Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5067

ARL-TR-2556 **August 2001**

Using Loop-Level Parallelism to Parallelize Vectorizable Programs

Daniel M. Pressel
Computational and Information Sciences Directorate, ARL

Jubaraj Sahu and Karen R. Heavey
Weapons and Materials Research Directorate, ARL

Abstract

One of the major challenges facing "high performance computing" is the daunting task of producing programs that achieve acceptable levels of performance when run on parallel architectures. Although many organizations have been actively working in this area for some time, many programs have yet to be parallelized. Furthermore, some programs that were parallelized were done so for obsolete systems. These programs may run poorly, if at all, on the current generation of parallel computers. Therefore, a straightforward approach to parallelizing vectorizable codes is needed without introducing any changes to the algorithm or the convergence properties of the codes. Using the combination of loop-level parallelism and RISC-based shared memory SMPs has proven to be a successful approach to solving this problem.

Acknowledgments

The authors would like to thank Marek Behr for permission to use his results in this report. They would also like to thank the entire **CHSSI** CFD-6 team for their assistance in this work as part of that team. They would like to thank their many colleagues that have graciously assisted them in all aspects of the preparation of this report. Additional acknowledgments to Tom Kendall, Denice Brown, and the Systems Staff for all of their help. They would also like to thank the employees of Business Plus, especially Deborah Funk and Maria Brady who assisted in the preparation and editing of this report.

This work was made possible through a grant of computer time by the **DOD HPCM** Program. The time was spent at the **ARL-MSRC, NAVO-MSRC, NRL-DC, TARDEC-DC,** and **SPAWAR-DC** along with smaller amounts of time at other sites. Funding was provided as part of the CHSSI administered by the DOD HPCM Program.

INTENTIONALLY LEFT BLANK.

Contents

Acknowledgments iii

List of Figures vii

List of Tables ix

1. Introduction 1

2. Class of Codes 2

3. Symmetric Multiprocessors 4

4. The Approach 6

5. Results 11

6. Tools 14

7. The Effect of the NUMA Architecture 15

8. Related Work 17

9. Conclusion 19

10. References 21

Glossary 23

Distribution List 25

Report Documentation Page 29

INTENTIONALLY LEFT BLANK.

List of Figures

Figure 1. Predicted speedup for loops with various levels of parallelism. 10

Figure 2. The performance of the shared memory version of the F3D code when run on a modern scalable SMP (l-million grid point test case). 13

Figure 3. The performance of the shared memory version of the F3D code when run on a modem scalable SMP (59-million grid point test case). 13

INTENTIONALLY LEFT BLANK.

List of Tables

Table 1. The minimum amount of work (in cycles) per parallelized loop required for efficient execution 5

Table 2. The available amount of work (in cycles) per synchronization event for a l-million grid point zone 6

Table 3. Predicted speedup for a loop with 15 units of parallelism........ 10

Table 4. Measured performance of the RISC-optimized shared memory version of F3D........ 12

Table 5. Systems used in tuning/parallelizing the RISC-optimized shared memory version of F3D........ 14

INTENTIONALLY LEFT BLANK.

1. Introduction

One of the major challenges facing "high performance computing" is the daunting task of producing programs that achieve acceptable levels of performance when run on parallel architectures. In order to meet this challenge, the program must simultaneously achieve three goals:

(1) Achieve a reasonable level of parallel **speedup** at an acceptable cost.

(2) Demonstrate an acceptable level of serial performance so that **moderate-**sized problems do not require enormous levels of resources.

(3) Use an algorithm with a high enough level of algorithmic efficiency that the problem remains tractable.

Even though many organizations have been actively working in this area for 5–10 years (or longer), many programs have yet to be parallelized. Furthermore, some programs that were parallelized, were done so for what are now obsolete systems (e.g., **SIMD'** computers from Thinking Machines and **MASPAR),** and these programs run poorly, if at all, on the current generation of parallel computers. There has also been a problem that some approaches to parallelization can subtly change the algorithm and result in convergence problems when using large numbers of processors [1]. This is a common problem, particularly when using "domain decomposition" with "implicit" **CFD** codes. There are algorithmic solutions to this problem (e.g., multigrid codes or the use of a **preconditioner);** however, many of these solutions have problems in their own right (e.g., poor scalability).

At the other end of the spectrum, there are those who champion automatic parallelization. They expect to soon be able to parallelize production codes for efficient execution on modem production hardware. Unfortunately, as a general rule, this has not happened.

From discussions such as this, talks with numerous researchers, and the authors' research at the U.S. Army Research Laboratory, the following can be concluded:

- Writing parallel programs is a challenge.
- Writing efficient serial programs on today's RISC and **CISC** processors with their memory hierarchies (i.e., cache) is a challenge.
- Requiring the program to show near-linear scalability out to hundreds or thousands of processors greatly complicates matters.

'All items in bold are defined in the Glossary.

- Requiring the program to show portable performance across all or most modem parallel architectures greatly complicates matters.
- Modem processors are fast enough that for many problems that have traditionally been considered to be the sole domain of supercomputers, they may now be solvable using a moderate-sized system (e.g., 10–100 GFLOPS of peak processing power) given a sufficiently efficient algorithm and implementation.

Therefore, a straightforward approach to the parallelization of one or more important classes of codes is needed. This approach meets the following requirements:

- It works with a class of machines that has more than one member in it, but it need not include the entire universe of parallel computers.
- It does not require an unreasonable amount of effort.
- The results achieved by using this approach must satisfy the needs of the user community.
- The combined hardware and software costs must be acceptable.
- At least for small- to moderate-sized problems, it must be possible to complete the project before the equipment is obsolete.

The remainder of this report begins by discussing an approach developed at the U.S. Army Research Laboratory that was designed to meet all of these requirements for a large and important class of codes. Following this, the results of applying this approach to a specific production code are discussed. It concludes by considering some issues in greater detail and with a discussion of related work.

2. Class of Codes

For this project, the class of codes that was selected was the class of vectorizable codes. Of particular interest were those vectorizable codes that were considered to be nonparallelizable. A representative code was selected (F3D, an implicit CFD code) [2]. The following factors make this class of codes particularly interesting:

- From the mid-1970s to the mid-1990s, the terms “vector computers” and “supercomputers” were nearly synonymous (e.g., Cray C90).

- Many of the traditional vectorizable algorithms are known to be computationally more efficient than the algorithms most frequently associated with parallel computing.
- If one can efficiently tune one of these jobs to run on a parallel computer, then any job that exhibits an acceptable level of performance when using one processor of a C90 should exhibit an acceptable level of performance when using a modest number of RISC processors.
- Problem sizes that are 10 times greater (that is, 10 times overall, not 10 times in each direction) are likely to exhibit an acceptable level of performance when using a moderate-sized system (e.g., 10–100 GFLOPS of peak processing power).
- In recent years, SGI, SUN, Convex/HP, DEC/Compaq, and IBM have produced SMPs with this level of performance.
- It is our belief and experience that many of these systems are extremely well suited for running programs that have been parallelized using "loop-level parallelism" (e.g., OpenMP).
- Since vectorization is a form of loop-level parallelism, there is reason to hope that many vectorizable programs can be parallelized using this technique. While it is not always clear that this will be the best approach to parallelizing these programs, if one focuses on programs that are hard to parallelize, that objection should be effectively eliminated.

Unfortunately, this is not the end of the story. It would be nice if things were this simple, but they are not. Three important obstacles had to be conquered before this project could get off of the ground:

(1) Sufficiently powerful SMPs had to come onto the market. At the start of this project, no one had yet produced a RISC-based SMP with a peak speed of 10 GFLOPS, let alone 100 GFLOPS.

(2) All RISC processors use caches, and these processors were generally considered to be poorly suited to the needs of running scientific codes [3].

(3) Vectorization is normally applied to the innermost loop of a loop nest. However, as will be seen in section 3, when using OpenMP and similar techniques, one is well advised to parallelize outer (or at least middle) loops. In some cases, this necessitates the interchanging of loops in the loop nest. It may also be desirable to perform other transformations such as combining loops under a common outer loop.

3. Symmetric Multiprocessors

There are primarily four types of parallel computers in use today:

(1) Distributed memory systems. These are also sometimes referred to as "shared nothing." They are almost always programmed using a message-passing library (in recent years, **MPI** has become the standard library).

(2) Globally addressable memory. Cray Research frequently referred to the Cray T3D and T3E as shared memory architectures. A more appropriate description is globally addressable memory. While it is true that each processor can access all of the memory associated with a job, there are some important differences between these systems and true shared memory:

- The normal load and store instructions that are normally used to access the local memory cannot be used to access the remote memory.
- While it is theoretically possible to use the same instructions to perform loads and stores involving both local and remote memory, these instructions are not cache coherent.
- The introduction of the synchronization events needed to replace hardware coherency with software coherency can significantly interfere with the performance of the code. Since this requires replacing an automatic function of the hardware with a manual function of the code writer, it can greatly complicate efforts to produce valid code.

(3) Master slave. There have been a wide range of these systems produced in the past, including many of the SIMD-based systems. For the purpose of this report, we will discuss only shared memory systems using this organization. Their advantage was that this organization made it easy to port a uniprocessor operating system to a parallel computer. Their disadvantage was that their performance scaled very poorly. As a result, few such systems are still being produced.

(4) Symmetric multiprocessor. While more than one type of system might conform to this title, this report uses it only to refer to shared memory systems in which most, if not all, critical sections of the operating system can run on any of the processors (this is where the term symmetric comes from) [4]. Furthermore, if the processors have one or more levels of cache (as all RISC-based systems do, but which most vector-based systems lack), the system is cache coherent (otherwise, this report would refer to it as globally addressable).

Now consider the constraints that the choice of hardware places on the effort to efficiently parallelize a program. When using loop-level parallelism based on the use of compiler directives, there is no need to explicitly generate messages. The data flows between processors and main memory (in some cases, it even flows directly between two processors) as needed. (For the time being, questions of efficiency in a NUMA or **COMA** architecture are ignored; section 7 discusses these questions.) Therefore, the main cost of parallelization is assumed to be the synchronization cost associated with exiting a parallel section of code. On different machines and load factors, the synchronization cost (for scalable systems) ranges from 2,000 to l-million cycles (or more). From the standpoint of efficiency, it is preferable to keep these costs below 1% of the **runtime.** Table 1 shows how many cycles of work must be associated with the loop when run on a single processor in order to achieve this goal. It is important to keep in mind that the synchronization cost is highly dependent on the system load and the design of the memory system but is almost independent of the design of the processor. Therefore, as the memory latency (when expressed in cycles) for a random memory location continues to increase, the synchronization costs would also be expected to increase.

Table 1. The minimum amount of work (in cycles) per parallelized loop required for efficient execution.

Number of Processors Used	Hypothetical Synchronization Cost (in Cycles)		
	10,000	100,000	1,000,000
2	2,000,000	20,000,000	200,000,000
8	8,000,000	80,000,000	800,000,000
32	32,000,000	320,000,000	3,200,000,000
128	128,000,000	1,280,000,000	12,800,000,000

Two conclusions can be reached from Table 1:

(1) It is imperative to minimize the synchronization costs.

(2) Every effort should be made to maximize the amount of work per synchronization event (see Table 2).

Table 2 clearly demonstrates the advantage of parallelizing primarily outer loops. It also demonstrates the difficulty associated with the efficient parallelization of boundary condition routines. As such, it is frequently desirable to leave such routines unparallelized. However, for larger numbers of processors, this may result in problems with Amdahl's Law (too much time spent executing serial code).

Table 2. The available amount of work (in cycles) per synchronization event for a l-million grid point zone.

Problem Type	Grid Dimension	No. of Loop Iterations	Work Per Grid Point (in Cycles)		
			10	100	1.000
1-D	1,000,000	1,000,000	10,000,000	100,000,000	1,000,000,000
2-D	1,000 x 1,000	1,000			
Inner loop			10,000	100.000	1.000.000
Outer loop			10,000,000	100,000,000	1,000,000,000
Boundary condition			10,000	100,000	1,000,000
3-D	100 x 100 x 100	100			
Inner loop			1,000	10,000	100,000
Middle loop			100,000	1.000.000	10.000.000
Outer loop			10,000,000	100,000,000	1,000,000,000
Boundary condition - inner loop			1,000	10,000	100,000
Boundary condition - outer loop			100,000	1,000,000	10,000,000

4. The Approach

Since this approach is predicated on the assumption that one can achieve the desired level of performance when using modest-to-moderate numbers of processors, the first step is to improve the serial efficiency of the code. As was previously stated, the general consensus was that this would be difficult, possibly even impossible, to achieve. However, our experience showed that this need not be the case, which is not to say that the effort could be completed in a few days. There were four main concepts used in this part of the effort.

(1) Use a large memory SMP, which was a key enabling factor for this part of the effort. It is easier to perform serial tuning when working with serial code.

(2) Use traditional techniques such as reordering of Ioops and/or array indices, blocking, and matrix transpose operations to increase the locality of reference.

(3) Reorder the work so that rather than streaming data into and out of the processor, the code would stress maximizing the amount of work per cache miss.[1]

[1] In this respect, implicit CFD codes have a definite advantage over explicit CFD codes, since they do more work per time step.

(4) Adjust the size of scratch arrays so that they can be locked into cache. In particular, there were two key loops in F3D that had dependencies in two out of three directions. In order to vectorize these loops, the original programmers had to process data one plane at a time. This meant that the size of the scratch arrays were proportional to the size of a plane of data. Clearly for large problems, these scratch arrays were unlikely to fit into even the largest caches. However, since RISC processors do not rely on vectorization to achieve high levels of performance, it was possible to resize these arrays to hold just a single row or column of a single plane of data. The arrays now comfortably fit in a 1-MB cache for zone dimensions ranging up to about 1,000.

Therefore, one can see that while SMPs suffer from a larger memory latency (relative to the memory latency of most workstations and MPPs), the presence of large caches can more than make up for this limitation.

Once most of the serial tuning had been completed, it was possible to parallelize the code. Here, one of the strengths of loop-level parallelism really shined. With both HPF and traditional message-passing code, one is generally left with an all or nothing proposition. In other words, either all of the code must be parallelized, or the program does not run at all. When using loop-level parallelism in conjunction with an SMP, this is definitely not the case. Therefore, it is possible to use profiling to find the expensive loops and then to parallelize them one (or a few) at a time. This allows one to alternate between parallelization and debugging, which, in most cases, greatly simplifies the debugging process. It also allows one to better judge the effectiveness of what has been tried. In many cases, this greatly simplifies the task of parallelization compared to all-or-nothing approaches such as HPF or using message-passing libraries (e.g., MPI). This is not a unique observation, a similar observation was made by Frumkin et al. [5], as well as others.

As previously mentioned, vectorization is a form of loop-level parallelism. Therefore, in theory, one should be able to use OpenMP to parallelize vectorizable code. In practice, there are four' additional steps that need to be taken if one is to achieve high levels of performance:

(1) Since vectorization deals with inner loops, it is generally desirable to parallelize the outer loops, even though they are not involved in the vectorization of the program (see example 1).

(2) It is frequently desirable to merge loops together (see example 2). It should be noted that in some cases the merging of loops for purposes of parallelization can be combined with the serial tuning technique of code blocking.

```
C$doacross local (L,J,K)
    DO 10 L=1,LMAX
       DO 10 K=1,KMAX
          DO 10 J=1,JMAX
             Some computation with no dependencies in any
             direction
 10  CONTINUE
```

Example 1. Parallelizing an outer loop.

```
C$doacross local (L,J,K)
       DO 20 L=1,LMAX
          DO 10 K=1,KMAX
             DO 10 J=1,JMAX
                Body of the first loop
 10       CONTINUE
          DO 20 K=1,KMAX
             DO 20 J=1,JMAX
                Body of the second loop
 20    CONTINUE
```

Example 2. Merging loops to reduce synchronization costs.

(3) In some cases, one can significantly improve performance by parallelizing a loop in a parent subroutine (in some cases, one has to first create such a loop) (see example 3). It should be noted that in general this optimization reduces the number of synchronization events by l-3 orders of magnitude!

(4) One has to be very careful when dealing with arrays that are parts of a common block. When possible, it is desirable to move the arrays out of the common blocks (in some cases, this has to be done during the serial tuning as a first step in the resizing of those arrays).

Two things are important to remember:

(1) The more processors that are used, the harder it is to justify the overhead associated with the parallelization of boundary condition subroutines (as well as other inexpensive subroutines).

(2) The more time is spent in serial code, the harder it is to show benefit from using larger (e.g., 50+) numbers of processors.

Therefore, one is left with the problem of choosing between the lesser of two evils:

(1) As the number of processors increases, the speed first peaks and then starts to drop off (assuming that the synchronization costs are a function of the number of processors being used).

(2) As the number of processors increases, the speed approaches an asymptotic limit.

```
      DO 10 J=1,JMAX
C   NOTE:  We will assume that there are dependencies in the J
C   direction  that  inhibit  parallelization.
         CALL  SUBA(...)
C   SUBA batches up a 2-dimensional buffer for processing by
C   SUBB.
         CALL  SUBB(...)
C   SUBB has a dependency in one direction, which requires
C   processing a 2-dimensional buffer if the code is to be
C   vectorizable.
 10   CONTINUE
```

(a) **What the original code looked like.**

```
C$doacross local (L,J)
         DO 10 L=1,LMAX
            DO 10 J=1,JMAX
               CALL SUBA(...)
C   SUBA now batches up a l-dimensional buffer that easily fits
C   in a large cache,  for processing by SUBB.
               CALL  SUBB(...)
C   SUBB has a dependency in one direction, but since we are no
C   longer dealing with vector processors, this does not matter.
C   Therefore SUBB can safely process the much smaller
C   l-dimensional  buffer,
 10   CONTINUE
```

(b) **The cache optimized parallelized code.**

Example 3. Parallelizing a parent subroutine.

The net effect of this is that past a certain point, it is hard to show good speedup when using loop-level parallelism. While the authors are familiar with a number of researchers who felt that this limit would be around 4-16 processors, it is our experience that one should be able to efficiently use 30–128 processors (depending on the problem and the problem size).

Another related problem is that most people are used to thinking in terms of problems that have a nearly infinite level of parallelism. In such cases, the ideal speedup is linear. However, with loop-level parallelism, one is frequently dealing with parallelizing loops that have between 10 and 1,000 iterations. This means that the available parallelism is in the range of 10–1,000. When the number of processors is within roughly a factor of 10 of the available parallelism, the ideal speedup is no longer linear. Instead, the curve shows a distinctly stair-step shape. Both Table 3 and Figure 1 show where this shape comes from.

Table 3. Predicted speedup for a loop with 15 units of parallelism.

Number of Processors	Maximum Units of Parallelism Assigned to a Single Processor	Predicted Speed Up
1	15	1.000
2	8	1.875
3	5	3.000
4	4	3.750
5-7	3	5.000
8-14	2	7.500
15	1	15.000

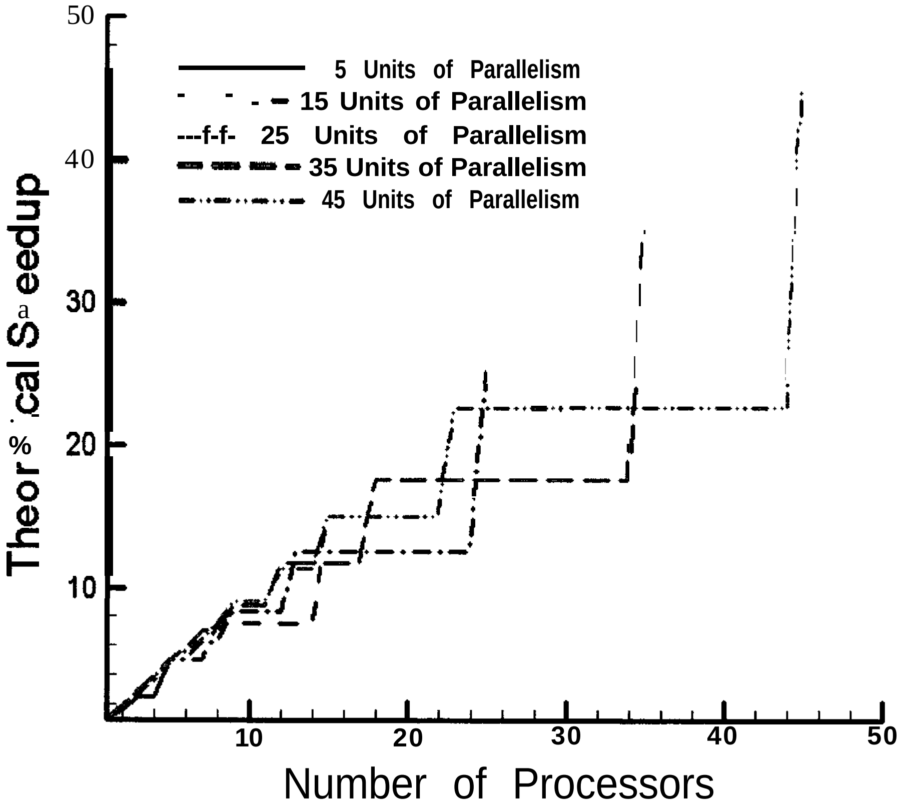

Figure 1. Predicted speedup for loops with various levels of parallelism.

5. Results

We have been able to do serial runs on an SGI Origin 2000 for problem sizes ranging from 1- to 200-million grid points without a significant decrease in the MFLOPS rate. This is the exact opposite of what was expected based on the literature at the time [3]. Furthermore, serial tuning on the SGI Power Challenge resulted in a speedup of more than a factor of 10. Attempts were made to measure the speedup on a Convex Exemplar SPP-1000. However, even though a 3-million grid point problem was being used, the vector version of the code was running so slowly that the job was killed before it had completed 10 time steps (the way things were going, this would probably have taken the better part of a day or more). The serial-tuned code completed 10 time steps in 70 min.

An interesting outcome from the parallelization of this code is that for larger numbers of processors, the performance as a function of the number of processors used is far from linear. Instead, the curve can best be described as a stair step. (See section 4 for an explanation of this effect.) When using loop-level parallelism with a three-dimensional code, there can be dependencies in one or more directions for key loops. With a maximum loop dimension of M, the available parallelism is roughly M. Therefore, one can expect to see jumps in performance at M/5, M/4, M/3, M/2, and M processors. This effect is demonstrated in Table 3 and Figure 1 and can be seen in the results in Table 4 and Figures 2 and 3 (e.g., the nearly flat performance between 48 and 64 processors for the l-million grid point test case and between 88 and 104 processors for the 59-million grid point test case).

Table 4 shows some representative results for the R12000-based SGI Origin 2000 (128 processors, 300 MHz) and the UltraSPARC II-based SUN HPC 10000 (64 processors, 400 MHz).* In reporting these results, there is the question of what is the best metric to use. We prefer to avoid speedup, since it fails to describe the actual performance of the code and can actually favor poorly performing codes (the lower the serial performance, the easier it is to show good speedup). From the user's perspective, what really matters are metrics such as run time and turnaround time. However, as the number of processors approach the available parallelism, the predicted run time should asymptotically approach some low value, making it difficult to evaluate the performance of the code. Our preferred metric is "time steps/hour," since it allows the user to easily estimate

*On the SGI Origin, the C$doacross directives were used. On the SUN HPC 10000, the SUN specific PCF directives were used, since at the time, SUN did not support either SGI's C$doacross directives or the OpenMP directives.

Table 4. Measured performance of the RISC-optimized shared memory version of F3D.

Number of Processors Used	Problem Size (Million Grid Points)	Performance			
		SUN HPC 10000		SGI R12000 Origin 2000	
		Time Steps/Hr	MFLOPS	Time Steps/Hr	MFLOPS
1	1[a]	138	1.80E2	181	2.37E2
32	1	2,786	3.64E3	2,877	3.76E3
48	1	3,093	4.04E3	3,545	3.63E3
64	1	2,819	3.69E3	3,694	4.83E3
72	1	N/A	N/A	4,105	5.37E3
88	1	N/A	N/A	5,087	6.65E3
1	59[b]	2.1	1.63E2	2.3	1.79E3
32	59	45	3.50E3	59	4.59E3
48	59	61	4.75E3	73	5.68E3
64	59	73	5.68E3	91	7.08E3
72	59	N/A	N/A	101	7.86E3
88	59	N/A	N/A	128	9.96E3
104	59	N/A	N/A	131	1.02E4
112	59	N/A	N/A	144	1.12E4
120	59	N/A	N/A	150	1.17E4
124	59	N/A	N/A	153	1.19E4

[a]The l-million grid point test case consists of three zones with dimensions of 15 x 75 x 70, 87 x 75 x 70, and 89 x 75 x 70.

[b]The 59-million grid point test case consists of three zones with dimensions of 29 x 450 x 350, 173 x 450 x 350, and 175 x 450 x 350.

what the run time should be (not counting start-up and termination costs, which have been eliminated from our results). This metric also has the advantage that for problems with large amounts of parallelism, it gives the linear performance curve that one normally equates with parallel programs. Another useful metric that has been included in Table 4 is the delivered MFLOPS, which allows one to determine not only the parallel efficiency, but also the serial efficiency of our implementation. The peak speed of a processor on the SUN system is 800 MFLOPS and 600 MFLOPS on the SGI system. From the results, one can see that the per processor delivered performance of the two systems is actually very similar. This is probably the result of the two vendors taking different approaches in designing their chips. Some vendors prefer to make the fastest chips possible, even though they have a lot of hazards that can limit their delivered performance, while other vendors prefer to make somewhat slower but friendlier chips. Both approaches are valid and can result in a good product. But as our results demonstrate, it is important to compare products based on their delivered performance, not their peak performance.

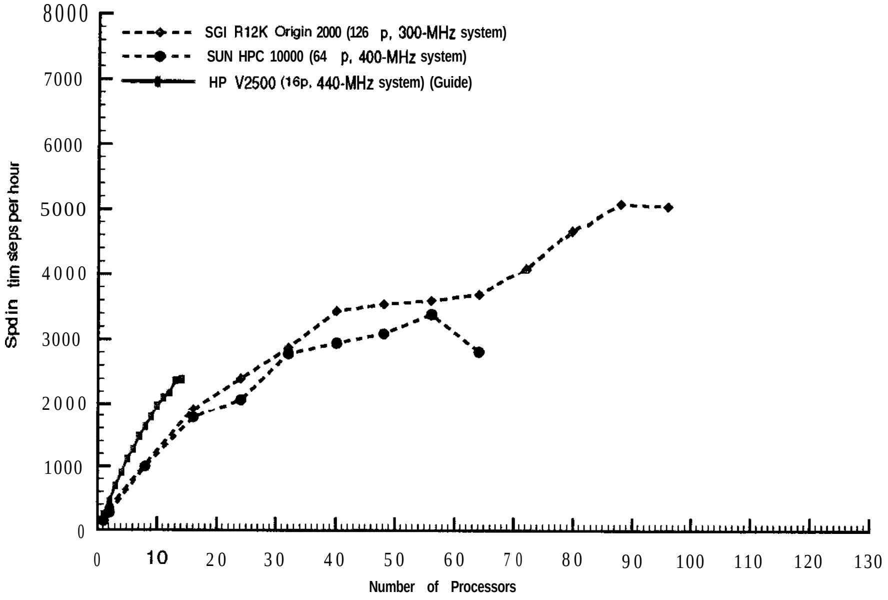

Figure 2. The performance of the shared memory version of the F3D code when run on a modern scalable SMP (l-million grid point test case).*

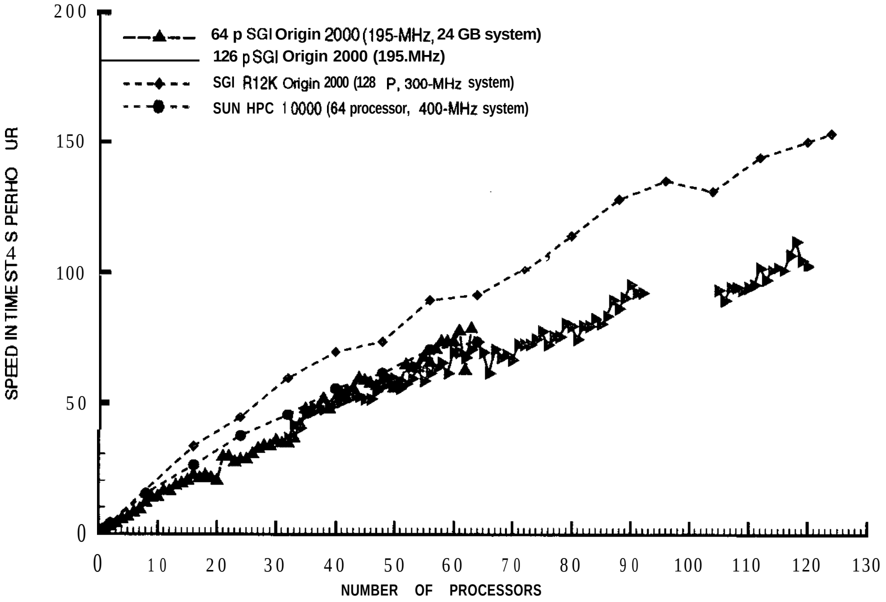

Figure 3. The performance of the shared memory version of the F3D code when run on a modem scalable SMP (59-million grid point test case).*

* The speeds have been adjusted to remove startup and termination costs.

6. Tools

In performing the serial tuning, the principal tools used were various profiling tools. Initially, this meant using prof with and without pixie. Without pixie, prof measures the actual run time for the individual subroutines. With pixie, prof measures the theoretical run time for the subroutines, assuming an infinitely fast memory system. By subtracting those two sets of numbers, one can then estimate the cost of cache and TLB misses. Fortunately, most of today's systems come with tools that allow one to directly measure those values (on systems lacking those tools, codes can be instrumented using PAPI, which is being developed at the University of Tennessee, Knoxville, TN, as a project for the PTOOLS organization).

After tuning for cache and TLB misses, a few loops had a low cache/TLB miss rate but were still expensive to run. Additional hand optimizations guided by assembly code dumps were performed to achieve a higher data reuse rate at the register level while eliminating unnecessary register spilling, pipeline stalls, and low instruction issue rates from excessive numbers of loads and stores. A key aspect of this phase of the tuning was to run the program on as wide a range of RISC-based systems as possible to better understand which optimizations would be of universal value. Table 5 contains a list of systems used in this effort. Using this wide range of systems and compilers allowed tuning for a wider range of TLB and cache sizes. It also allowed us to better anticipate what types of codes current production compilers could handle without producing performance problems.

Table5. Systems used in tuning/parallelizing the RISC-optimized shared memory version of F3D.

SGI R4400-based Challenge and Indigo 2
SGI R8000- and R10000-based Power Challenges
SGI R10000- and R12000-based Origin 2000s
SUN SuperSPARC-based SPARCCenter 2000
SUN Ultra SPARC II-based HPC 10000
Convex HP PA-7100-based SPP-1000 and HP PA-ROO-based SPP-1600
HP PA-8500-based V-Class

Another key aspect of this effort was to validate the results [7]. There were several stages to this effort, ranging from quick and dirty tests involving only a few time steps, to more elaborate tests performed on fully converged solutions, to finally a complete manual review of the code as part of a formal validation and verification exercise. Unfortunately, when performing some of the intermediate level tests, a mistake that had been introduced into the code would sometimes be found. One of the most useful tools for locating these mistakes was to update the version number of the code daily so that one could go back and find which was the first version to have the bug. One could then use "diff' to identify what the differences were between that version of the code and the previous one. In most cases, this was sufficient in rapidly identifying and fixing the bug (in all of the affected versions). One extreme example of applying this technique occurred early on. In reordering the indices of several key arrays throughout the program, changing almost every executable line of code in the entire program became necessary. Furthermore, all of the lines had to be changed at the same time. Unfortunately, the odds of getting this right proved to be vanishingly small. Additionally, manual inspection of the code failed to find the problems. Fortunately, after going through the entire exercise a second time, we were able to diff the two modified versions of the code, locate the mistakes, and get a code that produced the correct answers in about half the time as the previous version of the code.

The principal tools used for evaluating the performance of the parallelized code were various versions of profiling tools (e.g., Speed Shop on the SGI systems, CXPERF on the Convex and HP systems, and Loop Tool on the SUN HPC 10000). The tools, in combination with simply timing the runs for various combinations of problem sizes, numbers of processors, and systems, allowed us to identify the issues that affected parallel speedup. The single biggest issue was that since parallelization was being done incrementally, we needed to know which loops were expensive enough to justify being parallelized (both in terms of the effort and additional overhead that would be introduced). Once this was well in hand, the main remaining issue was to understand how the variable costs associated with memory latency and bandwidth in a NUMA environment were affecting the code performance. Unfortunately, experience indicated that not all NUMA platforms were created equal, and these performance problems on the Convex Exemplars were never satisfactorily solved. Fortunately, the results on the SGI system and the SUN HPC 10000 were much better, in part due to all of the work done to try to make the code perform better on the Convex Exemplar [6].

7. The Effect of the NUMA Architecture

With memory latencies ranging from 310-945 NS in a 128-processor SGI Origin 2000 [8], without using any form of out-of-order execution and/or prefetching, one sees a range of usable per processor bandwidths of 412 MB/second down to 135 MB/second. Clearly for a shared memory program with a poor cache hit rate and a high proportion of off node accesses, this results in a significant performance problem. One potential solution is to make use of out-of-order execution and/or prefetching to overlap cache misses in an attempt to achieve a lower effective memory latency. Unfortunately, the maximum per processor usable bandwidth for off node accesses is estimated to be only 195 MB/second, which severely limits the effectiveness of this approach. Our solution to this problem was to produce a highly tuned code with a low enough cache miss rate that the NUMA nature of the Origin 2000 did not matter. According to the output of Perfex when our code is run on a 180-MHz R10000-based Origin 200 using a single processor, we have only 68 MB/second of memory traffic. Since this is far less than the 135-195 MB/second of usable bandwidth for off node accesses on the Origin 2000, we have been able to treat the Origin 2000 as though it had Uniform Memory Access. Unfortunately, this approach did not work nearly as well on the more heavily NUMA systems such as the Convex Exemplar.

This is not to say that the memory systems on the SGI Origin 2000 and the SUN HPC 10000 did not cause problems. The traditional shared memory systems (e.g., the Cray C90 or the SGI Power Challenge) store data in small blocks or lines, with successive blocks or lines being interleaved between multiple memory banks. On some systems, the size of a block may be as small as a single word (e.g., 8 bytes), while on most cache-based systems, a cache line would be treated as a single block (e.g., 64 or 128 bytes). On systems that group memory and processors into nodes (e.g., the SGI Origin 2000, the SUN HPC 10000, and the hypernodes on the Convex Exemplar), the unit of interleaving becomes a page of memory (e.g., 4-16 KB). In that case, one can easily have data from the same page being shared by multiple processors. In extreme cases, this results in a severe amount of contention with a resulting drop in performance. It is important to note that no amount of page migration solves this problem-neither does data placement directives. Data replication/caching can help. But the best solution is to initially avoid the problem. It is also interesting to note that as far as the authors know, there are no tools that identify this problem. The best way to identify the problem now is to profile fixed size runs with varying numbers of processors and look for subroutines that are consuming additional CPU cycles as the number of processors increases. Using tools such as Perfex or Speedshop can determine if the number of cache misses is remaining relatively constant. If this is the case, then one almost certainly has a problem with contention. Example 4 illustrates the three cases (ideal, acceptable, and unacceptable) of concern. Please note that even though Example 4c is using a STRIDE-N access pattern to batch up the buffer, it can still have an acceptable cache miss rate. Unfortunately, the process of batching up the buffer can result in an unacceptable amount of contention on some of these systems. To confuse matters further, and for reasons that the authors were never able to fully explain, the SGI Origin 2000 and the

```
        DIMENSION    A(JMAX,KMAX,LMAX)
CSdoacross local (J,K,L)
        DO 10 L=1,LMAX
           DO 10 K=1,KMAX
              DO 10 J=1,JMAX
                 A(J,K,L) = . . .
 10     CONTINUE
```

(a) An example of the best possible access ordering.

```
        DIMENSION    A(JMAX,KMAX,LMAX)
CSdoacross  local  (K,J,L)
        DO 10 K=1,KMAX
           DO 10 L=1,LMAX
              DO 10 J=1,JMAX
                 A(J,K,L) = . . .
 10 CONTINUE
```

(b) An example of an acceptable, but less desirable ordering.

```
        DIMENSION    A(JMAX,KMAX,LMAX),    BUFFER(KMAX)
CSdoacross local (J,L,K,BUFFER)
        DO 20 J=1,JMAX
           DO 20 L=1,LMAX
              DO 10 K=1,KMAX
                 BUFFER  =  A(J,K,L)
 10           CONTINUE
              DO 20 K=1,KMAX
                 Perform extensive calculations using BUFFER
 20     CONTINUE
```

(c) An example of an unacceptable ordering.

Example 4. The effect of memory access patterns on contention.

SUN HPC 10000 exhibited this problem under different conditions, thereby making it necessary to eliminate all instances of this type of code from the program.

8. Related Work

The simplest approach to using loop-level parallelism is to use an automatic parallelizing compiler. Unfortunately, as Michael Wolfe (the author of “High Performance Compilers for Parallel Computing,” Addison-Wesley, 1996) has pointed out-“parallelizing compilers don’t work and they never will [9].” A slightly more sophisticated approach has been suggested by Dixie Hisley of the U.S. Army Research Laboratory. This approach uses a combination of compiler directives and hand tuning to parallelize those expensive loops that the automatic parallelizing compilers are unable to handle on their own. The remaining loops are left for the compiler to figure out. In some cases, this was shown to increase the scalability of the code from 8 to 16 processors, with little or no additional work over the use of compiler directives (e.g., C$doacross) and hand tuning on their own. In contrast, using only an automatic parallelizing compiler in this case actually produced parallel slowdown [10].

An excellent comparison, using the combination of hand tuning, automatic parallelization, and compiler directives, to the HPF and CAPTools parallelization tools was presented at SC98 [5]. It found that all three approaches had merit, although none produced good results when used on a fully automatic basis. In many cases, the results from using compiler directives (e.g., C$doacross or CAPTools specific directives) were as good as those produced by hand parallelizing the code using MPI.

Marek Behr attempted to parallelize the F3D program on the Cray T3D using the CRAFT programming model. Unfortunately, this effort had to be abandoned due to poor levels of performance (something that was a common complaint with this programming model) [11]. He then proceeded to manually parallelize the code using message passing calls (SHMEM on the Cray T3D, Cray T3E, and SGI Origin 2000, and MPI on other platforms) to implement loop-level parallelism. While this approach worked and produced a credible level of performance, it was significantly more difficult to implement. Furthermore, because many of the target platforms (e.g., the Cray T3D, Cray T3E, and IBM SP with the Power 2 Super Chip) had caches ranging in size from 16 to 128 KB, it was impossible to perform many of the cache optimizations that we performed using caches with l-8 MB of memory [12].

An approach similar to the one we used in tuning and parallelizing F3D was used by James Taft at NASA Ames Research Center to tune and parallelize the ARC3D code for the SGI Origin 2000 [13]. More recently, Mr. Taft has used multiple levels of shared memory parallelism (MLP)—an approach that he has successfully demonstrated with the Overflow code and more recently, several other commonly used codes at NASA Ames Research Center [14, 15]. Straight loop-level parallelism and MLP appear to be complementary techniques, each with their own strengths and weaknesses.

There is also a significant body of research involving the use of software distributed shared memory [16]. Unfortunately, there is generally a significant performance penalty when using these systems, which keeps them from being widely accepted for production environments. One of the key problems is that modem SMP and MPP systems usually have per processor memory bandwidths ranging from 200 MB/second to over 1 GB/second, with memory latencies of 100–1,000 NS. In contrast, the communications bandwidth for most workstation clusters and many MPPs is rarely much better than 100 MB/second on a per processor basis, with a latency that is frequently in the range of 50-100 microseconds for the better systems (Note that the primary exception to this rule is the Cray T3E when using SHMEM). Attempting to maintain coherency with the 128-byte granularity used in the SGI Origin 2000 with a latency of 100 microseconds results in a per processor bandwidth for off node accesses of 1.3 MB/second. For programs that are parallelized in more than one direction and therefore inevitably have a high level of off node memory accesses, this low level of performance is virtually impossible to overcome on today's high performance systems. Even in hardware implementations of a COMA environment, keeping all of the processors for a single job on a single node is highly desirable [17].

9. Conclusion

It was demonstrated that careful tuning at the implementation level can make a significant difference in the performance of code running on RISC-based systems. Furthermore, it was shown that there are two key enabling technologies for this effort:

(1) Large caches.

(2) Access to a large amount of main memory, something that SMPs excel at.

Additionally, it was shown that the use of loop-level parallelism can be of significant benefit when trying to parallelize certain classes of vectorizable code. However, several limitations to this approach were discussed:

- The need for sufficiently large and powerful SMPs.
- The importance of tuning the code for a high level of serial efficiency.
- The limitations on scalability (overhead, Amdahl's Law, and stair stepping).

In summary, the combination of loop-level parallelism and SMPs represents a powerful tool for the parallelization of vectorizable programs. As long as everyone understands the inherent limitations, it should have a most productive future.

INTENTIONALLY LEFT BLANK.

10. References

1. Wang, G., and D. K. Tafti. "Performance Enhancement on Microprocessors With Hierarchical Memory Systems for Solving Large Sparse Linear System." *International Journal of Supercomputing Applications, 1997.*

2. Steger, J. L., S. X. Ying, and L. B. Schiff. "A Partially Flux-Split Algorithm for Numerical Simulation of Compressible Inviscid and Viscous Flows." *Proceedings of the Workshop on CFD,* Davis, CA, 1986.

3. Bailey, D. H. "Microprocessors and Scientific Computing." Proceedings of *Supercomputing* 93, Los Alamitos, CA, 1993.

4. Schimmel, C. "UNIX Systems for Modern Architectures, Symmetric Multiprocessing, and Caching for Kernel Programmers." Reading, MA: Addison-Wesley, 1994.

5. Frumkin, M., M. Hribar, H. Jin, A. Waheed, and J. Yan. "A Comparison of Automatic Parallelization Tools/Compilers on the SGI Origin 2000." *Proceedings for the* SC98 Conference, Supercomp Organization, IEEE Computer Society, and ACM, http:/ /www.supercomp.org/sc98/TechPapers, 1998.

6. Pressel, D. M. "Results From the Porting of the Computational Fluid Dynamics Code F3D to the Convex Exemplar (SPP-1000 and SPP-1600)." ARL-TR-1923, U.S. Army Research Laboratory, Aberdeen Proving Ground, MD, 1999.

7. Edge, H. L., J. Sahu, W. B. Sturek, D. M. Pressel, K. R. Heavey, P. Weinacht, C. K. Zoltani, C. J. Nietubiu, J. Clarke, M. Behr, and P. Collins. "Common High Performance Computing Software Support Initiative (CHSSI) Computational Fluid Dynamics (CFD) Project Final Report: ARL Block-Structured Gridding Zonal Navier-Stokes Flow (ZNSFLOW) Solver Software." ARL-TR-2084, U.S. Army Research Laboratory, Aberdeen Proving Ground, MD, 2000.

8. Laudon, J., and D. Lenoski. 'The SGI Origin: A ccNUMA Highly Scalable Server." Proceedings of the 24th Annual International Symposium on Computer Architecture (ISCA '97), Denver, CO, 2-4 June 1997, IEEE Computer Society, Los Alamitos, CA.

9. Theys, M. D., T. D. Braun, and H. J. Siegel. 'Widespread Acceptance of General-Purpose, Large-Scale Parallel Machines: Fact, Future, or Fantasy?" *IEEE Concurrency* Parallel, *Distributed, and Mobile Computing, vol. 6,* no. 1, 1998.

10. Hisley, D. M., G. Agrawal, and L. Pollock. ‘Performance Studies of the Parallelization of a CFD Solver on the Origin 2000.” Proceedings for the 21st Army Science Conference, Department of the Army, 1998.

11. Oberlin, S. “Keynote Address at the International Symposium on Computer Architecture (ISCA ’99),” 1999. (At the time, S. Oberlin had been the Vice President for Software at SGI, having previously been the Vice President for Hardware.)

12. Behr, M., D. M. Pressel, and W. B. Sturek, Jr. “Comments on CFD Code Performance on Scalable Architectures.” *Computer Methods in Applied Mechanics and Engineering, vol. 190,* pp. 263–277, 2000.

13. Taft, J. “Initial SGI Origin 2000 Tests Show Promise for CFD Codes.” *NAS* News, vol. 2, no. 25, NASA Ames Research Center, 1997.

14. Taft, J. R. “Shared Memory Multi-Level Parallelism for CFD, Overflow-MLP: A Case Study.” Presented at the Cray User Group Origin 2000 Workshop, Denver, CO, 11-13 October 1998.

15. Taft, J. R. “Achieving 60 GFLOP/S on the Production CFD CODE OVERFLOW-MLP.” Presented at WOMPAT 2000, Workshop on OpenMP Applications and Tools, San Diego, CA, 6-7 July 2000.

16. Keleher, P., A. L. Cox, S. Dwarkadas, and W. Zwaenepoel. “TreadMarks: Distributed Shared Memory on Standard Workstations and Operating Systems.” Proceedings of the Winter 94 Usenix Conference, http://-.cs.rice.edu/~willy/TreadMarks/papers.html, 1994.

17. Hagersten, E., and M. Koster. “WildFire: A Scalable Path for SMPs.” Proceedings of the 5th International Symposium on High Performance Computer Architecture (HPCA), Orlando, FL, 9-13 January 1999, IEEE Computer Society, Los Alamitos, CA.

Glossary

Cache	A small high-speed memory that sits between the processor and main memory, which is designed to store values that are likely to be needed by the processor in the very near future.
CFD	Computational Fluid Dynamics
CISC	Complicated Instruction Set Computer
COMA	Cache Only Memory Architecture
DC	Distributed Center
GFLOPS	Giga Floating-Point Operations Per Second
HPF	High Performance Fortran
MFLOPS	Mega Floating-Point Operations Per Second
MPI	Message Passing Interface
MPP	Massively Parallel Processor
MSRC	Major Shared Resource Center
NUMA	Non Uniform Memory Access
RISC	Reduced Instruction Set Computer
SIMD	Single Instruction Multiple Data
SMP	Symmetric Multiprocessor
TLB	Translation Lookaside Buffer

INTENTIONALLY LEFT BLANK.

NO. OF
COPIES ORGANIZATION

2 DEFENSE TECHNICAL
INFORMATION CENTER
DTIC OCA
8725 JOHN J KINGMAN RD
STE 0944
FT BELVOIR VA 22060-6218

1 HQDA
DAMOFDT
400 ARMY PENTAGON
WASHINGTON DC 20310-0460

1 OSD
OUSD(A&T)/ODDR&E(R)
DRRJTREW
3800 DEFENSE PENTAGON
WASHINGTON DC 20301-3800

1 COMMANDING GENERAL
us ARMY MATERIEL CMD
AMCRDA TF
5001 EISENHOWER AVE
ALEXANDRIA VA 22333-0001

1 INST FOR ADVNCD TCHNLGY
THE UNIV OF TEXAS AT AUSTIN
3925 W BRAKER LN STE 400
AUSTIN TX 78759-5316

1 DARPA
SPECIAL PROJECTS OFFICE
J CARLINI
3701 N FAIRFAX DR
ARLINGTON VA 22203-1714

1 US MILITARY ACADEMY
MATH SCI CTR EXCELLENCE
MADN MATH
MAJ HUBER
THAYER HALL
WEST POINT NY 109961786

1 DIRECTOR
US ARMY RESEARCH LAB
AMSRL D
DRDSMITH
2800 POWDER MILL RD
ADELPHI MD 20783-1197

NO. OF
COPIES ORGANIZATION

1 DIRECTOR
US ARMY RESEARCH LAB
AMSRL CI AI R
2800 POWDER MILL RD
ADELPHI MD 20783-1197

3 DIRECTOR
US ARMY RESEARCH LAB
AMSRL CI LL
2800 POWDER MILL RD
ADELPHI MD 20783-1197

3 DIRECTOR
US ARMY RESEARCH LAB
AMSRL CI IS T
2800 POWDER MILL RD
ADELPHI MD 20783-1197

ABERDEEN PROVING GROUND

2 DIR USARL
AMSRL CI LP (BLDG 305)

NO. OF COPIES ORGANIZATION

1 PROGRAM DIRECTOR
C HENRY
1010 N GLEBE RD STE 510
ARLINGTON VA 22201

1 DPTY PROGRAM DIRECTOR
L DAVIS
1010 N. GLEBE RD STE 510
ARLINGTON VA 22201

1 DISTRIBUTED CENTERS
PROJECT OFFICER
V THOMAS
1010 N GLEBE RD STE 510
ARLINGTON VA 22201

1 HPC CTRS PROJECT MNGR
J BAIRD
1010 N GLEBE RD STE 510
ARLINGTON VA 22201

1 CHSSI PROJECT MNGR
L PERKINS
1010 N GLEBE RD STE 510
ARLINGTON VA 22201

1 RICE UNIVERSITY
MECHANICAL ENGRNG &
MATERIALS SCIENCE
M BEHR MS 321
6100 MAIN ST
HOUSTON TX 77005

1 J OSBURN CODE 5594
4555 OVERLOOK RD
BLDG A49 RM 15
WASHINGTON DC 20375-5340

1 NAVAL RSCH LAB
J BORIS CODE 6400
4555 OVERLOOK AVE SW
WASHINGTON DC 20375-5344

1 WL FIMC
B STRANG
BLDG 450
2645 FIFTH ST STE 7
WPAFB OH 45433-7913

1 NAVAL RSCH LAB
R RAMAMURTI CODE 6410
WASHINGTON DC 20375-5344

NO. OF COPIES ORGANIZATION

1 ARMY AEROFLIGHT
DYNAMICS DIRECTORATE
RMEAKINMS2581
MOFFETT FIELD CA 94035-1000

1 NAVAL RSCH LAB
HEAD OCEAN DYNAMICS
& PREDICTION BRANCH
J W MCCAFFREY JR CODE 7320
STENNIS SPACE CENTER MS
39529

1 US AIR FORCE WRIGHT LAB
WLFIM
JJSSHANG
2645 FIFTH ST STE 6
WPAFB OH 45433-7912

1 US AIR FORCE PHILIPS LAB
OLAC PL RKFE
CAPTSGWIERSCHKE
10 E SATURN BLVD
EDWARDS AFB CA 935247680

1 NAVAL RSCH LAB
CODE 6390
DR D PAPACONSTANTOPOULOS
WASHINGTON DC 20375-5000

1 AIR FORCE RSCH LAB DEHE
R PETERKIN
3550 ABERDEEN AVE SE
KIRTLAND AFB NM 87117-5776

1 NAVAL RSCH LAB
RSCH OCEANOGRAPHER CNMOC
G HEBURN
BLDG 1020 RM 178
STENNIS SPACE CENTER MS
39529

1 AIR FORCE RSCH LAB
INFORMATION DIRECTORATE
R W LINDERMAN
26 ELECTRONIC PKWY
ROME NY 134414514

1 SPAWARSYSCEN D4402
R A WASILAUSKY
BLDG 33 RM 0071A
53560 HULL ST
SAN DIEGO CA 921523001

NO. OF
COPIES ORGANIZATION

1 USAE WATERWAYS
EXPERIMENT STATION
CEWES HV C
J P HOLLAND
3909 HALLS FERRY RD
VICKSBURG MS 39180-6199

1 US ARMY CECOM RSCH
DEVELOPMENT & ENGRNG CTR
AMSEL RD C2
B S PERLMAN
FT MONMOUTH NJ 07703

1 SPACE & NAVAL WARFARE
SYSTEMS CTR
K BROMLEY CODE D7305
BLDG 606 RM 325
53140 SYSTEMS ST
SAN DIEGO CA 92152-5001

1 DIRECTOR
DEPARTMENT OF ASTRONOMY
P WOODWARD
356 PHYSICS BLDG
116 CHURCH ST SE
MINNEAPOLIS MN 55455

1 RICE UNIVERSITY
MECHANICAL ENGRNG &
MATERIALS SCIENCE
T TEZDUYAR MS 321
6100 MAIN ST
HOUSTON TX 77005

1 ARMY HIGH PERFORMANCE
COMPUTING RSCH CTR
B BRYAN
1200 WASHINGTON AVE
s MINNEAPOLIS MN 55415

1 ARMY HIGH PERFORMANCE
COMPUTINGRSCHCTR
G V CANDLER
1200 WASHINGTON AVE
s MINNEAPOLIS MN 55415

1 NAVAL CMD CNTRL &
OCEAN SURVEILLANCE CTR
L PARNELL
NCCOSC RDTE DIV D3603
49590 LASSING RD
SAN DIEGO CA 92152-6148

NO. OF
COPIES ORGANIZATION

1 UNIVERSITY OF TENNESSEE
COMPUTER SCIENCE DEFT
S MOORE
1122 VOLUNTEER BLVD
STE 203
KNOXVILLE TN 379963450

ABERDEEN PROVING GROUND

26 DIR USARL
AMSRL CI
N RADHAKRISHNAN
AMSRL CI H
c NIETUBICZ
W STUREK
AMSRL CI HC
D PRESSEL
D HISLEY
R NAMBURU
R VALISETTY
D SHIRES
R MOHAN
M HURLEY
P CHUNG
J CLARKE
C ZOLTANI
A MARK
AMSRL CI HS
D BROWN
R PRABHAKARAN
T PRESSLEY
T KENDALL
P MATTHEWS
K SMITH
AMSRL WM BC
J SAHU
K HEAVEY
P WEINACHT
J DESPIRITO
P PLOSTINS
AMSRL WT BF
HEDGE

INTENTIONALLY LEFT BLANK.

REPORT DOCUMENTATION PAGE	**Form** *Approved* **OMB No. 07044188**

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed. and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project(0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE August 2001	3. REPORT TYPE AND DATES COVERED Final, October 1999 – April 2001

4. TITLE AND SUBTITLE
Using Loop-Level Parallelism to Parallelize Vectorizable Programs

5. FUNDING NUMBERS
665803.73 1

6. AUTHOR(S)
Daniel M. Pressel, Jubaraj Sahu, and Karen R. Heavey

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
U.S. Army Research Laboratory
ATTN: AMSRL-CI-HC
Aberdeen Proving Ground, MD 2 1005-5067

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-2556

9. SPONSORING/MONITORING AGENCY NAMES(S) AND ADDRESS(ES)

10.SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT*(Maximum 200* words)

One of the major challenges facing "high performance computing" is the daunting task of producing programs that achieve acceptable levels of performance when run on parallel architectures. Although many organizations have been actively working in this area for some time, many programs have yet to be parallelized. Furthermore, some programs that were parallelized were done so for obsolete systems. These programs may run poorly, if at all, on the current generation of parallel computers. Therefore, a straightforward approach to parallelizing vectorizable codes is needed without introducing any changes to the algorithm or the convergence properties of the codes. Using the combination of loop-level parallelism and RISC-based shared memory SMPs has proven to be a successful approach to solving this problem

14. SUBJECT TERMS
parallel programming, symmetric multiprocessor, loop-level parallelism supercomputer, high performance computing

15. NUMBER OF PAGES
32

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL

INTENTIONALLY LEFT BLANK.

USER EVALUATION SHEET/CHANGE OF ADDRESS

This Laboratory undertakes a continuing effort to improve the quality of the reports it publishes. Your comments/answers to the items/questions below will aid us in our efforts.

1. ARL Report Number/Author ARL-TR-2556 (Pressel) Date of Report August 2001

2. Date Report Received ____________________

3. Does this report satisfy a need? (Comment on purpose, related project, or other area of interest for which the report will be used.) ____________________

4. Specifically, how is the report being used? (Information source, design data, procedure, source of ideas, etc.) ____________________

5. Has the information in this report led to any quantitative savings as far as man-hours or dollars saved, operating costs avoided, or efficiencies achieved, etc? If so, please elaborate. ____________________

6. General Comments. What do you think should be changed to improve future reports? (Indicate changes to organization, technical content, format, etc.) ____________________

CURRENT
ADDRESS

Organization

Name E-mail Name

Street or P.O. Box No.

City, State, Zip Code

7. If indicating a Change of Address or Address Correction, please provide the Current or Correct address above and the Old or Incorrect address below.

OLD
ADDRESS

Organization

Name

Street or P.O. Box No.

City, State, Zip Code

(Remove this sheet, fold as indicated, tape closed, and mail.)
(DO NOT STAPLE)

DEPARTMENT OF THE ARMY

OFFICIAL BUSINESS

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT NO 0001,APG,MD

POSTAGE WILL BE PAID BY ADDRESSEE

DIRECTOR
US ARMY RESEARCH LABORATORY
ATTN AMSRL CI HC
ABERDEEN PROVING GROUND MD 21005-5067