Benchmarking Fortran and Ada Programs on Parallel Machines [†]

L. D. Fosdick
C. J. C. Schauble
K. M. Olender [‡]

CU-CS-420-89 January, 1989

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, CO 80309-0430

[†] This work was supported in part by a grant from IBM.

[‡] Current address: Computer Science Dept., Colorado State University, Fort Collins, CO 80523.

ABSTRACT

Benchmarking runs were made on a group of programs furnished by IBM in both *Fortran* and *Ada* to compare sequential execution times on the Vax 11/780 against parallel executions on both the Encore Multimax and the Alliant FX/8. Timings were made with the original versions of the programs and with versions rewritten to take advantage of the parallelism and vectorization possible on the machines. In some cases, the *Force* language was used for the improved versions of *Fortran* programs.

The *Force* language allowed the parallel versions of the *Fortran* programs to be written quickly and easily. Rewriting *Fortran* code improved the speedups of the programs in many cases. Further revisions concerned with efficient paging also showed an effect. Depending upon the amount of revisions and the mode of execution, speedups ranged from 1.12 to 67.1.

1. INTRODUCTION

1.1. Statement of Project

At the request of IBM, a group of programs written in both *Fortran* and *Ada* were run on different machines to compare execution times of sequential, vector, and parallel machines. The Vax 11/780 was used as a sequential machine to provide a base for both the *Fortran* and *Ada* programs. The Alliant FX/Series machines which include vector operations in their parallel, shared-memory computers were used with both *Fortran* and *Ada* programs to show the speedups currently available under such a system. The Encore Multimax, a twenty-processor, shared-memory machine was used as an additional parallel example for the *Fortran* programs.

The resultant speedups were only about 1.63 to 2.33 for the *Ada* codes on the Alliant FX/8. The *Fortran* programs run on the Alliant were timed using scalar, vector-only, concurrent-only, and concurrent with vector processing modes; the speedups between the first and last modes varied from 1.12 to 3.18 on the original programs. Some of the *Fortran* programs rewritten to take advantage of the machine architecture provided speedups of 6.82 to 67.1 on the Alliant. Other *Fortran* programs were rewritten using the *Force*, a parallel extension of *Fortran* which is available for both the Encore and the Alliant (see [Jor87] for further information). With some additional but simple modifications to the original code, the *Force* versions of the programs had speedups ranging from 8.16 to 10.1 on the Alliant and from 6.31 to 16.7 using sixteen of the twenty processors on the Encore. Later optimizations to reduce paging resulted in Encore *Force* programs with speedups of up to 23.5 using sixteen processes.

2. EXECUTION OF PROJECT

The main work on this project was done between October, 1987 and December, 1987. The work was shared between the Academic Computing Services, which houses the Alliant FX/8 computer, and the Computer Science Department of the University of Colorado at Boulder.

2.1. Machines Used

Three different types of computers were used. Two were parallel multiprocessors, one of which had vector processing capability, and the third machine was sequential.

2.1.1. The Alliant FX/8

The Alliant FX/8 at the University of Colorado at Boulder handled all Alliant *Fortran* and *Force* runs. The Boulder Alliant is a shared-memory multiprocessor with eight computational elements (CE's) or processors, each of which has vector processing capability for vectors of length 32. The machine also has five interactive processors (IP's) to handle I/O. All the CE's are connected to the same cache. The Concentrix operating system, which conforms to Unix externally, runs on all the processors [All87b].

Because the University of Colorado Alliant does not have an *Ada* compiler for the Alliant FX/8, Alliant Computer Systems Corporation provided their computer in Littleton, MA for the *Ada* testing.

2.1.2. The Vax 11/780

To better judge the performance of both the *Ada* and the *Fortran* programs, they were also run on a sequential machine. The computer chosen for this was a Vax 11/780 belonging to the Computer Science Department of the University of Colorado at Boulder.

2.1.3. The Encore Multimax

For supplemental testing of the *Fortran* programs on a different parallel machine, the Encore Multimax belonging to the Computer Science Department of the University of Colorado at Boulder was used. This parallel computer is a shared-memory, MIMD (Multi-Instruction, Multi-Data) machine with twenty 32-bit processors, each of which can supply up to 0.75 MIPS. Each processor shares a 32K byte cache with one other processor and is connected to the main shared memory and to the I/O interfaces by a wide high-speed bus, called the Nanobus. The operating system is a version of Unix, called UMAX 4.2, which allows multi-threading [Enc87].

2.2. Languages Used

The programs provided by IBM were written in *Fortran* and in *Ada*. The rewritten parallel versions were also in *Fortran* and in *Ada*; in addition, some *Fortran* programs were rewritten in the *Force.*

2.2.1. Fortran

To compile and run the *Fortran* programs in the experiment, three different *Fortran* compilers were involved. For the sequential runs done on the Vax 11/780, the Berkeley Unix *f77 Fortran* compiler was invoked. The Encore UMAX *f77 Fortran* compiler handled programs for the Encore Multimax, and the *FX/Fortran* compiler was used for the *Fortran* programs run on the Alliant FX/8 [All87b]. Hence, the comparisons being made in this report are among different *Fortran* compilers as well as different machines.

2.2.2. Ada

With very little change, the *Ada* programs were compiled and run using the *FX/Ada* compiler on the Alliant [All87a]. The Verdix VADS *Ada* compiler, Version 5.41, handled the *Ada* programs run sequentially on the Vax [VER87]. The two compilers have basically the same front end; the main differences are in the code generation [Pom87].

2.2.3. The Force

The *Force* is a *Fortran* extension which allows parallel constructs such as *DOACROSS* loops and critical sections. Parallelism is inherent in the *Force* since all the requested processes are started at the beginning of the execution of the program and run to the end, executing essentially the same code. *DOACROSS* loops may be *pre*-scheduled, meaning that the various iterations are assigned to particular processes at compile-time, or *self*-scheduled, meaning that the iterations are assigned at run-time as the processes become available. The processes may be synchronized by a *Barrier* construct which forces all the processes to wait for the others to reach the same point before proceeding, possibly with some sequential code. Variables may be global to the whole program or local to a single process; they may also be considered *critical* variables. For further information on the *Force*, see [JBA87] or [Jor87].

The *Force* preprocessor is available on both the UCB Encore Multimax and the UCB Alliant FX/8. In both cases, the preprocessor produces *Fortran* code with additional parallel constructs appropriate for the particular machine on which it is run. This code is then processed by the *Fortran* compiler for that machine: the Encore UMAX *f77 Fortran* compiler or the Alliant *FX/Fortran* compiler.

2.3. How Results were Obtained

The actual timings were of one or two subroutine calls in each program. These calls were enclosed in a loop which was executed twenty times. The user time was recorded before and after each call. The printed output included the time for each call, as well as the average time over twenty calls.

Each program was executed at least three times in a single-user environment on each machine. The lowest of the three or more average times produced by these runs is the result given for this experiment.

3. FORTRAN BENCHMARKS

3.1. Original Programs

The original programs were provided by Dr. T. J. Smith of IBM of Boulder. Most of the programs required changes to meet the requirements of the tests. As the programs were to be run over a range of data values, the DATA statements which previously defined the input data were replaced by READ statements. Some comments were also added. In a few cases, the array dimensions had to be increased to accommodate the new data sizes. User time was used in computing the timed runs; so, the programs actually used for the base runs were somewhat different from the original programs as delivered.

For the Alliant, it was necessary to include additional timing loop iterations. The timings for the first few times through a loop were always more than those for the middle iterations, and the last pass through the loop tended to be out of line as well. Hence, the number of repetitions was increased from twenty to twenty-five or forty, with the timings used coming from the middle iterations. We suspect that the extra time used by these iterations are related to paging and caching on the Alliant.

3.2. Methods Used to Improve Performance

Appendix A has a list of hints for parallelizing or vectorizing programs. This list includes techniques used for this project.

Other improvements were made by simply rewriting bad code or eliminating unnecessary and untimed code. For most programs run on the Alliant, few changes were made; most of the parallelization and vectorization was performed by the compiler itself. The only program with significant alterations was the stereo line-of-site code, `loscode1`, and the majority of those changes were to make it run correctly.

3.2.1. Rewriting a Fortran Program for the Alliant/FX

The alterations on the original programs for the base timing runs included simple changes such as changing variable names (some programs used names over 15 characters long) and altering or adding WRITE statements to correctly identify the output. Of course, the modifications also included those mentioned above: changing DATA statements to READ statements, adjusting array sizes to match the input data, and altering the program to collect user time. Since not all the programs computed out the average iteration time which was needed for this experiment, this calculation and output were added to the programs, when necessary. Additional alterations included adding comments or rewriting the code to make it clearer, the consolidation of independent DO loops with similar limits, and switching the inner and outer limits of DO loops, either to improve vectorization or to allow a better distribution of the loops among the concurrent processors.

The rewritten programs also made some use of the vector processors and concurrency available on the Alliant. This involved replacing some DO loops with vector notation, e.g.,

```
  DO 5, I1=1,NUMSEN,1
  DO 6, I2=1,NUMOBJ,1
  DO 7, I3=1,NUMSEN,1
  DO 8, I4=1,NUMOBJ,1
    PAIRS(I1,I2,I3,I4) = .FALSE.
8 CONTINUE
7 CONTINUE
6 CONTINUE
5 CONTINUE
```

becomes the following:

```
PAIRS(1:NUMSEN, 1:NUMOBJ, 1:NUMSEN, 1:NUMOBJ) = .FALSE.
```

The Alliant *FX/Fortran* compiler will produce code to execute this statement as efficiently as possible. The original code, from the Stereo Line-of-Site program, loscode1, initializes the array PAIRS such that the rightmost subscript varies first. Since arrays in *Fortran* are stored in column-major order, this is the worst way available. The altered version permits the compiler to decide the actual order of execution. Further, when compiling in concurrent or vector mode, the generated code will attempt to use the vector processors and concurrency whenever possible.

3.2.2. Rewriting a Fortran Program into a Force Program

In Appendix C are four versions of one of the IBM programs as used on the Encore Multimax. The first is the version used for the base timing runs. Note the DATA statements are replaced by READ statements. The second version is the *Force* program. The *Force* commands are all in lower case with the first character in upper case. Note how the DO loops spread their work among the processors using the *Force* construct Presched DO. For further improvement of performance, the bodies of some subroutine DO loops are consolidated into a single loop. Actions that must be done in sequential mode, like reading input values, are within a Barrier. The third version shows the output of the *Force* preprocessor. Here the *Force* commands have been replaced with the appropriate Encore Multimax parallel processing *Fortran* primitives. The fourth and final version of `att_xfrm`, in Appendix C.4 is a *Force* program to be discussed later.

Appendix B contains a list of the changes required to rewrite a *Fortran* program as a *Force* program for the Encore Multimax, step by step. This shows the actual procedure used to parallelize and improve the performance for the `att_xfrm` program as given in Appendix C.

3.3. Execution of Fortran Programs

Tables 1 and 2 show the size in lines of the eight *Fortran* programs in their various forms. Tables 3, 4, and 5 compare the size in bytes of the executable object file of each of these programs.

Two columns of Tables 1 and 2 contrast the number of lines in the *Force* program (the actual *Force* code) against the number of lines in the *Fortran* program produced by the *Force* preprocessor. The *Force* output includes all the translations of the *Force* commands to the more primitive *Fortran* commands of the parallel constructs given for the Encore Multimax. Note that the *Force* programs are a little larger than the original

Table 1: Size in Fortran Lines of Alliant FX/8 and Vax 11/780 Programs

Name of Program	Original Program	Alliant Program Used	Rewritten Program	Force Program	Force Output	Program on Vax 11/780
`att_xfrm`	67	71	74			79
`intcode1`	71	83	84	99	343	83
`intcode2`	84	96	94	119	399	96
`kalman2`	876	878	824			878
`loscode1`	282	286	259			288
`lsbm1`	91	89	90			98
`stprp1`	120	121	103			124
`stprp2`	153	153	140			158

Table 2: Size in Fortran Lines of Encore Multimax Programs

Name of Program	Original Program	Program Used	Rewritten Program	Force Program	Force Output
`att_xfrm`	67	79		94	304
`intcode1`	71	83		98	324
`intcode2`	84	96		108	343
`kalman2`	876	878	867	564	1212
`loscode1`	282	288	249	240	602
`lsbm1`	91	98		158	431
`stprp1`	120	124		148	377
`stprp2`	153	158		212	442

Fortran programs while the preprocessed programs are much larger. This difference suggests the simplicity of programming in the *Force*, as compared to using the actual library routines given by the machine manufacturer.

The *Fortran* programs were compiled and executed in four different ways on the Alliant FX/8:

(1) scalar sequential, which allowed no vector processing and was confined to run in a single processor;

(2) vector processing on a single processor or CE,

(3) concurrent execution on all eight processors with no vector processing and

(4) a combination of full concurrency with vector processing in all eight processors.

This last mode, as expected, showed the most improved time. The *Force* programs were all compiled and run in the last mode. Since the vector length for the vector processors is 32, the maximum linear speedup has an upper bound of 8 *x* 32 or 256.

Table 3: Alliant FX/8 and Vax 11/780 Original Fortran Object Files Size in Bytes

Name of Program	Seq Scal Object	Vector Object	Con-current Object	Con Vect Object	Vax 11/780 Object
att_xfrm	16384				38912
intcode1	16384	16384	16384	16384	38912
intcode2	16384	16384	16384	16384	39936
kalman2	32768	28672	24576	28672	57344
loscode1	16378				21653
lsbm1	16384				45056
stprp1	16384				46080
stprp2	16384				46080

Table 4: Size in Bytes of Alliant FX/8 Rewritten Fortran Object Files

Name of Program	Seq Scal Object	Vector Object	Concurrent Object	Con Vect Object	Force Object
att_xfrm		16384	16384	16384	
intcode1	20480	20480	16384	20480	20480
intcode2	20480	20480	16384	20480	20480
kalman2	24576	32768	28672	32768	
loscode1		19928	12906	15696	
lsbm1		20480	20480	20480	
stprp1		16384	16384	16384	
stprp2		16384	16384	16384	

Table 5: Size in Bytes of Encore Multimax Fortran Object Files

Name of Program	Base Object	Rewritten Object	Force Object
att_xfrm	56632		73344
intcode1	57986		73494
intcode2	57986		73494
kalman2	72687	72687	84154
loscode1	63168	62144	77460
lsbm1	71594		79798
stprp1	70790		78794
stprp2	70810		79054

It is interesting to note that the *Force* object files are not as large as one might expect from the number of source lines for the preprocessed programs. Most of the code inserted by the *Force* preprocessor consists of comments or primitive commands which do not generate much object code.

3.4. Fortran Results

Results of runs made on the various machines are given in Appendices D.2 and D.3. Runs on the Encore Multimax were done on one, two, four, eight, and sixteen of the twenty available processors. Concurrent execution of the Alliant FX/8 implied all eight processors were being utilized.

Many of the programs have a computational complexity of $O(n)$, where n is the size of the input. These programs are `att_xfrm`, `intcode1`, `intcode2`, `stprp1`, and `stprp2`. The Kalman filter code, `kalman2`, is of complexity $O(n^3)$. The subroutine being timed in the least squares batch program, `lsbm1`, has complexity $O(nmp^2)$, where n is the number of rows, m is the number of columns, and p is the number of points being used. The stereo line-of-site program, `loscode1`, has complexity $O(n^2 s^2)$, where n is the number of objects and s is the number of sensors given as input.

3.4.1. Original Programs

The original programs, modified as described above, were used to provide the base times on each machine. In Appendix D these times are marked as *Original* or *Orig.*

3.4.2. Rewritten Programs

The results of timing the execution of the rewritten programs against the base runs of the original programs are given in Appendix D. These times are marked as *Rewritten* or *Rewrtn.* While the *Force* was used for most of the *Fortran* programs run in parallel on the Encore Multimax, the majority of the parallel runs on the Alliant FX/8 were done with rewritten programs.

The speedups shown for the rewritten programs for the Alliant over the base versions of the programs run in scalar sequential mode ranged from 2.13 to 4.11. When the base programs were run in the full concurrent/vector mode, the speedups ranged from

2.82 to 21.1. Comparing the sequential scalar runs of the base programs against the concurrent/vector runs of the rewritten versions, the speedups ranged from 5.31 to 67.1. Note that these speedups are all far less than the upper bound of 256 for a linear speedup of a machine with eight processors, each equipped with vector processors of length 32.

Code rewritten for the Encore Multimax showed little speedup, up to 1.05, since it was not parallelized and was only run on a single processor. Inclusion of the Encore Multimax *Fortran* parallel library calls would have allowed the programs to be executed in parallel, but this was not done due to insufficient time.

3.4.3. Force Programs

Most of the *Force* program revisions were done on the Encore, as that is the easiest way to alter *Fortran* programs to run in parallel on the Multimax. Use of the *Force* avoids the use of primitive commands for parallelization, which varies by manufacturer. A few Alliant *Force* programs were also written. These provide a comparison against both the Encore *Force* speedup and the concurrent vector mode of the Alliant *FX/Fortran* compiler.

Graphs are provided in Appendices D.4 and D.5, which show the timed results of the Encore *Fortran* and *Force* programs. The keys denote *Fortran* as *Ftn* and *Force* as *Frc-n* where *n* is the number of processors used for the given run.

The speedups on the Encore for *Force* programs versus the original base programs ranged from 6.31 to 16.7, using a maximum of sixteen of the twenty available processors. Some programs were more easily redesigned into a parallel algorithm than others. For instance, modifying a loop executed many times in the `intcode2` code, by spreading the work over all the processes, showed considerable speedup. On the other hand, the `stprp1` code had a large loop limit of eight; spreading it over more than eight processors showed no improvement. In this case, the best times were shown when it ran with eight processors, rather than sixteen, with a speedup of 8.10.

The *Force* programs included some of the alterations in the rewritten programs as well, which explains the better than linear speedup. During the rewriting process, some loops were combined or reordered. This reduced loop overhead since there were fewer loops; it also improved the efficiency of the executing loops as each processor had a fair amount of work to do on each iteration before getting the data for the next. Further, interchanging the loops so that a large *Fortran* array will be accessed by columns instead of by rows can improve the execution time of the program, as less paging will be required. A sequential run of the revised programs without the *Force* directives would have shown some improvement as well.

The *Force* programs to be tested on the Alliant showed a speedup of up to 10.1 over the base code run sequentially, using all eight processors. This same program showed a 8.81 speedup over the rewritten sequential version. The speedups over the base and rewritten versions running in concurrent/vector mode are only 2.47 and 1.24. This suggests that simply recasting a *Fortran* program into a *Force* program can produce a parallel version as good as a hand-rewritten program.

Again, the *Force* proved to be an easy way to rewrite a sequential *Fortran* program into a parallel *Fortran* program. These experiments have shown that the *Force* programs have speedups comparable to (if not better than) the *Fortran* programs which were simply rewritten using the manufacturer's parallel *Fortran* constructs. Since the *Force* is available on more than one parallel computer, unlike most manufacturer's parallel constructs, the results of these experiments strongly recommend the *Force* as a medium for converting sequential to parallel *Fortran* programs.

Along this line, however, it should be noted that the Alliant *FX/Fortran* compiler includes many automatic parallelization techniques. This makes it fairly easy to use for the creation of parallel programs as well as for the conversion of sequential to parallel programs. An Alliant parallel version of a sequential *Fortran* program can sometimes be accomplished by a single computation with the correct optimizations. Otherwise, the insertion of a few compiler directives into the code itself and/or some simple loop revisions may be all that is required to make the parallel version fairly efficient. Because the *Force* on the Alliant makes use of all the optimization available, full concurrency and vectorization, there were occasional problems with Alliant *Force* programs being over-parallelized; that is, the Alliant compiler would attempt to parallelize a loop already taken care of by the *Force*. Hence, using the *Force* on the Alliant required a little more care than on the Encore Multimax.

4. IMPROVED FORTRAN BENCHMARKS

Although the benchmarking tests were completed in December, 1987, a few programs were reconsidered for further optimization during the summer of 1988. These optimizations were concerned with efficient memory management while the earlier rewriting attempts had been more interested in making a program spread its work in parallel as much as possible. While doing the work of a program in parallel will certainly speed up the execution time (as can be seen by the results in Appendix D), more careful consideration of the use of memory in order to avoid excess paging should also help.

It should be noted that the original *Fortran* programs provided by IBM were designed for a vector machine like the IBM 3090. Vector processors can handle large groups of data efficiently, but do so quite differently than parallel multiprocessors. In particular, the order of the data which is best for a vector processor may result in extraneous variables and be a poor arrangement for a parallel processor like the Encore Multimax. Since the Alliant FX/8 includes vector processing elements in its CE's, these new optimizations were done only for the Encore programs.

4.1. Further Optimizations

The actual optimizations done involve redeclaring the arrays and interchanging nested DO loops so that all the arrays are accessed in column order so that the memory locations accessed by the program are contiguous whenever possible. In turn, this means that the information in each page allocated to the array is fully used before the next page is brought into the cache, and paging is minimized.

The bar graphs in Figures 1, 2, and 3 show the difference in speedup between the two *Force* versions. The label, **Force**, implies the parallel version used to provide the benchmarking data discussed earlier; **OptForce** refers to the later *Force* version, which was rewritten from the first *Force* program in an attempt to optimize memory use. It is clear from these graphs that proper memory management can make a difference in the

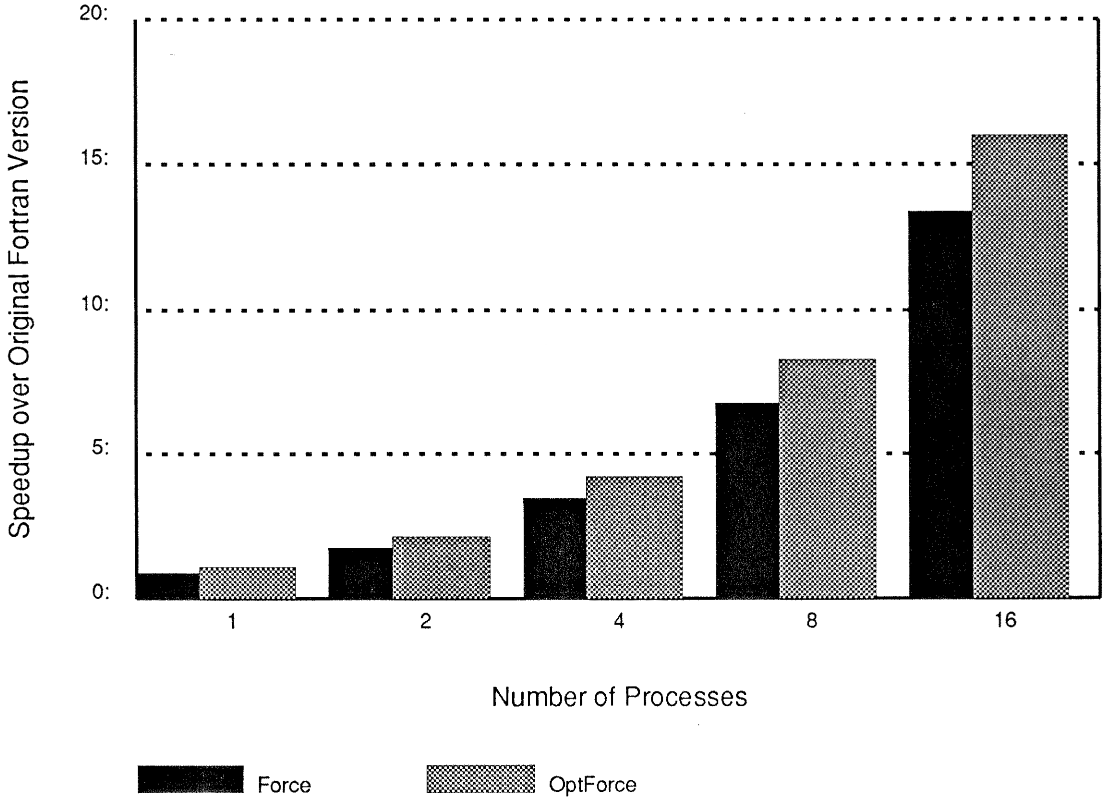


Figure 1: Speedup Shown by Both Force Versions of `att_xfrm` for n=50,000

execution time of parallel programs.

A change was made to the attitude transformation program, `att_xfrm`, to ease memory use. Because the program had been designed for use by a vector machine, the main loop of the subroutine being timed used several temporary arrays to store partial results of the vector operations. In a vector machine, the *Fortran* compiler would probably optimize these temporary arrays into vector registers. However, this is not the case on a machine like the Encore Multimax. Thus, the temporary arrays were removed, and the body of the loop was rewritten to include a single assignment statement. This last *Force* version of the program follows the other versions of `att_xfrm` in Appendix C.4.

4.2. Optimized Storage Results

Results of these optimizations are given in Appendix E for a subset of the *Fortran* programs, namely `att_xfrm`, `intcode1`, and `intcode2`. They are compared with reruns of the other versions.[1] Figures 1 through 3 also show the speedups obtained by

[1] The programs all needed to be recompiled and run, as a new *Fortran* compiler was installed on the Encore Multimax during the Spring of 1988 which generated much faster code. Thus the execution times given in these tables will be smaller than those in the tables in Appendix D.3; however, the speedups for the unchanged *Force* versions over the original programs are still from 13.4% to 16.1% for 16 processes.

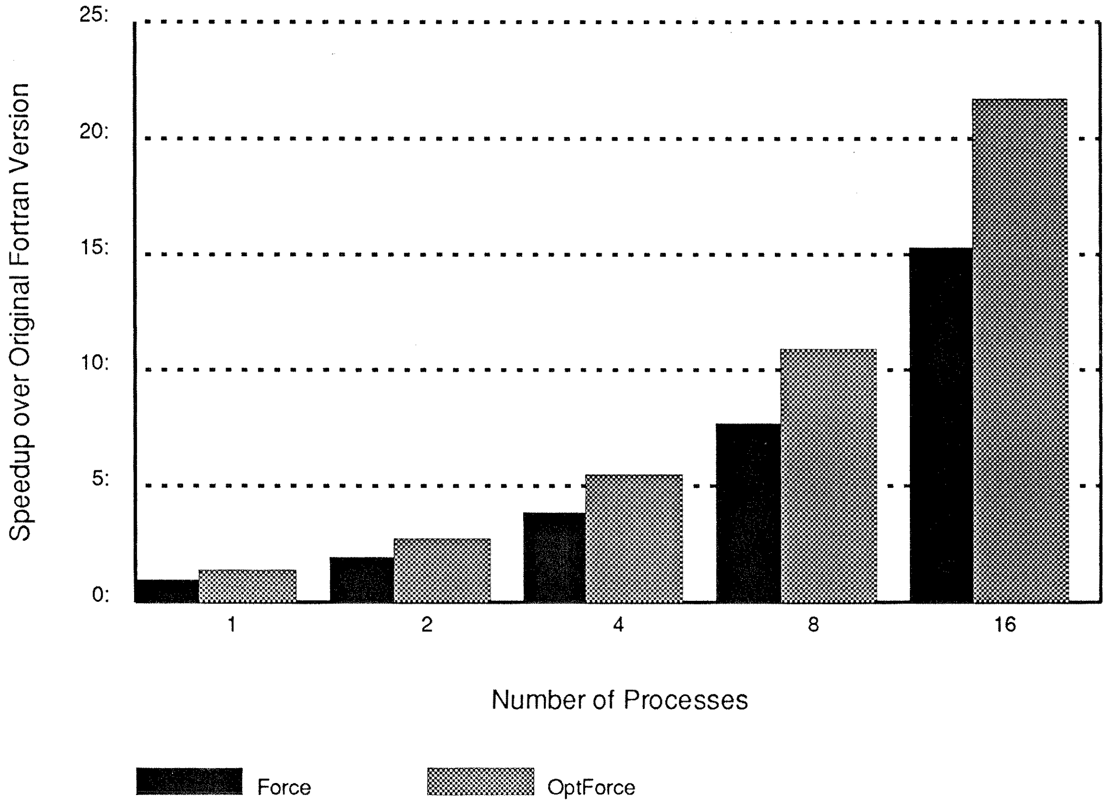


Figure 2: Speedup Shown by Both Force Versions of `intcode1` for n=50,000

this optimization for the three different programs. The new optimized *Force* programs now show speedups from 16.0% to 23.5% using 16 processes, while the single process runs were all faster than the sequential *Fortran* runs (1.07% to 1.49%). The three programs involved all had computational complexity of $O(n)$, as can be easily seen from the tables of results.

5. ADA BENCHMARKS

A second series of benchmarks was performed comparing the performance of sequential *Ada* versions of the following programs on a Vax 11/780 to the Alliant: `xfrm`, `intcode1`, `intcode2`, `stprp1`, `stprp2`, `kalman1`, `kalman2`, `lsbm`, and `stereo`. These *Ada* versions were provided by IBM. In addition, a program with three simple tasks, also provided by IBM, was instrumented to determine the performance overhead associated with the *Ada* tasking mechanism on the Alliant. The clock accuracy on the Vax was insufficient to gather meaningful data for the tasking program. The compilers used were Verdix VADS, Version 5.41, on the Vax and the Alliant *FX/Ada*, which uses essentially the Verdix front end with an Alliant code generator [Pom87].

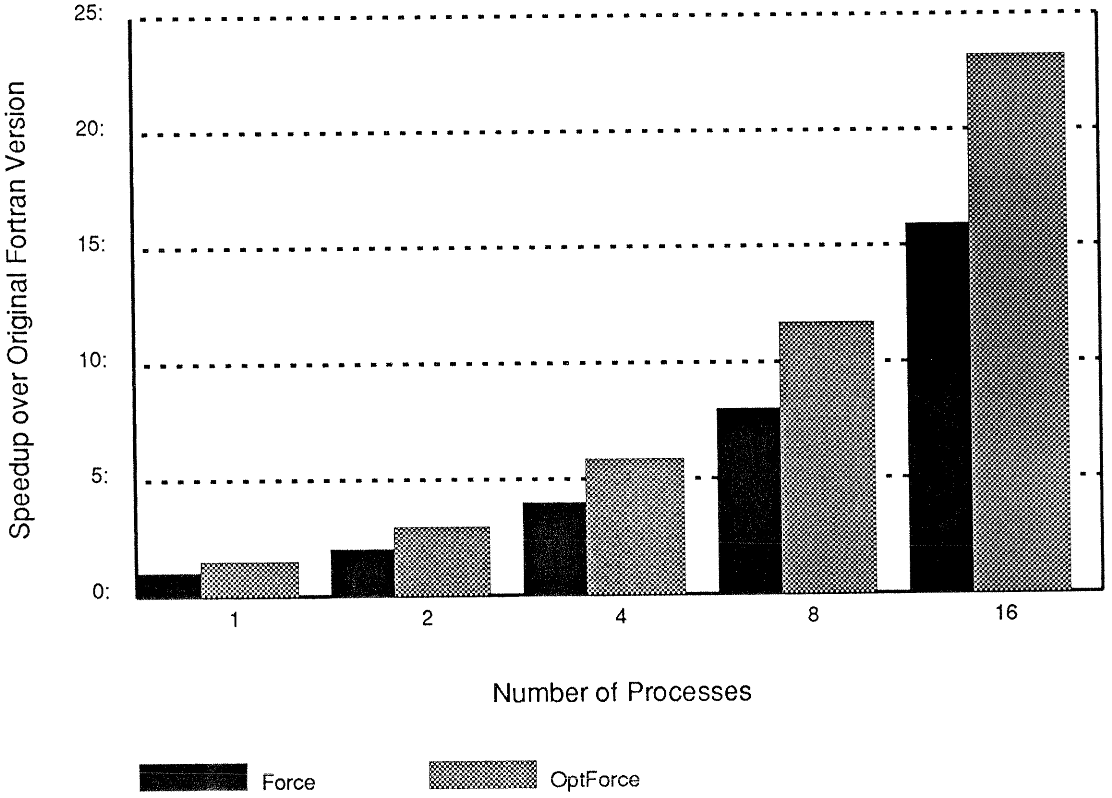


Figure 3: Speedup Shown by Both Force Versions of `intcode2` for n=50,000

5.1. Ada Benchmark Results

Tables in Appendix D.1 summarize the results obtained from the sequential *Ada* programs. Overall the Alliant executed the *Ada* versions of the benchmark programs about twice as fast as the Vax 11/780. This is the same speedup as between the strictly sequential *Fortran* versions on both machines, and so can be attributed to the faster scalar processor of the Alliant.

Also note that *Ada* timings on both machines were considerably slower than the sequential *Fortran* results, a factor of 8 on the Vax and 20 on the Alliant. Much of the blame must go to poor code generation and the run-time constraint checking requirements of the *Ada* language, although some of the difference between the Alliant and Vax slowdowns may be due the inability of the then-current Alliant code generator to take advantage of the vector and parallel hardware.

For a simple vector sum loop, *Ada* and *Fortran* versions of which are given in Figure 4, the size of the object codes generated for the Vax were drastically different. Although the object code was unavailable for the Alliant versions, we speculate that the problems are similar.

The *f77 Fortran* compiler on the Vax generates 3 instructions for the assignment statement in the loop with no optimization. The Verdix *Ada* compiler generates 45 instructions. Much of this code performs the bounds checking required by *Ada*. Under

```
    DO 10, I=1,N
      R(I) = A(I) + B(I)
10  CONTINUE
```

Figure 4a: A simple vector sum loop in *Fortran*

```
for I in 1..N loop
    R(I) := A(I) + B(I);
end loop;
```

Figure 4b: A simple vector sum loop in *Ada*

```
addl3 v.2-v.1+-4(r11)[r10],v.3-v.1+-4(r11)[r10],v.12-v.1+-4(r11)[r10]
```

Figure 5a: Optimized *Fortran* assembly code for R(I) = A(I) + B(I)

```
movl     -28(fp),r2
movl     -74(fp),r0
movl     -74(fp),r3
movl     -74(fp),r4
subl3    =1,r0,r0
mull3    =4,r0,r0
addl3    -28(fp),r0,r0
movl     (r0),r1
movl     -40(fp),r2
subl3    =1,r3,r0
mull3    =4,r0,r0
addl3    -40(fp),r0,r0
movl     (r0),r0
addl3    r1,r0,r1
subl3    =1,r4,r0
mull3    =4,r0,r0
addl3    -58(fp),r0,r0
movl     r1,(r0)
```

Figure 5b: Optimized *Ada* assembly code for R(I) := A(I) + B(I);

optimization, *f77* reduced those three statements to one by taking advantage of the full range of addressing modes available on the Vax. The *Ada* compiler, with full optimization and all constraint checking suppressed, still generates 18 instructions using strictly register operations. The fully optimized assembly codes generated by the two compilers are given in Figures 5a and 5b.

Although the *Ada* code generated appears to be eighteen times as long as the generated *Fortran* code, the *Ada* instructions use only register operations, which are faster. Hence, the *Fortran* code for this simple loop only runs about four times faster overall than the *Ada* code, even when both are optimized fully (a factor of 4.1 between the unoptimized versions and 3.6 between the optimized versions). The loop overhead per iteration was approximately the same, 5 instructions, in both languages. The *Fortran* code experienced a speedup of 1.43 due to optimization, while the *Ada* code did slightly better at 1.64, in part due to the elimination of the bounds checks as well as the more conventional optimizations performed. The following table gives the times to execute 1000 iterations of the given loop on the Vax.

Code	Time(sec)
Optimized *Fortran*	0.58
Unoptimized *Fortran*	0.83
Optimized *Ada*	2.12
Unoptimized *Ada*	3.47

It is not hard to see that even without the bounds checking, the code generation for *Ada* is not good. This is not necessarily an attribute of the *Ada* language. Several *Ada* compiler vendors claim generated code quality as good or better than *C* or *Pascal* compilers for the most recent versions of their compilers being distributed as measured by the Whetstone and Dhrystone benchmarks.

Since the Alliant *Ada* compiler did not utilize the machine's parallel capabilities, no effort was made to restructure the *Ada* programs to permit fuller use of the facilities as was done with *Fortran.*

We should note that a new version of the Alliant *FX/Ada* compiler has been distributed since these measurements were taken. This new version improved the code generation phase to take advantage of the parallel and vector hardware features. It is not known whether it addressed any of the other code generation problems.

5.2. Tasking overhead measurements

The results of the tasking overhead measurements are given in Tables 6 and 7. The Alliant did permit the assignment of *Ada* tasks to processors, the only use of the Alliant parallel hardware it made at the time. However, none of the data measured was of a nature that the number of processors assigned would be a major factor and this was borne out by the results, which were relatively consistent between the one and four computational element cases.

Note that these results are only for the Alliant. The clock on the Vax was insufficiently accurate (only about 1/60 seconds) for conclusive timings.

The overhead to execute an Ada *delay* statement can be broken down into two parts, delay computation and blocking. The delay computation is the time to compute the amount of delay requested and to decide if blocking is necessary; blocking is unnecessary, for example, when the expression giving the amount of time to delay is zero or negative. The blocking time is the overhead incurred to suspend and reinstate the delayed task. The delay computation time was about 350 microseconds. Blocking time was about 2.2 milliseconds. As expected, the data showed no correlation between the amount of time actually delayed and the overhead incurred other than positivity of the requested delay.

Another important task-related overhead is task swap time, the amount of time from completion of a task entry until the calling task's context is restored. The values measured were very consistently about 500 microseconds with small deviation.

Inter-task data exchange is the amount of time taken by a task whose sole purpose is to exchange protected data between tasks. It is quite consistently about the same value. This shows that the dominant factor in task invocation is context switching and not data transfer.

Rendezvous with no parameters is the amount of time between calling a task entry that has no parameters and the actual completion of the rendezvous with the called task. Rendezvous with one parameter is the equivalent time when the entry has a single parameter. The values here varied more widely, but as expected, the parameterized rendezvous takes slightly longer. All times are in microseconds.

Table 6: Tasking overhead data for one CE

Value	High	Low	Average	Std. Dev.
0 Delay	400	360	370	±100
+ Delay	4,960	1,390	2,580	±630
Task swap	580	480	510	±16
Inter-task data exchange	650	550	588	±19
Rendezvous with no parameters	310	260	290	±14
Rendezvous with one parameter	720	320	360	±38

Table 7: Tasking overhead data for four CE's

Value	High	Low	Average	Std. Dev.
0 Delay	380	320	340	±140
+ Delay	5,700	1,230	2,490	±630
Task swap	580	470	496	19
Inter-task data exchange	650	530	576	±23
Rendezvous with no parameters	310	240	274	±17
Rendezvous with one parameter	380	300	336	±17

6. SUMMARY

The purpose of this research was to see how well the given programs could perform on the parallel machines, and in particular, on the Alliant FX/8. In hindsight, of course, many other optimizations now appear possible which might have improved the speedups even more. However, during the limited time given to the experiments, much was accomplished.

The main results of this work are as follows:

(1) As expected, the project showed that the execution of many *Fortran* programs could be improved on a parallel machine with vector processing, such as the Alliant FX/8. Even using a parallel multiprocessor can reduce processing time. With appropriate rewriting of the code, the improvements on some could be more than linear. However, this is not the case for all the programs, as the timings for

the `stprp1` code on the Encore indicate.

In particular, rewriting *Fortran* code improves the speedup by a factor of at least 1.05 on the Encore Multimax and 2.13 on the Alliant FX/8 in all cases, and by a factor of four to eight in many cases.

(2) The *Force* is shown to be a convenient, portable language for rewriting sequential *Fortran* programs into parallel *Fortran* programs. Execution of the programs rewritten in the *Force* have speedups comparable with (or better than) the programs that were rewritten to make use of the available parallel constructs. Use of the *Force* combined with some revised code can result in a better than linear speedup of the original code.

(3) When additional revisions were made to the *Force* programs for the Encore Multimax in an attempt to reduce paging faults and caching, the improvements of speedups ranged from a factor of 1.19 to 1.47. The importance of efficient paging for parallel machines with shared memory must be stressed. Here machines must compete for a single resource, the shared memory; the more efficient the use of this resource, the better the speedup obtained by the parallel execution.

(4) The state of the Alliant *FX/Ada* compiler at the time of these tests did not allow for parallel code generation. Without any concurrency or use of the vector operations available, the Alliant *Ada* over the Vax *Ada* showed only the speedup expected from the different types of processors. Hence, the possible speedup for concurrent *Ada* cannot be derived from this experiment.

What this project did show was that the *Ada* compiler performance on Alliant appeared poor compared to that of the *Fortran* compiler. Clearly, the code generation could use some improvement. It would be interesting to test these codes on the new compiler.

6.1. Further Work

Now that the Alliant Computer Systems Corporation has a new *Ada* compiler which incorporates the parallel capabilities of the machine, it would be interesting to rerun the experiments using the new compiler. Is the parallelism now provided by the compiler able to provide a reasonable (almost linear) speedup? Has the code generation for the new compiler significantly improved over the previous version? Unfortunately, we do not have an *Ada* compiler for the UCB Alliant FX/8. If it were to be made available in the future, such experiments could be done.

The performance of loops on the Alliant FX/8 is worth further investigation. While paging faults and cache misses can easily account for extra time during the first iteration of a loop, the second, third, and last iterations also seem to require extra time.

It would also be interesting to attempt to optimize the other Encore *Fortran* programs with respect to memory management to see if the speedup of these programs would improve in the same manner. Would carrying this work to the Alliant which has a much different hardware configuration also have an effect? Could something similar be done to the *Ada* programs?

6.2. Acknowledgements

We would like to acknowledge Dr. Howard Pomerantz of Alliant Computer Systems, Inc. for his assistance in locating an Alliant *FX/Ada* manual and in allowing us to use the Alliant FX/8 in Littleton, Massachusetts to run the *Ada* tests.

Most of the *Fortran* timing runs on the Alliant FX/8 were done by Donna Pattee of the Administrative Computing Center of the University. Both she and Dave Wood of ACS should be recognized for their part in the project.

We also would like to thank Dr. T. J. Smith of IBM for his support and suggestions.

And, of course, thanks goes to Muhammed Benten for his counsel on both the Encore and the Alliant, as well as the *Force*.

7. BIBLIOGRAPHY

[All87a] Alliant Computer Systems Corporation, ''FX/ADA User's Guide'', Version 1.1, Alliant Computer System Corporation, Littleton, MA, Aug. 1987.

[All87b] Alliant Computer Systems Corporation, ''FX/FORTRAN Programmer's Handbook'', Version 3.0, Alliant Computer Systems Corporation, Littleton, MA, Mar. 1987.

[Ame83] American National Standards Institute, ''Ada Language Reference Manual '', American National Standards Institute/MIL-STD-1815A, U. S. Government, Ada Joint Program Office, 1983.

[Don81] J. J. Dongarra, ''Some Linpack Timings on the CRAY-1'', *Tutorial on Parallel Processing*, 1981, 363-380.

[Enc87] Encore Computer Corporation, *Multimax Technical Summary*, Encore Computer Corporation, Marlboro, MA, 1987.

[FWB85] S. I. Feldman, P. J. Weinberger and J. Berkman, ''A Portable Fortran 77 Compiler'', in *Unix Programmer's Manual Supplementary Documents, Volume 1*, Univ. of California, Berkeley, CA, Sep. 1985.

[Jor87] H. Jordan, ''The Force'', in *The Characteristics of Parallel Algorithms* , L. H. Jamieson, D. B. Gannon and R. J. Douglass (editor), MIT Press, Cambridge, MA, 1987, 395-436.

[JBA87] H. F. Jordan, M. S. Benten, N. S. Arenstorf and A. V. Ramanan, *Force User's Manual, Revised edition*, Department of Electrical and Computer Engineering, University of Colorado, Boulder, CO , June 1987.

[Pom87] H. Pomerantz, *Personal Communication*, Alliant Computer Systems Corporation, Nov. 1987.

[Pon88] C. G. Ponder, ''Benchmark Semantics'', *SIGPLAN Notices 23*, 2 (Feb. 1988), 44-48.

[Syd80] P. J. Sydow, *CRAY-1 Optimization Guide*, Cray Research, Inc., Mendota Heights, Minnesota, 1980.

[VER87] VERDIX Corporation, ''VADS© VERDIX Ada Development System Manual'', Version 5.41 for VAX/UNIX, VERDIX Corporation, Chantilly, VA, 1987.

APPENDIX A

Modifying Fortran Code for Vector/Parallel Speedup on the Alliant/FX

(1) **Avoid loops that include a two-way branch.** Such loops can rarely take advantage of vectorization and may be difficult to parallelize. If one branch will only be taken a small percentage of the time, include it after the main loop has run.

For instance, consider the code to initialize a matrix to zeros except for the diagonal elements which should be set to one. First, set the matrix to all zeros in a doubly-nested loop. After the completion of the inner loop, set the appropriate diagonal element. E.g.,

```
              DO 10  I=1,N
              DO 20  J=1,N
      20        A(I,J) = 0.0
      10        A(I,I) = 1.0
```

In this manner, the inner loop can be done concurrently and with vectorization. The outer loop may be handled concurrently as well. The extra time used for a double assignment to `A(I,I)` is more than made up by the speedup of doing the whole loop concurrently or with vectorization.

(2) For better vectorization, **let innermost loop have the greatest range.** There is always some inherent setup time for vectorization. The larger the vector, the better amortization of this time.

(3) For better vectorization, **the innermost loop should work with the leftmost subscript of a multiply-dimensioned array.** If this conflicts with (3) above, consider altering the declarations of the array to make the largest subscript range leftmost.

(4) **Avoid data dependencies in DO loops.** A data dependency occurs between different iterations of a loop if the later iteration uses the results of the earlier iteration. For instance, consider the following loop:

```
        A(1) = 0.0
        DO 5  I=1,N
      5   A(I+1) = A(I) + 2.5
```

Here each iteration **must** follow the previous one or the value of `A(I)` will be undefined. The iterations of the loop are not independent of each other; they cannot be run in parallel. A better solution would be the following loop.

```
      DO 5  I=1,N
5       A(I) = FLOAT(I-1) * 2.5
```

(5) **Avoid carry-around scalars in DO loops.** Suppose a loop contains a statement like `S = S + A(I)*B(I)/C(I)`. This cannot be executed in parallel as every iteration will be adding a value to the same scalar, `S`. However, there are several methods of dealing with this.

One way is to substitute an array for the scalar, e.g. `SUM(I) = A(I)*B(I)/C(I)`, adding the elements of the array `SUM` after the loop has been exited. This is especially good when there is a special vector operation available (as on the Alliant) which efficiently sums the elements of a vector.

Another way is to use a private (non-shared) variable to gather the partial sum within each process during the loop:

```
PRIVATE PSUM
 ...
PSUM = PSUM + A(I)*B(I)/C(I)
```

When the loop has finished, these partial sums can be added together to form the total sum, perhaps in a critical section.

(6) **Combine DO loops when possible.** This may mean moving some loop statments up in a module. Do not do this if it means adding data dependencies.

There is always some overhead for running a `DO` loop. If the index is used as a subscript, some address computation may be saved by including references to more than one array within the same loop.

(7) **Define arrays with fewer dimensions.** A singly-dimensioned array is easier to vectorize. A single subscript will also allow quicker address computation.

(8) **Avoid subroutine calls in DO loops.** Because of possible side effects, subroutines cannot be called in parallel. It is better to include the actual subroutine code within the loop, when possible, or to put the loop in the subroutine. The Alliant does have a special compiler directive which will allow parallel subroutine calls within a loop, but the user must be sure of the possible side effects before using such a directive. The *Force* has a similar capability.

(9) **Become familiar with special compiler directives and options for concurrency and vectorization** for the machine and compiler you use. For instance, the *-alt* option of the Alliant compiler allows alternate code to be compiled for the case when the loop is long and for when it is short. The code that will actually be executed is decided dynamically when the loop range is known.

(10) **Become familiar with special library routines and language extensions for concurrency and vectorization** for the machine and compiler you use. For instance, the Alliant FX/Fortran supplies *dotproduct* and *sum* routines which handle efficient vector multiplications and the summing of the elements of a vector. It allows arrays to be addressed as vectors, e.g., `A(1:N)` implies `A(1)`, `A(2)`, ..., `A(N)`, and also has a number of matrix operations.

More ideas for efficient Fortran programming with vectorization and concurrency can be found in the Alliant *FX/Fortran Programmer's Handbook* [All87b] and in the *CRAY-1 Optimization Guide* [Syd80].

APPENDIX B

Modifying Fortran Code to Force Code for the Encore Multimax

(1) Change the `PROGRAM` statement to a `Force` statement. The *Force* preprocessor will add its own main program to start the correct number of processes at runtime.

(2) Adjust the declarations to denote which variables are `Shared` or global to all the processes and which are to be `Private` or local to the individual processes. Be sure to include an `End declarations` statement.

(3) Place a `Barrier` and an `End barrier` around each `READ` statement and/or global scalar initialization to assure that these will be be done by only one process. Try to include as much sequential code within a single `Barrier` as possible.

(4) For subroutines that can be run in parallel without conflict, change the `CALL` statement to these routines to `Forcecall` statements. This allows all the processes to call and execute the subroutine in parallel.

(5) Make sure that the timing calls come just before and just after the main subroutine call. It will not be correct to put the timing calls before and after the main loop and then divide the time by the number of loop iterations. The timing is to be of the actual call to the subroutine and should not include the overhead of the loop itself.

(6) Put all the `WRITE` statements and the timing statements within an `IF` statement to assure that they will only be performed by a single process and that that process will always be on the same processor. Without this restriction, chaos could result. Different processors may empty their output buffers at different times, causing the output to be out of order. Also, the clocks of the processors are not exactly in sequence. By having one processor record the beginning time and another the ending time, the execution time may even appear to be negative.

(7) For those subroutines which have been called using `Forcecall` statements, alter the `SUBROUTINE` statement to a `Forcesub` statement.

(8) The declarations for the `Forcesub` routines will also need to be altered as in (2) above. The parameters must remain as is, not marked `Shared` or `Private`. Be sure to include an `End declarations` statement.

(9) All `DO` loops should be examined for possible parallelization. If the body of a loop is independent, that is, if each iteration has no data dependencies with other iterations of the loop, the loop can be changed to a `Presched DO` or a

`Selfsched DO` loop. Doubly nested loops can be handled by `Pre2do` or `Self2do` statements. Be sure to end the loop with the appropriate `End presched do` or `End selfsched do` statement.

(10) It might be possible to break up an unparallelizable loop into two loops, one of which can execute in parallel and the other sequentially. This will allow concurrent execution whenever possible while keeping the synchronization constraints inherent in the program.

(11) See if some small loops can be combined. A larger loop body is often more efficient than a smaller one because of the overhead involved in spreading out the work over the processes.

For more information, see the *Force User's Manual* [JBA87].

APPENDIX C

A Sample Program done in Fortran and in the Force

C.1. Fortran Version of Altitude Transformation Program

```
      PROGRAM ATTXFM

      REAL A(3,3,50000),X(3,50000),Y(3,50000)
      REAL TT,TX,TA(100),TB(100)
      REAL TEMP(2), TMPTIM
      INTEGER LOOP,NLOOP,NUMOBJ
C
C     GET NUMBER OF TIMES TO LOOP AND THE ARRAY DIMENSIONS
C
      DATA NLOOP /21/
C     DATA NUMOBJ /1000/
C
      READ (5,*) NUMOBJ
C
      CALL INIT(A,X,NUMOBJ)
C
C     PERFORM ATTITUDE TRANSFORMATION "NLOOP" TIMES
C
      DO 100,LOOP=1,NLOOP,1
      TMPTIM = etime(TEMP(1), TEMP(2))
      TB(LOOP) = TEMP(1)
      CALL XFRM(A,X,Y,NUMOBJ)
      TMPTIM = etime(TEMP(1), TEMP(2))
      TA(LOOP) = TEMP(1)
100   CONTINUE
C
      TT = 0.0
      DO 150,LOOP=2,NLOOP,1
      TX  = TA(LOOP) - TB(LOOP)
      TT = TT + TX
      WRITE (6,*) 'ITERATION NUMBER: ',LOOP-1,
     1            '   TIME:',TX
150   CONTINUE
C
      WRITE (6,*)
      WRITE (6,*) '***NUMBER OF OBJECTS***', NUMOBJ
      WRITE (6,*)
      WRITE (6,*) '***TOTAL TIME***', TT
      WRITE (6,*) '***AVERAGE TIME***', TT/20.
      WRITE (6,*)
C
```

```
      END
C
      SUBROUTINE XFRM(A,X,Y,NUMOBJ)
      REAL A(3,3,NUMOBJ),X(3,NUMOBJ),Y(3,NUMOBJ)
      REAL TEMP1(50000),TEMP2(50000),TEMP3(50000)
      INTEGER NUMOBJ
      INTEGER AXIS,OBJECT
C
      DO 100,AXIS=1,3,1
      DO 10,OBJECT=1,NUMOBJ,1
      TEMP1(OBJECT)=A(AXIS,1,OBJECT)  * X(1,OBJECT)
10    CONTINUE
      DO 20,OBJECT=1,NUMOBJ,1
      TEMP2(OBJECT)=A(AXIS,2,OBJECT)  * X(2,OBJECT)
20    CONTINUE
      DO 30,OBJECT=1,NUMOBJ,1
      TEMP3(OBJECT)=A(AXIS,3,OBJECT)  * X(3,OBJECT)
30    CONTINUE
      DO 40,OBJECT=1,NUMOBJ,1
      Y(AXIS,OBJECT)=TEMP1(OBJECT) + TEMP2(OBJECT)
     1             + TEMP3(OBJECT)
40    CONTINUE
100   CONTINUE
      END
C
      SUBROUTINE INIT(A,X,NUMOBJ)
      REAL A(3,3,NUMOBJ),X(3,NUMOBJ)
      INTEGER OBJECT,NUMOBJ,ROW,COL
      DO 10,OBJECT=1,NUMOBJ,1
      DO 20,ROW=1,3,1
      X(ROW,OBJECT)=ROW
      DO 30,COL=1,3,1
      A(ROW,COL,OBJECT)=ROW + COL
30    CONTINUE
20    CONTINUE
10    CONTINUE
      RETURN
      END
```

C.2. Force Version of Altitude Transformation Program

```
      Force ATTXFM of NPROCS ident ME
C
        Shared  REAL  A(3,3,50000),X(3,50000),Y(3,50000)
        Shared  REAL  TT,TX,TA(100),TB(100)
        Shared  REAL  TEMP(2), TMPTIM
        Shared  INTEGER  LOOP,NLOOP,NUMOBJ
      End declarations
C
C     GET NUMBER OF TIMES TO LOOP AND THE ARRAY DIMENSIONS
C
C     DATA NLOOP /21/
C     DATA NUMOBJ /1000/
C
      Barrier
        NLOOP = 21
        READ (5,*) NUMOBJ
      End barrier
C
      Forcecall INIT(A,X,NUMOBJ)
C
C     PERFORM ATTITUDE TRANSFORMATION "NLOOP" TIMES
C
      DO 100  LOOP=1,NLOOP,1
        IF  (ME .EQ. NPROCS)  THEN
          TMPTIM = etime(TEMP(1), TEMP(2))
          TB(LOOP) = TEMP(1)
        ENDIF
C
        Forcecall XFRM(A,X,Y,NUMOBJ)
C
        IF  (ME .EQ. NPROCS)  THEN
          TMPTIM = etime(TEMP(1), TEMP(2))
          TA(LOOP) = TEMP(1)
        ENDIF
100   CONTINUE
C
      IF  (ME .EQ. NPROCS)  THEN
        TT = 0.0
        DO 150,LOOP=2,NLOOP,1
          TX  = TA(LOOP) - TB(LOOP)
          TT = TT + TX
          WRITE (6,*) 'ITERATION NUMBER: ',LOOP-1,
       1              '  TIME:',TX
150     CONTINUE
C
        WRITE (6,*)
        WRITE (6,*) '***NUMBER OF OBJECTS***', NUMOBJ
        WRITE (6,*)
        WRITE (6,*) '***TOTAL TIME***', TT
        WRITE (6,*) '***AVERAGE TIME***', TT/20.
        WRITE (6,*)
C
      ENDIF
      Join
      END
C
      Forcesub XFRM(A,X,Y,NUMOBJ) of NPROCS ident ME
C
        INTEGER  NUMOBJ
        REAL  A(3,3,NUMOBJ), X(3,NUMOBJ), Y(3,NUMOBJ)
        Private  REAL  TEMP1(50000), TEMP2(50000), TEMP3(50000)
        Private  INTEGER  AXIS, OBJECT
      End declarations
C
      Pre2do  100  AXIS=1,3,1 ; OBJECT=1,NUMOBJ,1
        TEMP1(OBJECT) = A(AXIS,1,OBJECT) * X(1,OBJECT)
        TEMP2(OBJECT) = A(AXIS,2,OBJECT) * X(2,OBJECT)
        TEMP3(OBJECT) = A(AXIS,3,OBJECT) * X(3,OBJECT)
        Y(AXIS,OBJECT) = TEMP1(OBJECT) + TEMP2(OBJECT)
     1             + TEMP3(OBJECT)
100   End Presched Do
      RETURN
      END
C
      Forcesub  INIT(A,X,NUMOBJ) of NPROCS ident ME
C
        INTEGER  NUMOBJ
        REAL  A(3,3,NUMOBJ), X(3,NUMOBJ)
        Private  INTEGER  OBJECT, ROW, COL
      End declarations
C
      Presched DO  10  OBJECT=1,NUMOBJ,1
        DO 20,ROW=1,3,1
          X(ROW,OBJECT) = ROW
          DO 30,COL=1,3,1
            A(ROW,COL,OBJECT) = ROW + COL
30        CONTINUE
20      CONTINUE
10    End Presched Do
      RETURN
      END
```

C.3. Fortran Output of Force Preprocessor

```
C      Force ATTXFM of NPROCS ident ME

       SUBROUTINE PMAIN
        INTEGER NPROCS, FFFNUM,NUMLCK, BARWIT,BARLCK,FFNBAR
        INTEGER BARWIN,BARWOT,ZZNBAR,ZPCASE
        INTEGER i0i(256),i00j(256)
        COMMON /PRVTNV/  ME,ZPCASE
        COMMON /PARENV/ i0i,i00i,NPROCS, FFFNUM,NUMLCK,LRD,LWR
        COMMON /PARENV/ BARWIN,BARWOT,ZZNBAR
        COMMON /PARENV/ BARLCK,FFNBAR,BARWIT,master,i00j
        LOGICAL getstat
        INTEGER timer
C
C   The parallel environment variables are:
C   NPROCS  --  total number of processes
C   FFFNUM     --  process numbering variable
C   FFNBAR    --  barrier process counter
C   BARWIT     --  barrier wait  variable
C   BARLCK     --  barrier lock variable
C   ME   --  unique process index
C
        INTEGER ME,getpid
        DIMENSION i001i(256),i000j(256)
        COMMON / ATTXFG/ i001i,i000i

C
C        Shared  REAL  A(3,3,50000),X(3,50000),Y(3,50000)

         REAL    A(3,3,50000),X(3,50000),Y(3,50000)
       COMMON / ATTXFG / A(3,3,50000),X(3,50000),Y(3,50000)

C        Shared  REAL  TT,TX,TA(100),TB(100)

         REAL    TT,TX,TA(100),TB(100)
       COMMON / ATTXFG / TT,TX,TA(100),TB(100)

C        Shared  REAL  TEMP(2), TMPTIM

         REAL    TEMP(2), TMPTIM
       COMMON / ATTXFG / TEMP(2), TMPTIM

C        Shared  INTEGER  LOOP,NLOOP,NUMOBJ

         INTEGER    LOOP,NLOOP,NUMOBJ
       COMMON / ATTXFG / LOOP,NLOOP,NUMOBJ

C      End declarations

        COMMON / ATTXFG/ i000j
C
C     GET NUMBER OF TIMES TO LOOP AND THE ARRAY DIMENSIONS
C
C     DATA NLOOP /21/
C     DATA NUMOBJ /1000/
```

```
C
C      Barrier
       CALL spin_lock(BARLCK)
        IF (FFNBAR.LT.(NPROCS - 1)) THEN
          FFNBAR = FFNBAR + 1
          CALL spin_unlock(BARLCK)
          CALL spin_lock(BARWIT)
        ENDIF
        IF (FFNBAR .EQ. (NPROCS-1))  THEN
        NLOOP = 21
        READ (5,*) NUMOBJ
C      End barrier
       ENDIF
        IF(FFNBAR.EQ.0) THEN
          CALL spin_unlock(BARLCK)
         ELSE
          FFNBAR = FFNBAR - 1
          CALL spin_unlock(BARWIT)
        ENDIF
C
C      Forcecall INIT(A,X,NUMOBJ)
          CALL  INIT(A,X,NUMOBJ)
C
C     PERFORM ATTITUDE TRANSFORMATION "NLOOP" TIMES
C
      DO 100  LOOP=1,NLOOP,1
        IF  (ME .EQ. NPROCS)  THEN
          TMPTIM = etime(TEMP(1), TEMP(2))
          TB(LOOP) = TEMP(1)
        ENDIF
C
C        Forcecall XFRM(A,X,Y,NUMOBJ)
          CALL  XFRM(A,X,Y,NUMOBJ)
C
        IF  (ME .EQ. NPROCS)  THEN
          TMPTIM = etime(TEMP(1), TEMP(2))
          TA(LOOP) = TEMP(1)
        ENDIF
100   CONTINUE
C
      IF  (ME .EQ. NPROCS)  THEN
        TT = 0.0
        DO 150,LOOP=2,NLOOP,1
          TX  = TA(LOOP) - TB(LOOP)
          TT = TT + TX
          WRITE (6,*) 'ITERATION NUMBER: ',LOOP-1,
       1              '  TIME:',TX
150     CONTINUE
C
        WRITE (6,*)
        WRITE (6,*) '***NUMBER OF OBJECTS***', NUMOBJ
        WRITE (6,*)
        WRITE (6,*) '***TOTAL TIME***', TT
        WRITE (6,*) '***AVERAGE TIME***', TT/20.
        WRITE (6,*)
C
```

```
      ENDIF
C      Join
       RETURN

      END
C
C      Forcesub XFRM(A,X,Y,NUMOBJ) of NPROCS ident ME

        SUBROUTINE GATTXF
        INTEGER i001i(256),i000j(256),i000i,share
        COMMON / ATTXFG/ i001i,i000i
         REAL    A(3,3,50000),X(3,50000),Y(3,50000)
       COMMON / ATTXFG / A(3,3,50000),X(3,50000),Y(3,50000)

         REAL    TT,TX,TA(100),TB(100)
       COMMON / ATTXFG / TT,TX,TA(100),TB(100)

         REAL    TEMP(2), TMPTIM
       COMMON / ATTXFG / TEMP(2), TMPTIM

         INTEGER    LOOP,NLOOP,NUMOBJ
       COMMON / ATTXFG / LOOP,NLOOP,NUMOBJ

        COMMON / ATTXFG/ i000j
        IF (i000i.NE.99) THEN
           ISHARE =  share(i000i,(loc(i000j)-loc(i000i)))
           i000i = 99
        ELSE
           ISHARE = 1
        ENDIF
        IF (ISHARE.NE.1) THEN
          write(*,*) ' Force: '
          write(*,*) ' Memory Sharing error ' ,
     1               (loc(i000j)-loc(i000i))
          call exit(0)
        ENDIF
        RETURN
        END

         SUBROUTINE XFRM (A,X,Y,NUMOBJ)
        INTEGER NPROCS, FFFNUM,NUMLCK, BARWIT,BARLCK,FFNBAR
        INTEGER BARWIN,BARWOT,ZZNBAR,ZPCASE
        INTEGER i0i(256),i00j(256)
        COMMON /PRVTNV/  ME,ZPCASE
        COMMON /PARENV/ i0i,i00i,NPROCS, FFFNUM,NUMLCK,LRD,LWR
        COMMON /PARENV/ BARWIN,BARWOT,ZZNBAR
        COMMON /PARENV/ BARLCK,FFNBAR,BARWIT,master,i00j
        LOGICAL getstat
        INTEGER timer
C
C   The parallel environment variables are:
C   NPROCS  --  total number of processes
C   FFFNUM     --  process numbering variable
C   FFNBAR    --  barrier process counter
C   BARWIT     --  barrier wait  variable
C   BARLCK     --  barrier lock variable
```

```
C   ME   --  unique process index
C
        INTEGER ME,getpid
        DIMENSION i001i(256),i000j(256)
        COMMON / XFRMG/ i001i,i000i
C
        INTEGER  NUMOBJ
        REAL  A(3,3,NUMOBJ), X(3,NUMOBJ), Y(3,NUMOBJ)
C        Private  REAL  TEMP1(50000), TEMP2(50000), TEMP3(50000)
          REAL  TEMP1(50000), TEMP2(50000), TEMP3(50000)
C        Private  INTEGER  AXIS, OBJECT
          INTEGER  AXIS, OBJECT
C      End declarations
        COMMON / XFRMG/ i000j
C
C     Pre2DO  100  AXIS=1,3,1 ; OBJECT=1,NUMOBJ,1
      N1000T = (3 - 1)/1 + 1
      N2000T = (NUMOBJ - 1)/1+ 1
      DO 100  L000T = ME, N1000T*N2000T, NPROCS
        AXIS = 1 + 1 *MOD(L000T-1, N1000T)
        OBJECT = 1 + 1*((L000T - 1)/N1000T)
        TEMP1(OBJECT) = A(AXIS,1,OBJECT) * X(1,OBJECT)
        TEMP2(OBJECT) = A(AXIS,2,OBJECT) * X(2,OBJECT)
        TEMP3(OBJECT) = A(AXIS,3,OBJECT) * X(3,OBJECT)
        Y(AXIS,OBJECT) = TEMP1(OBJECT) + TEMP2(OBJECT)
     1           + TEMP3(OBJECT)
C100   End Presched Do
100     CONTINUE
      RETURN
      END
C
C      Forcesub  INIT(A,X,NUMOBJ) of NPROCS ident ME

        SUBROUTINE GXFRM
        INTEGER i001i(256),i000j(256),i000i,share
        COMMON / XFRMG/ i001i,i000i

        COMMON / XFRMG/ i000j
        IF (i000i.NE.99) THEN
           ISHARE =  share(i000i,(loc(i000j)-loc(i000i)))
           i000i = 99
        ELSE
           ISHARE = 1
        ENDIF
        IF (ISHARE.NE.1) THEN
          write(*,*) ' Force: '
          write(*,*) ' Memory Sharing error ' ,
     1               (loc(i000j)-loc(i000i))
          call exit(0)
        ENDIF
        RETURN
        END

         SUBROUTINE INIT (A,X,NUMOBJ)
        INTEGER NPROCS, FFFNUM,NUMLCK, BARWIT,BARLCK,FFNBAR
        INTEGER BARWIN,BARWOT,ZZNBAR,ZPCASE
        INTEGER i0i(256),i00j(256)
        COMMON /PRVTNV/  ME,ZPCASE
        COMMON /PARENV/ i0i,i00i,NPROCS, FFFNUM,NUMLCK,LRD,LWR
        COMMON /PARENV/ BARWIN,BARWOT,ZZNBAR
        COMMON /PARENV/ BARLCK,FFNBAR,BARWIT,master,i00j
        LOGICAL getstat
        INTEGER timer
C
C   The parallel environment variables are:
C   NPROCS  --  total number of processes
C   FFFNUM     --  process numbering variable
C   FFNBAR    --  barrier process counter
C   BARWIT     --  barrier wait  variable
C   BARLCK     --  barrier lock variable
C   ME   --  unique process index
C
        INTEGER ME,getpid
        DIMENSION i001i(256),i000j(256)
        COMMON / INITG/ i001i,i000i

C
        INTEGER  NUMOBJ
        REAL  A(3,3,NUMOBJ), X(3,NUMOBJ)
C        Private  INTEGER  OBJECT, ROW, COL
          INTEGER  OBJECT, ROW, COL
C      End declarations

        COMMON / INITG/ i000j
C
C      Presched DO  10  OBJECT=1,NUMOBJ,1
       DO  10  OBJECT = 1 +1*( ME - 1), NUMOBJ, 1*NPROCS
        DO 20,ROW=1,3,1
          X(ROW,OBJECT) = ROW
          DO 30,COL=1,3,1
            A(ROW,COL,OBJECT) = ROW + COL
30        CONTINUE
20      CONTINUE
C10    End Presched Do
10     CONTINUE
      RETURN
      END

        SUBROUTINE GINIT
        INTEGER i001i(256),i000j(256),i000i,share
        COMMON / INITG/ i001i,i000i
        COMMON / INITG/ i000j
        IF (i000i.NE.99) THEN
           ISHARE =  share(i000i,(loc(i000j)-loc(i000i)))
           i000i = 99
        ELSE
           ISHARE = 1
        ENDIF
        IF (ISHARE.NE.1) THEN
          write(*,*) ' Force: '
          write(*,*) ' Memory Sharing error ' ,
     1               (loc(i000j)-loc(i000i))
```

```
  call exit(0)
ENDIF
RETURN
END

SUBROUTINE MEMSHR
CALL GATTXF

CALL GXFRM

CALL GINIT

RETURN
END
```

C.4. Optimized Force Version of Altitude Transformation Program

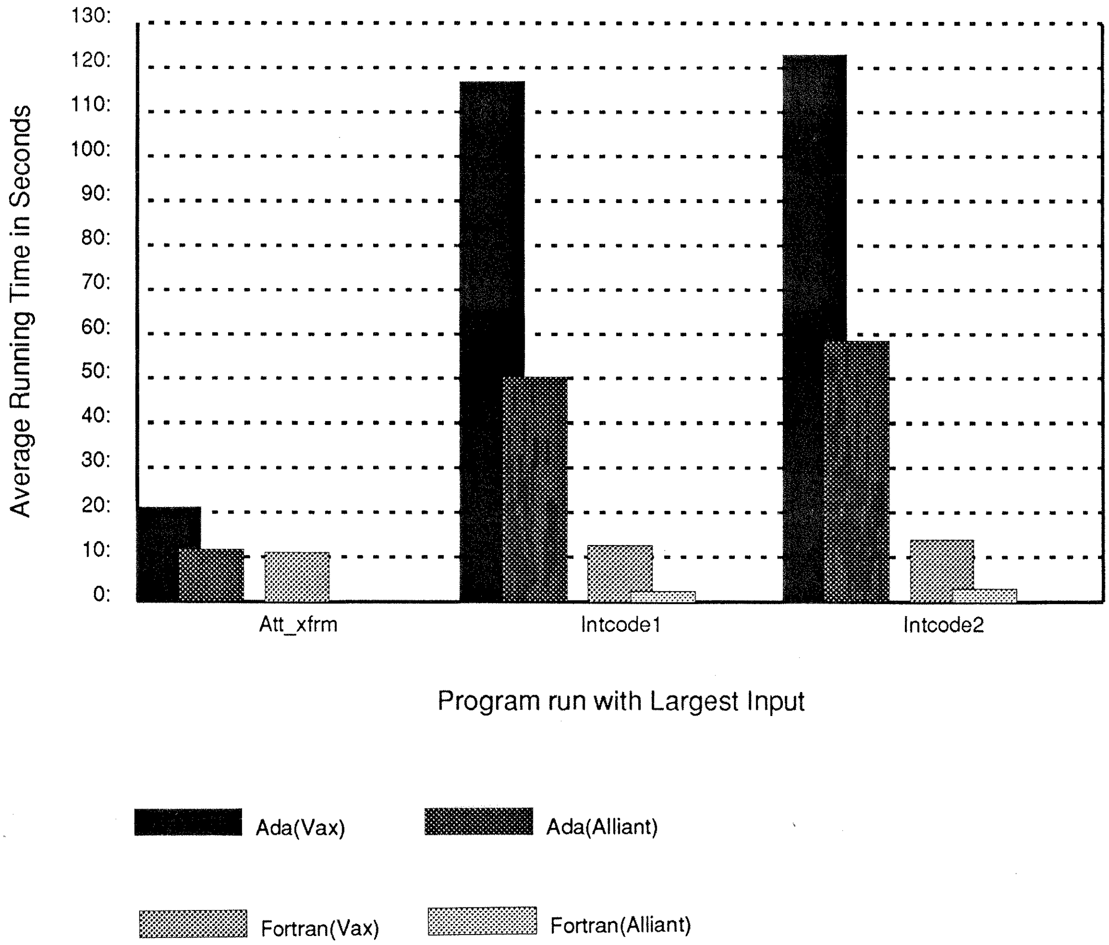
```
      Force ATTXFM of NPROCS ident ME
C
        Shared  REAL  A(3,3,50000),X(3,50000),Y(3,50000)
        Shared  REAL  TT,TX,TA(100),TB(100)
        Shared  REAL  TEMP(2), TMPTIM
        Shared  INTEGER  LOOP,NLOOP,NUMOBJ
      End declarations
C
C     GET NUMBER OF TIMES TO LOOP AND THE ARRAY DIMENSIONS
C
C     DATA NLOOP /21/
C     DATA NUMOBJ /1000/
C
      Barrier
        NLOOP = 21
        READ (5,*) NUMOBJ
      End barrier
C
      Forcecall INIT(A,X,NUMOBJ)
C
C     PERFORM ATTITUDE TRANSFORMATION "NLOOP" TIMES
C
      DO 100  LOOP=1,NLOOP,1
        IF  (ME .EQ. NPROCS)  THEN
          TMPTIM = etime(TEMP(1), TEMP(2))
          TB(LOOP) = TEMP(1)
        ENDIF
C
        Forcecall XFRM(A,X,Y,NUMOBJ)
C
        IF  (ME .EQ. NPROCS)  THEN
          TMPTIM = etime(TEMP(1), TEMP(2))
          TA(LOOP) = TEMP(1)
        ENDIF
100   CONTINUE
C
      IF  (ME .EQ. NPROCS)  THEN
        TT = 0.0
        DO 150,LOOP=2,NLOOP,1
          TX  = TA(LOOP) - TB(LOOP)
          TT = TT + TX
          WRITE (6,*) 'ITERATION NUMBER: ',LOOP-1,'  TIME:',TX
150     CONTINUE
C
        WRITE (6,*)
        WRITE (6,*) '***NUMBER OF OBJECTS***', NUMOBJ
        WRITE (6,*)
        WRITE (6,*) '***TOTAL TIME***', TT
        WRITE (6,*) '***AVERAGE TIME***', TT/20.
        WRITE (6,*)
C
      ENDIF
C
      Join
      END
C
C
C
      Forcesub XFRM(A,X,Y,NUMOBJ) of NPROCS ident ME
C
        INTEGER  NUMOBJ
        REAL  A(3,3,NUMOBJ), X(3,NUMOBJ), Y(3,NUMOBJ)
        Private  INTEGER  AXIS, OBJECT
      End declarations
C
      Pre2do  100  OBJECT=1,NUMOBJ,1 ; AXIS=1,3,1
        Y(AXIS,OBJECT) = A(AXIS,1,OBJECT) * X(1,OBJECT)
     1                 + A(AXIS,2,OBJECT) * X(2,OBJECT)
     2                 + A(AXIS,3,OBJECT) * X(3,OBJECT)
100   End Presched Do
      RETURN
      END
C
      Forcesub  INIT(A,X,NUMOBJ) of NPROCS ident ME
C
        INTEGER  NUMOBJ
        REAL  A(3,3,NUMOBJ), X(3,NUMOBJ)
        Private  INTEGER  OBJECT, ROW, COL
      End declarations
C
      Presched DO  10  OBJECT=1,NUMOBJ,1
        DO 20,ROW=1,3,1
          X(ROW,OBJECT) = ROW
          DO 30,COL=1,3,1
            A(ROW,COL,OBJECT) = ROW + COL
30        CONTINUE
20      CONTINUE
10    End Presched Do
      RETURN
      END
```

APPENDIX D

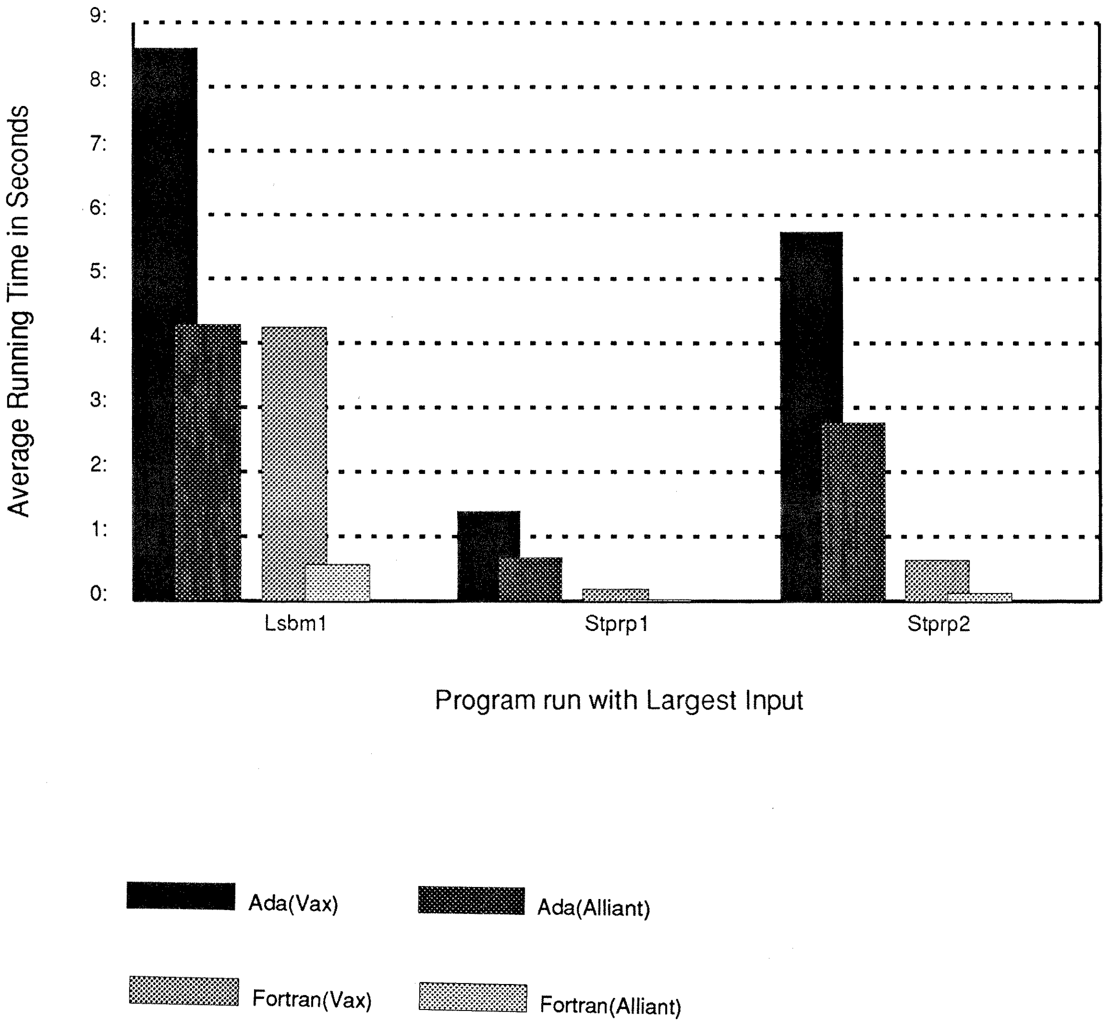
Graphical Results

D.1. Ada: *Vax 11-780 and Alliant Results*

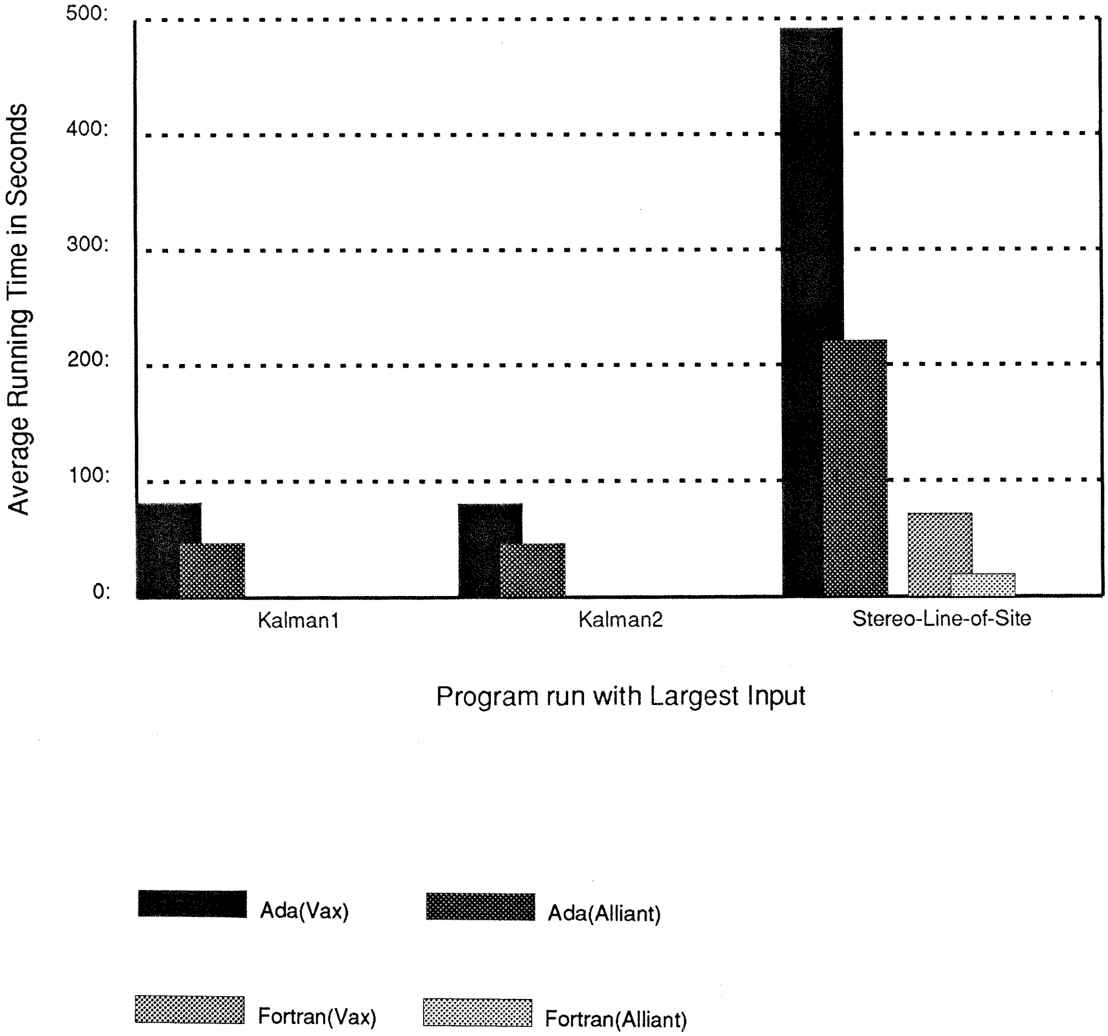
For comparison, the Alliant *Fortran* sequential scalar and Vax *Fortran* results are included.



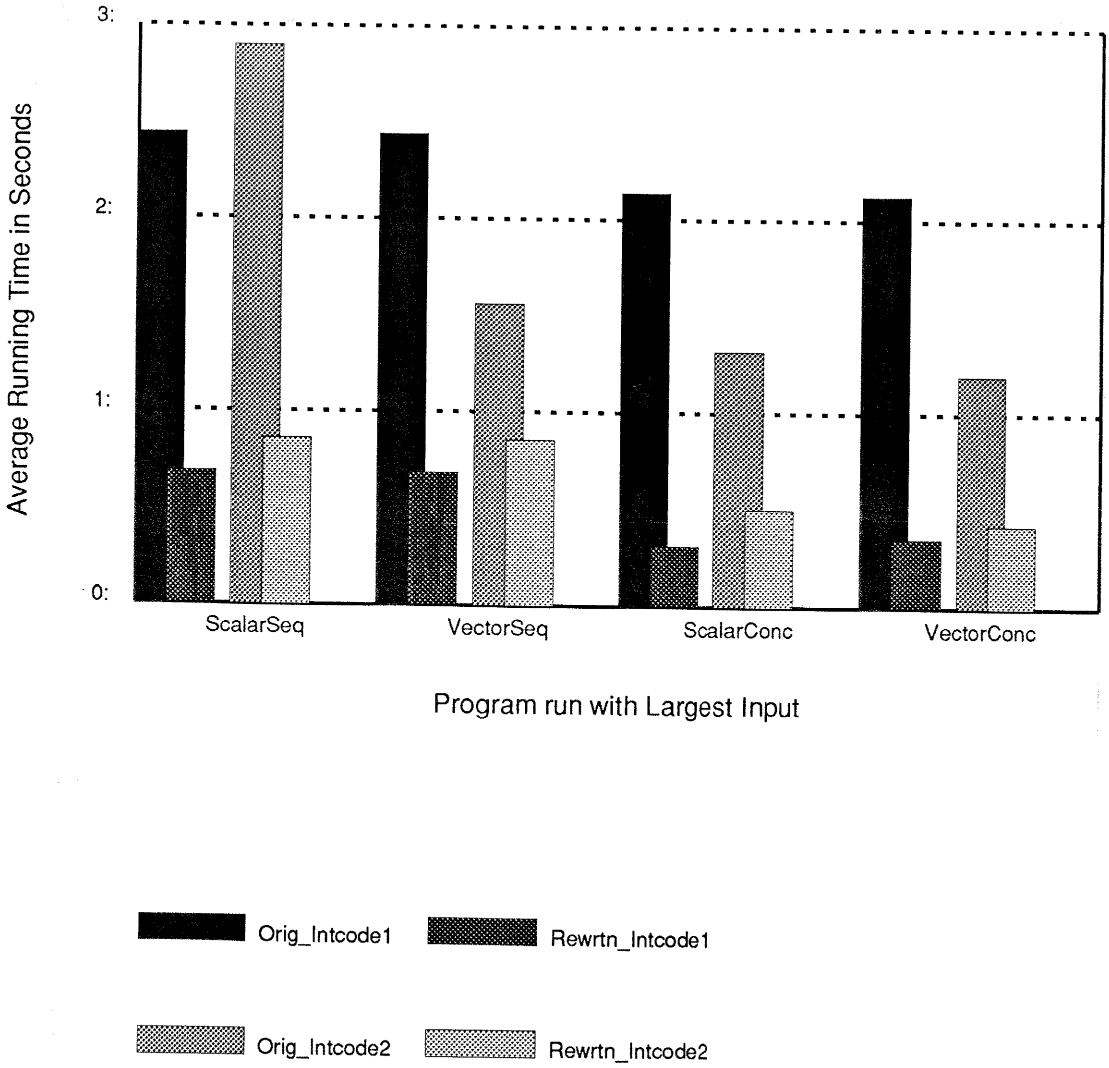
D.1. Ada: *Vax 11-780 and Alliant Results (cont.)*



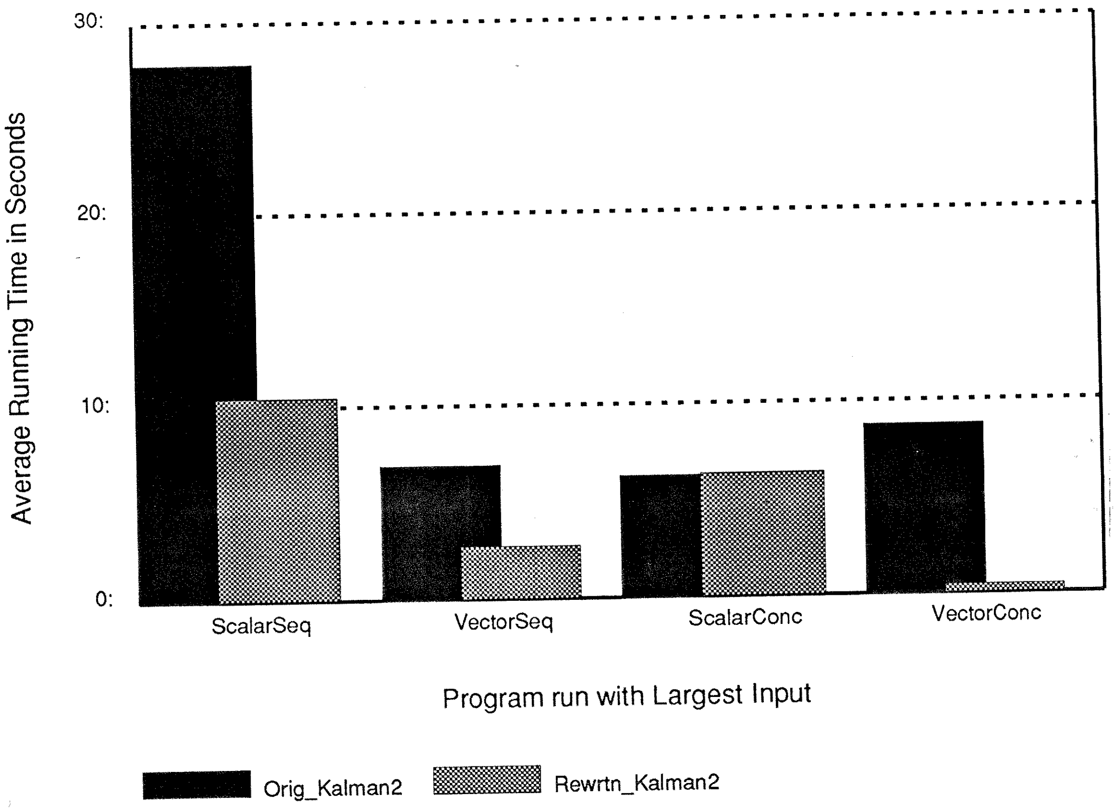
D.1. Ada: *Vax 11-780 and Alliant Results (cont.)*



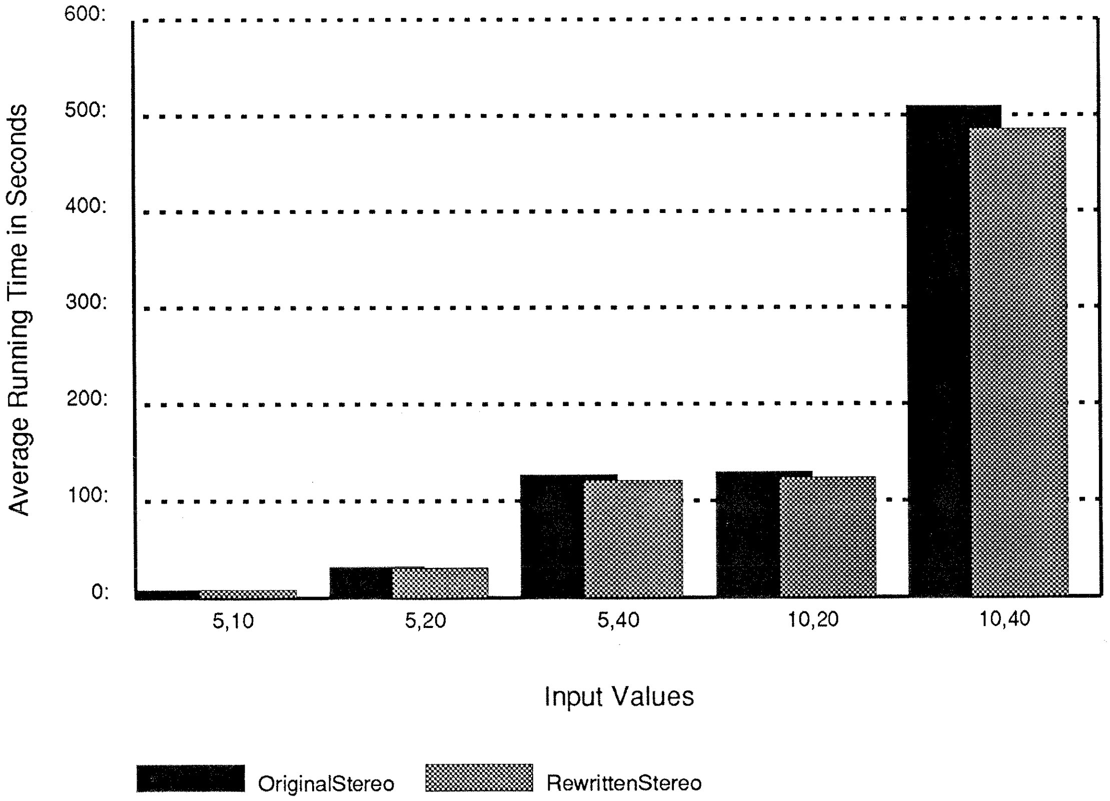
D.2. Fortran: *Alliant Results*



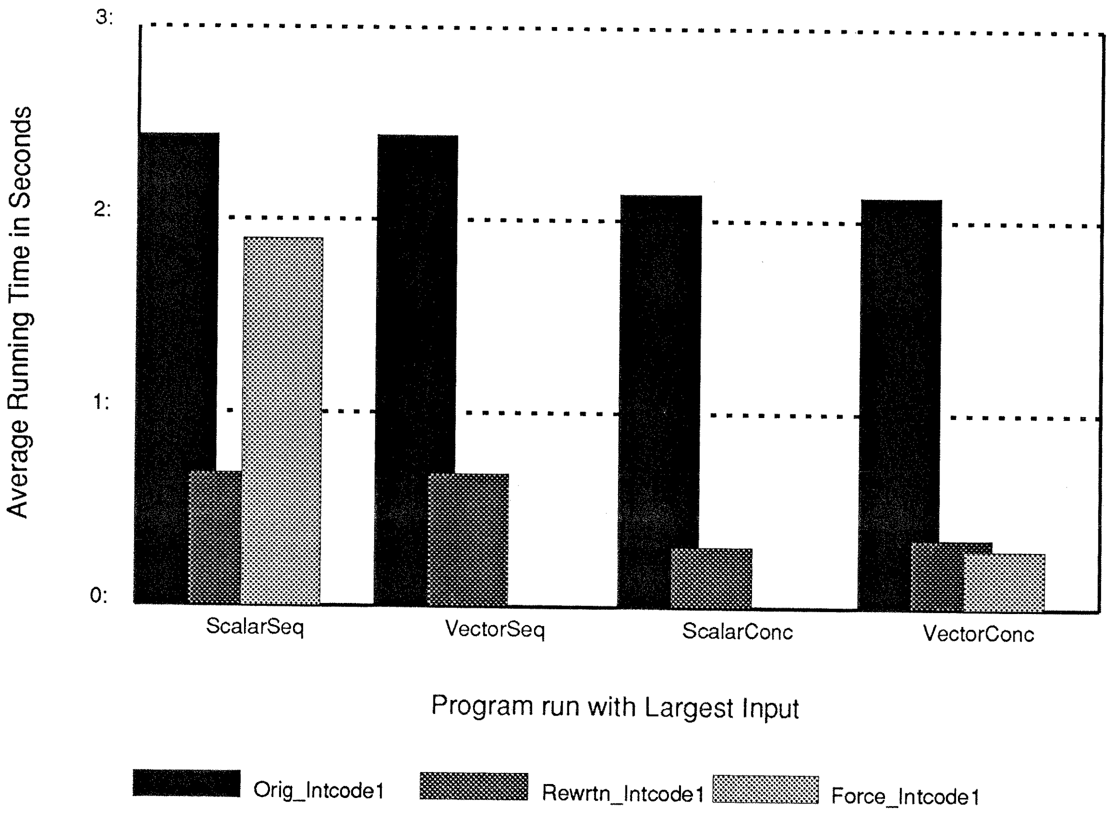
D.2. Fortran: *Alliant Results (cont.)*



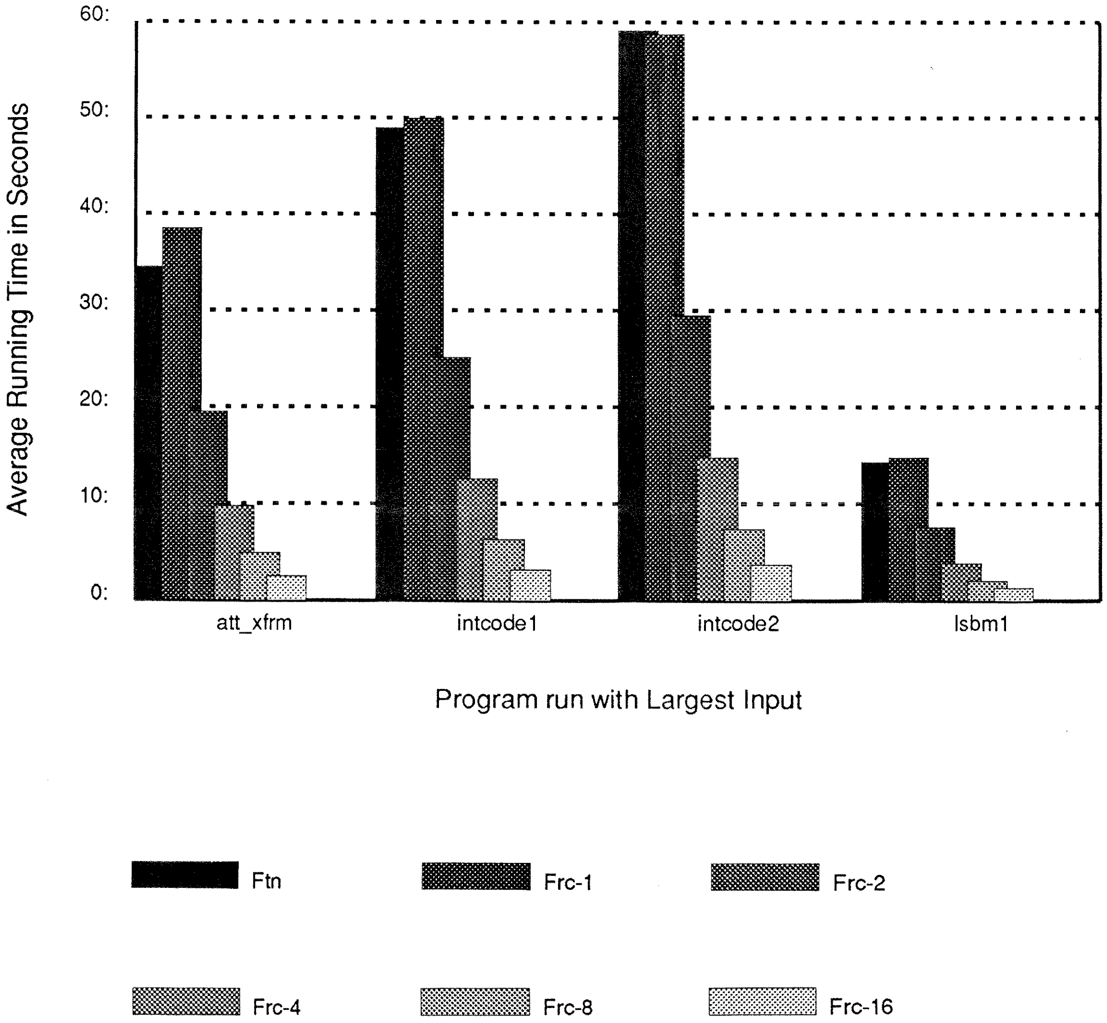
D.3. Fortran: *Encore Multimax Results*



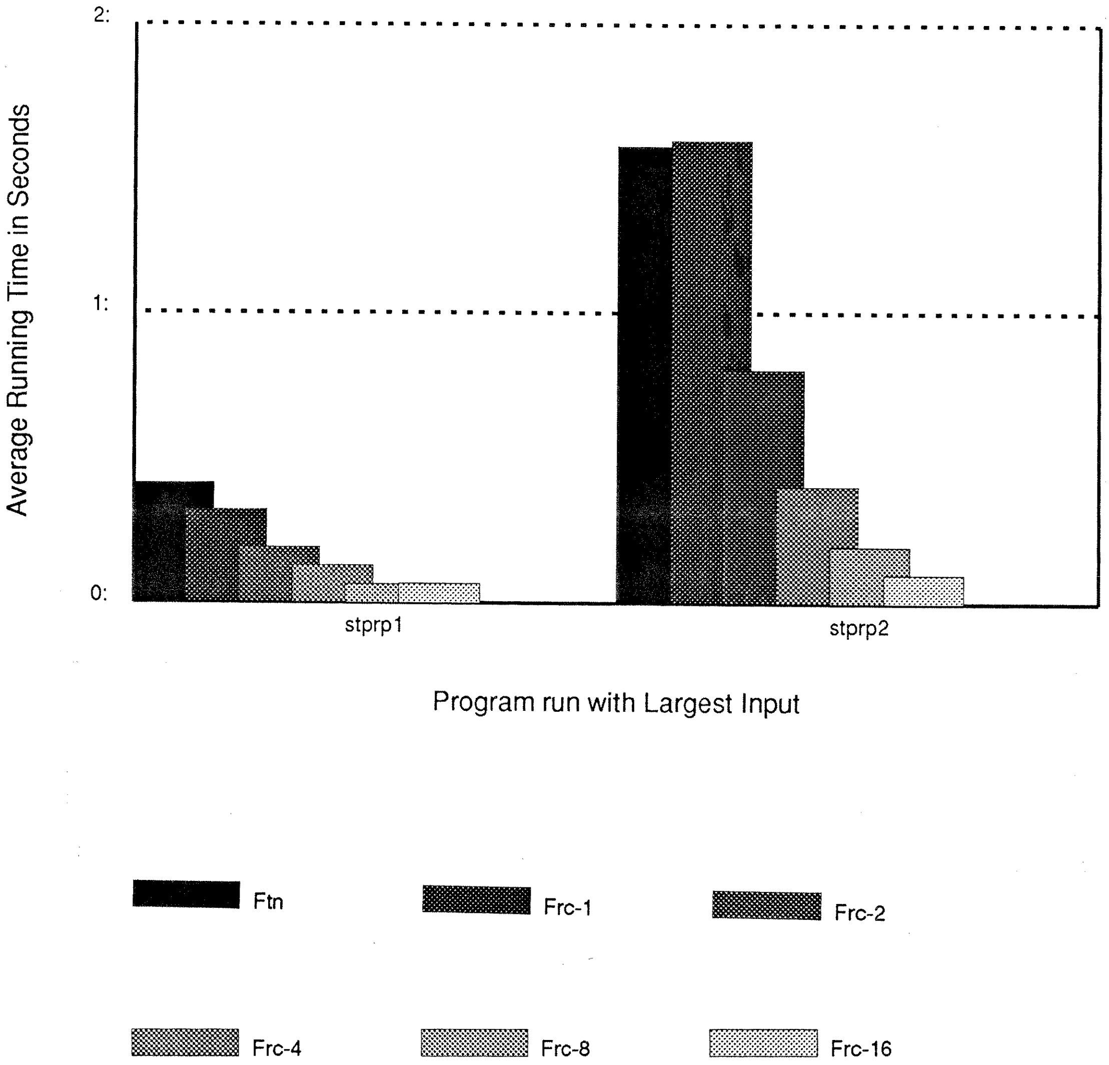
D.4. Fortran vs. The Force: *Alliant Results*



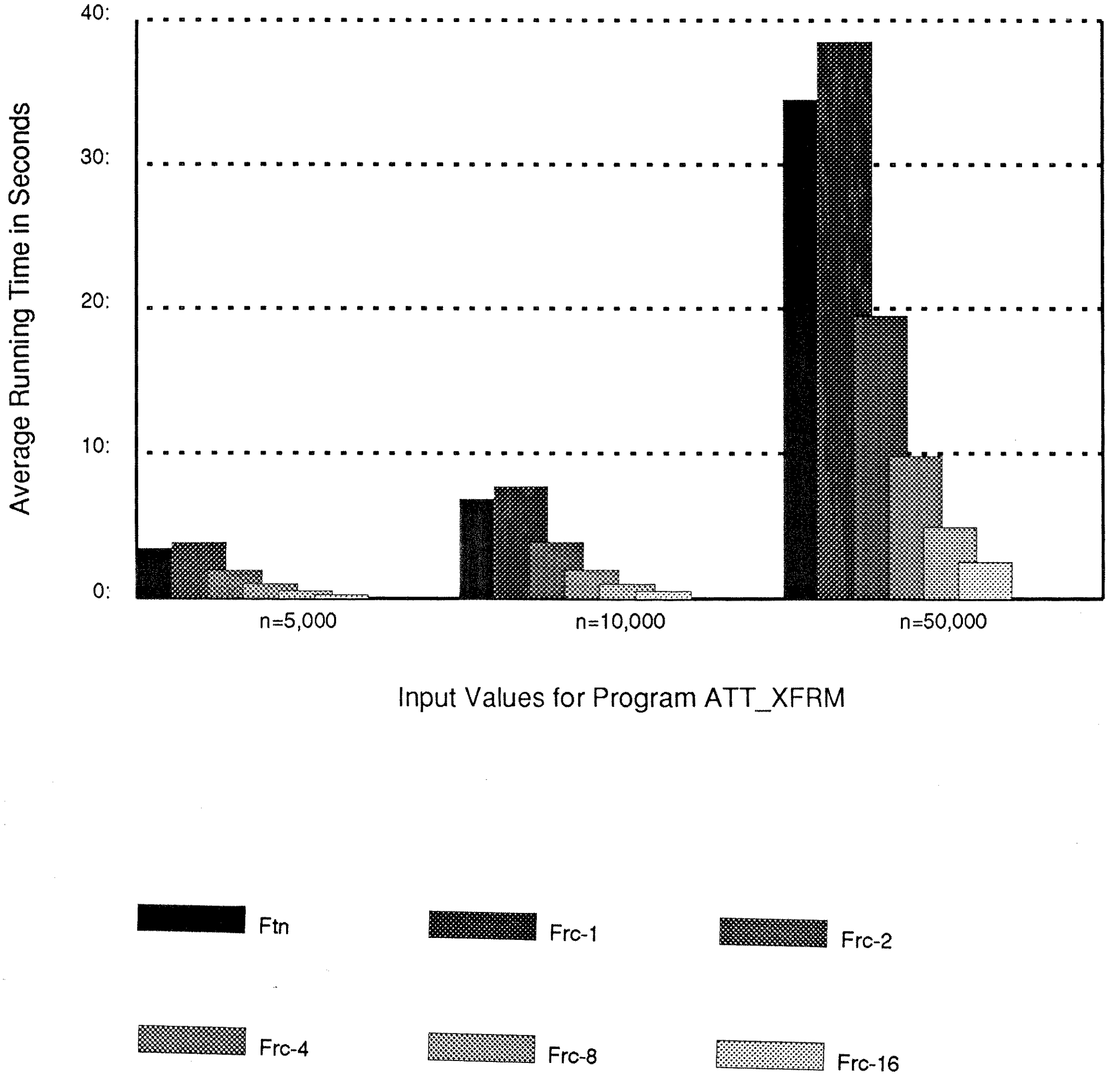
D.5. Fortran vs. The Force: *Encore Multimax Results*



D.5. Fortran vs. The Force: *Encore Multimax Results (cont.)*



D.5. Fortran vs. The Force: *Encore Multimax Results (cont.)*



APPENDIX E

Tables of Optimized Storage Results for Encore Fortran Programs

Encore		**Fortran vs Force vs Optimized Force** **Attitude Transformation**				
Number of Objects	*Number of Procs*	Lowest Average User Time/Iteration for Number of Processors				
		Sequential	Force	*Speedup*	Opt. Force	*Speedup*
5,000	*1*	1.85	2.10	*0.881*	1.72	*1.08*
	2		1.07	*1.73*	0.867	*2.13*
	4		0.539	*3.43*	0.433	*4.27*
	8		0.274	*6.75*	0.218	*8.49*
	16		0.138	*13.4*	0.113	*16.4*
10,000	*1*	3.70	4.19	*0.883*	3.45	*1.07*
	2		2.13	*1.74*	1.74	*2.13*
	4		1.08	*3.43*	0.884	*4.19*
	8		0.543	*6.81*	0.448	*8.26*
	16		0.276	*13.4*	0.228	*16.2*
50,000	*1*	18.6	21.0	*0.886*	17.2	*1.08*
	2		10.7	*1.74*	8.73	*2.13*
	4		5.38	*3.46*	4.41	*4.22*
	8		2.75	*6.76*	2.24	*8.30*
	16		1.39	*13.4*	1.16	*16.0*

Time in seconds

Encore **Fortran vs Force vs Optimized Force**
State Vector Integration 1

Number of Objects	*Number of Procs*	Lowest Average User Time/Iteration for Number of Processors				
		Sequential	Force	*Speedup*	Opt. Force	*Speedup*
5,000	*1*	3.92	4.06	*0.966*	2.84	*1.38*
	2		2.05	*1.91*	1.43	*2.74*
	4		1.03	*3.81*	0.718	*5.46*
	8		0.512	*7.66*	0.357	*11.0*
	16		0.258	*15.2*	0.181	*21.7*
10,000	*1*	7.83	8.11	*0.965*	5.70	*1.37*
	2		4.08	*1.92*	2.85	*2.75*
	4		2.06	*3.80*	1.44	*5.44*
	8		1.02	*7.68*	0.714	*11.0*
	16		0.518	*15.2*	0.359	*21.8*
50,000	*1*	39.2	40.6	*0.966*	28.4	*1.38*
	2		20.4	*1.92*	14.3	*2.74*
	4		10.2	*3.84*	7.15	*5.48*
	8		5.11	*7.67*	3.61	*10.9*
	16		2.56	*15.3*	1.82	*21.5*

Time in seconds

Encore **Fortran vs Force vs Optimized Force**
State Vector Integration 2

Number of Objects	*Number of Procs*	Lowest Average User Time/Iteration for Number of Processors				
		Sequential	Force	*Speedup*	Opt. Force	*Speedup*
5,000	*1*	4.76	4.70	*1.01*	3.20	*1.49*
	2		2.36	*2.02*	1.61	*2.96*
	4		1.19	*4.00*	0.812	*5.86*
	8		0.593	*8.03*	0.403	*11.8*
	16		0.298	*16.0*	0.203	*23.4*
10,000	*1*	9.52	9.39	*1.01*	6.40	*1.49*
	2		4.73	*2.01*	3.22	*2.96*
	4		2.37	*4.02*	1.62	*5.88*
	8		1.19	*8.00*	0.806	*11.8*
	16		0.593	*16.1*	0.405	*23.5*
50,000	*1*	47.6	47.1	*1.01*	32.0	*1.49*
	2		23.6	*2.02*	16.1	*2.96*
	4		11.9	*4.00*	8.12	*5.86*
	8		5.96	*7.99*	4.07	*11.7*
	16		3.00	*15.9*	2.05	*23.2*

Time in seconds