expression1>
                . . .
             a(j) = a(j) + <expression2>
                . . .
             a(j) = a(j) + <expression3>
                . . .
100    CONTINUE
```

While our restructurer could handle forms like `sum = sum + a(i)`, it was not prepared for multiple accumulation statements, nor for accumulation locations which were array elements.

The parallel reduction transformation turned out to be important for the routines BDNA, DYFESM, MDG, MG3D, and SPEC77. In MDG, very little speedup is possible without it.

4.1.4 Generalized induction variables

In Fortran DO loops, array subscripts often use the values of induction variables [1] which are updated in each iteration in the form of $V = f(V, K)$, where the values produced by f are monotonically increasing (e.g. $V = V + 1$). Such a recursive assignment causes cross-iteration flow dependences. If a compiler can solve such a recursion and rewrite each induction variable in terms of the loop indices, for example, $V = g(A, I, B, J)$, where I and J are loop indices and A and B are loop invariants, then the appearance of V in array subscripts can be replaced by the expression $g(A, I, B, J)$. The recursive assignment (and the dependence) can be eliminated as a result. There are well-known compiler techniques for recognizing and replacing an induction variable whose values form an arithmetic progression.

These techniques typically deal with induction variables assigned in the form of V = V + K.

In our experiment with the Perfect codes, we found induction variables whose values do not constitute arithmetic progressions. Here we call them *generalized induction variables* or GIVs. We found two types of GIVs. The first type is updated using multiplication instead of addition, thus forming a geometric progression. The second type is updated using addition, but forms no arithmetic progression nonetheless because the loops are *triangular*, that is, an inner loop limit depends on the value of an outer loop index. For both types of induction variables that we found in the Perfect code, we were able to determine the closed form expression for the value of the GIV. In the program OCEAN, one loop could be parallelized and sped up by a factor 15.8, thanks in part to the recognition of the multiplicative GIVs (that loop takes 46% of the serial execution time of the program). In the program TRFD, we found generalized induction variables of the second type.

4.1.5 Run-time dependence test

When the subscript expressions within loops contain variable coefficients, or the loop bound expressions contain variables, or both, traditional dependence tests have difficulty determining independence and therefore, in most cases conservatively assume that a dependence exists.

In the cases in which it is not possible to symbolically eliminate the variables from the subscript expressions and the loop bounds, the existence of a dependence cannot be known until run-time, when the values of the variables are fixed. All a compiler could do in such cases is to insert a dependence test which executes prior to the loop itself, using the values of the variables.

In OCEAN, 65% of the serial execution time of the program is spent in loops which contain potential dependences due to complex indexing expressions for singly-dimensioned arrays. The arrays are used inside multiply-nested loops. Both the indexing expressions and the loop bound expressions contain variables. If these loops are not parallelized, OCEAN is limited to an extremely small speedup.

To overcome this difficulty, we developed a dependence test which involves a test at run-time which chooses between a parallel and a sequential version of a loop. The test determines whether the indicated array is being indexed in a way which makes it a linearized version of a multi-dimensional array [15].

With this technique applied, all of these OCEAN loops with the complex indexing patterns turned out to be perfectly parallel in all cases, the aggregate speedup for them being 15.7 over the serial version.

4.1.6 DOACROSS loops and critical sections

Techniques for executing parallel loops with cross-iteration dependences have been known for many years. The Cedar restructurer can generate `await` and `advance` synchronization instructions to preserve cross-iteration dependences of simple types,

thus allowing `DOACROSS` parallel loop execution. In the TRACK program, we faced a variety of problems which required that we execute `DOACROSS` loops efficiently. For instance, in one loop, we found it more efficient to perform `await` synchronization conditional upon a runtime test.

CEDAR FORTRAN also provides locking and unlocking to protect *unordered* critical sections. Little has been published in the literature about compiler recognition and protection of unordered critical sections. However, in at least two programs (TRACK, and MDG) we parallelized the most time-consuming loops using unordered critical sections.

4.2 Important optimization techniques for Cedar programs

In this section we discuss some compiler issues that are specific to the Cedar machine. Several optimization techniques are discussed, and their effects on a few programs are presented. Some of these techniques have already been implemented; while others are being studied for future inclusion in the parallelizer.

4.2.1 Prefetching data from global memory

The memory hierarchy is one of the most significant characteristics of the Cedar architecture. In the presence of such a hierarchy it is particularly important to store and fetch data in such a way that keeps memory access cost low. This holds in particular for referencing data from the Cedar global memory.

A straightforward approach to reducing global memory access costs is to combine data requests and issue them as a block transfer to take advantage of the prefetch facility of Cedar. In CEDAR FORTRAN a natural program entity that refers to a block of data is the vector operation. Section 2.2.3 described how the compiler inserts prefetch instructions for vector operations. Figure 6 shows the effects of this optimization in two programs, the *Conjugate Gradient(CG)* Algorithm [23] and the Perfect code TRFD.

Although there is an improvement of up to 100% in CG, TRFD exhibits only a 15% gain, primarily because vector lengths are large in CG and small in TRFD. In addition, the manually optimized version of TRFD has a high percentage of its references privatized (diverted to cluster memory), while CG does not, further explaining the difference in improvement between the two programs.

There are many additional issues related to prefetching that we plan to study in the near future. For example, what is the effect of an aggressive floating of prefetching instructions [10]? The strategy used today in the CEDAR FORTRAN compiler is to generate prefetch code to precede each vector register load from global memory without any code motion optimizations.

4.2.2 Data privatization

Prefetching data reduces the latency for reading global data, but the latency still exists. Another source of performance loss stems from contention in the shared

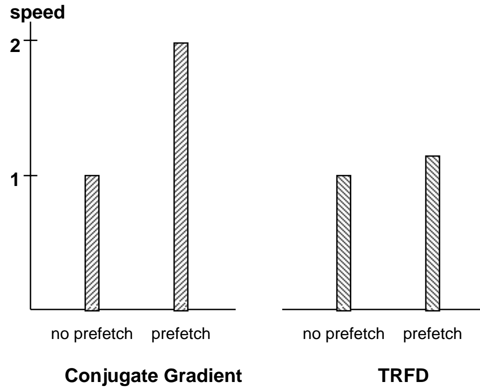

Figure 6: The effect of compiler-inserted prefetch instructions

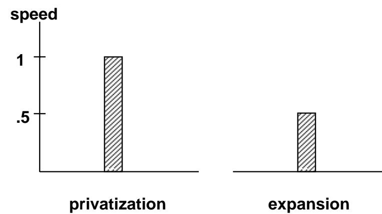

Figure 7: Data privatization vs expansion in MDG

global memory. An important architectural idea in the Cedar project is to overcome these problems by providing a local cluster memory, which grants faster and less contended access to data that need be seen by the local cluster only. In addition, cluster data references can benefit from the cache. A major challenge for the compiler is to find data that can be placed at this level of the memory hierarchy.

In Section 3.2 we described the compilation scheme for finding data that can be *privatized* to a given loop. All privatized data gets placed in cluster memory. We have found important code sections in the Perfect Benchmarks where this transformation improves performance. For example, in Figure 7 two variants of the major loop in the program MDG are measured. The first variant has privatized array data. In the second variant the same data elements were expanded and put in global memory. The figure shows a 50% slow down of the non-privatized version. The performance loss is not only attributable to the memory placement of the data, but also to the more costly addressing mode of the data which are now expanded by one array dimension. Although the measurement does not discriminate these two sources of performance loss, it clearly demonstrates the execution speed advantage of the privatization transformation.

4.2.3 Data partitioning and distribution

As determined in the previous section, data can be privatized when its life is confined to a loop iteration. When the lifetime spans several loops, one can attempt to place data partitions onto each cluster memory and assign corresponding subsets of the loop iteration spaces to the cluster processors. This works without further communication for data that is read-only or that is read by the same cluster on which it was written. Figure 8 shows the performance of the Conjugate Gradient algorithm before and after we have applied such a simple data partitioning and privatization strategy.

The figure shows the speed of the CG relative to a program variant that was optimized for a 1-cluster execution and which has its data in cluster memory. The solid curve corresponds to the automatically compiled algorithm, where most data is placed in global memory. On one cluster this causes a factor of 1.6 performance gain because of the high transfer rate of global memory and prefetch. On two clusters the performance is nearly twice the one-cluster performance; however, on 3 and 4 clusters the speed improves less and less. We attribute this effect to the program accessing global data near the maximum transfer rate of the global memory system. The dashed curve represents the data-partitioned implementation variant. This variant has 50% of its data references localized to the cluster memory. On one cluster the speed is less than the global-data version, but then it achieves a near-linear speedup through four clusters.

We intend to implement some data partitioning scheme in our compiler. This area of research is not mature and the practical value of proposed techniques is yet unclear [4, 9, 29, 11]. More experiments are needed. We have found that the Cedar architecture is a useful testbed for this purpose. It allows us to combine shared-memory and distributed-memory programming schemes. It lets us take advantage of newly explored data placement strategies while retaining in shared memory data whose distribution would cause intolerable communication overhead.

4.2.4 Making large concurrent loops

Within a Cedar cluster, hardware is available to quickly start, end, and synchronize parallel loops, whereas the global memory is the only mediator for the inter-cluster communication needed for cross-cluster parallel loops, and the overhead for it is large. This raises the issue of providing the appropriate large grain parallelism at the program level. In loop-oriented programs, large granularity means loops with a high number of iterations and a large loop body.

Large loop iteration counts can often be obtained with large input data sets. Although current "real program" benchmark suites, such as the Perfect codes exhibit relatively small iteration counts for many of their crucial loops, we have hopes that larger data sets will make it possible to obtain larger speedups. We have shown above that linear algebra routines working on matrices of size 1000 by 1000 run quite efficiently on Cedar.

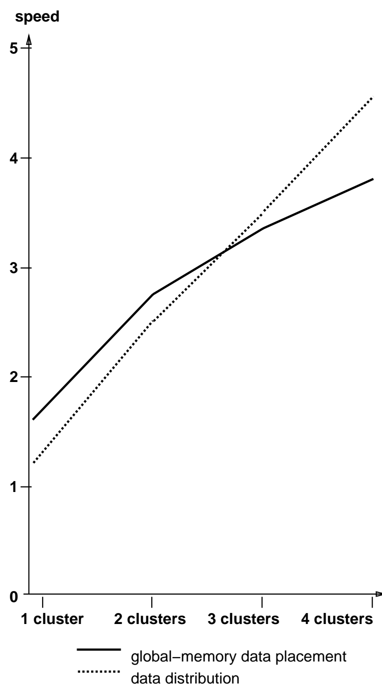

Figure 8: Data partitioning in the Conjugate Gradient Algorithm

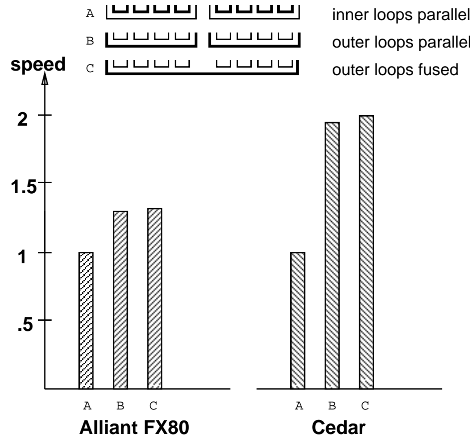

Figure 9: combining multiple parallel loops into a single parallel loop

The issue of finding large bodies for concurrent loops is a challenge to the compiler. Figure 9 shows the effect of restructuring techniques that increase the size of parallel loops in FlO52, one of the Perfect codes. The major subroutine of this program consists of two loops, each having a sequence of small inner loops. The first version of our compiler parallelized the inner loops only, which is represented by variant A. Variant B shows a program where the two outer loops were parallelized. In variant C these two loops were fused, thus the whole subroutine becomes one parallel loop.

The fusion of the outer loops (variant C) was made possible by replicating the code between the original outer loops on all clusters, adding redundant computations to the program. This technique has been applied successfully in other areas of the code, as well [19].

The parallel loops were stripmined into `CDOALL` / vector loops for the Alliant FX/80 and into `SDOALL` / `CDOALL` / vector loops for Cedar.

On the Alliant FX/80 architecture the resulting performance gain amounts to 50%, whereas on Cedar, a 100% speedup results, which illustrates the difference in startup latencies between the `CDO` and `SDO` loops and shows that compiling a structure of multiple small `SDOALL` loops into a single `SDOALL` can be a significant improvement on Cedar.

Our current compiler is often able to find large concurrent loops or to interchange parallel loops to an outer position (see Section 3.4). In other cases it fails because too many potential data dependences are detected or the outer loop is in a calling subroutine. These problems constitute important issues for the ongoing project.

5 Conclusions

We have designed and implemented the CEDAR FORTRAN language. The compiler and language support software have operated reliably since the first Cedar configuration came up in mid-1988.

We have retargeted KAP, a parallelizing restructurer, to automatically translate FORTRAN77 programs into CEDAR FORTRAN programs. We have extended the restructurer to cope with the challenges presented by the Cedar machine. The modified restructurer performs well on some linear algebra routines and synthetic loops. However, it does not perform as well on some large application programs.

We have engaged in an effort to study how to improve the current techniques for automatic parallelization, and in particular, how to improve our restructurer. We found several techniques which improved the performance of the Perfect Benchmarks programs on both the Alliant FX/80 and the Cedar. Some techniques are new and some are extensions to current techniques. Many of the techniques were useful for enhancing the recognition of parallelism where it exists. Such techniques are applicable to all parallel machines. We plan to incorporate all these techniques into later versions of our restructurer. When we have implemented our techniques, we hope that our restructurer will be able to automatically generate efficient parallel code for a wide range of existing sequential application programs that are written in FORTRAN77.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.

[2] William Blume and Rudolf Eigenmann. Performance Analysis of Parallelizing Compilers on the Perfect Benchmarks^{TM} Programs. *IEEE Transactions of Parallel and Distributed Systems*, November 1992.

[3] M. Booth and K. Misegades. Microtasking: A New Way to Harness Multiprocessors. *Cray Channels*, pages 24–27, 1986.

[4] David Callahan and Ken Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2(2):151–169, October 1988.

[5] S. C. Chen and D. J. Kuck. Time and Parallel Processor Bounds for Linear Recurrence Systems. *IEEE Trans. on Computers*, C-24(7):701–717, July, 1975.

[6] Rudolf Eigenmann. Towards a methodology of optimizing programs for high-performance computers. Technical Report 1178, Univ. of Illinois at Urbana-Champaign, Center for Supercomp. R&D, December 1991.

[7] Rudolf Eigenmann, Jay Hoeflinger, Zhiyuan Li, and David Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs.

Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA, pages 65–83, August 1991.

[8] K. A. Gallivan, R. J. Plemmons, and A. H. Sameh. Parallel Algorithms for Dense Linear Algebra Computations. *SIAM Review*, 32(1):54–135, March 1990.

[9] Kyle Gallivan, William Jalby, and Dennis Gannon. On the problem of optimizing data transfers for complex memory systems. *Proc. of 1988 Int'l. Conf. on Supercomputing, St. Malo, France*, pages 238–253, July 1988.

[10] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed Data Prefetching in Multiprocessors with Memory Hierarchies . *Proceedings of ICS'90, Amsterdam, The Netherlands*, 1:342–353, June 1990.

[11] M. Gupta and P. Banerjee. Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[12] Mark D. Guzzi, David A. Padua, Jay P. Hoeflinger, and Duncan H. Lawrie. Cedar Fortran and other vector and parallel Fortran dialects. *Journal of Supercomputing*, pages 37–62, March 1990.

[13] W. Ludwell Harrison, III and David Padua. PARCEL: Project for the Automatic Restructuring and Concurrent Evaluation of Lisp. *Proceedings of 1988 Int'l. Conf. on Supercomputing, St. Malo, France*, pages 527–538, July 1988.

[14] Jay Hoeflinger. Interval libraries for program analysis. Technical Report 1224, Center for Supercomputing Research and Development, 1992.

[15] Jay Hoeflinger. Run-time dependence testing by integer sequence analysis. Technical Report 1194, Center for Supercomputing Research and Development, 1992.

[16] Jay Hoeflinger. Cedar Fortran Programmer's Handbook. Technical report, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., October 1991. CSRD Report No. 1157.

[17] Christopher Alan Huson. An In-Line Subroutine Expander for Parafrase. Master's thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Sci., Dec., 1982.

[18] International Business Machines Corporation. *Parallel FORTRAN: Language and Library Reference*, 1988. SC23-0431-0.

[19] William Jalby, 1991. Private communication.

[20] Kuck & Associates, Inc., Champaign, Illinois. *KAP User's Guide*, 1988.

[21] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. *Proceedings of the 8th ACM Symp. on Principles of Programming Languages (POPL)*, pages 207–218, Jan., 1981.

[22] David B. Loveman. Program Improvement by Source-to-Source Transformation. *Journal of the ACM*, 24(1):121–145, January 1977.

[23] Ulrike Meier and Rudolf Eigenmann. Parallelization and Performance of Conjugate Gradient Algorithms on the Cedar Hierarchical-Memory Multiprocessor. *Proceedings of the 3rd ACM Sigplan Symp. on Principles and Practice of Parallel Programming, Williamsburg, VA*, pages 178–188, April 21-24, 1991.

[24] Samuel Midkiff and David Padua. Compiler Algorithms for Synchronization. *IEEE Transactions on Computers*, C-36(12):1485–1495, December 1987.

[25] David A. Padua and Michael J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):1184–1201, December 1986.

[26] M. Berry; D. Chen; P. Koss; D. Kuck; L. Pointer, S. Lo; Y. Pang; R. Roloff; A. Sameh; E. Clementi, S. Chin; D. Schneider; G. Fox; P. Messina; D. Walker, C. Hsiung; J. Schwarzmeier; K. Lue; S. Orszag; F. Seidl, O. Johnson; G. Swanson; R. Goodrum, and J. Martin. The Perfect Club Benchmarks: Effective Performance Evalution of Supercomputers. *Int'l. Journal of Supercomputer Applications, Fall 1989*, 3(3):5–40, Fall 1989.

[27] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes: The Art of Scientific Computing (FORTRAN Version)*. Cambridge University Press, 1989.

[28] Michael J. Wolfe. *Optimizing Compilers for Supercomputers*. PhD thesis, University of Illinois, October 1982.

[29] Hans P. Zima and Michael Gerndt. SUPERB: A tool for semi-automatic MIMD/SIMD parallelization. *Parallel Computing*, 6:1–18, 1988.