Autotasking STAGE-2

Robert J. Bergeron
Computer Sciences Corporation
NASA Ames Research Center
Moffett Field, CA 94035, USA

Abstract

This paper presents Cray Y-MP performance results for a parallel version of a two-dimensional CFD code, STAGE-2. The Cray autotasker combined with minimal user effort produced a 91% parallel code displaying a fivefold increase in performance on 8 CPUs. The high level decomposition implicit in the code formulation facilitated the generation of autotasked code. Examination of STAGE-2 parallel performance indicates that two factors limit the current version: the load imbalance in the block solver and the intensity of memory traffic in the implicit solution subroutines.

1.0 Introduction

This report discusses performance results obtained by using Cray's automatic preprocessor to create a parallel version of the CFD code STAGE-2 (Gundy et al., 1989). This effort is part of a larger effort to create a suite of codes for testing automatic parallelization. Tools for automatic parallelization of application codes fit into two categories: low-level tools which exploit machine architectural characteristics and high-level tools which exploit algorithmic parallelism. This report describes results obtained with a low-level tool, FPP, which exploits high speed registers to achieve parallelism at the DO loop level. Test suite ground rules currently limit manual restructuring of the code to inserting parallel directives and rearranging some DO loops. Future application of high-level tools to codes in the suite may require a relaxation of these ground rules. The following paragraphs discuss salient features of the Cray autotasker and the measurement of parallel performance.

The Cray Y-MP is a shared memory multiprocessor, and high speed registers shared among all the CPUs transmit the status of arithmetic operations for CPU synchronization. Cray's implementation of automatic parallel processing employs these high speed shared registers to assist in the parallel execution of FORTRAN DO loops. For scientific and engineering applications, the DO loops will contain most of the computation and since individual iterations of such DO loops tend to contain an equal amount of calculation, emphasis on DO loops assists in load balancing. Implicit synchronization occurs at the end of all autotasked DO loops.

Loop-level parallel processing requires compiler directives. Autotasking is the automatic insertion of the directives by a preprocessor; microtasking is the insertion of equivalent loop-level directives by the user (Cray, 1988). Cray terms the autotasking preprocessor FPP (FORTRAN PreProcessor) and the microtasking preprocessor FMP (FORTRAN MidProcessor). Directives tell the compiler which memory locations can be shared by all processors executing the parallel job and which memory locations must belong to the individual processors of the parallel job. Directives also specify the distribution of parallel work to the processors.

FPP directives (statements beginning with CFPP$) allow the user to improve the dependency analysis performed by the autotasker. FMP directives (statements beginning with CMIC$) allow users to either perform their own dependency analysis or to override the FPP dependency analysis. The user typically invokes FPP to insert the parallel directives into the code with the type and location of these directives being determined by FPP's dependency analysis. The user then supplies the modified code to FMP for translation into parallel library calls and autotasking intrinsic functions. Finally, the user provides the FMP-modified code to the compiler for object code generation.

The figure of merit for parallel performance is speedup, defined as the ratio of the elapsed time for the code executing on 1 CPU to the elapsed time for the code executing on a specific number of CPUs. Except where stated, the specific number of CPUs in this report is 8. The ultimate test of an automatic parallelization tool is the performance of the code generated by the tool. On a vector machine, however, a tool can generate code which is highly parallel (meaning that the generated code executing on multiple CPUs displays high performance relative to the same code executing on a single CPU) and still not take advantage of the vector registers. In such a case, the algorithm may be ill-suited for parallel processing, but the tool may be performing quite well. Thus, in addition to speedup, this report will refer to the fraction of parallel code. The fraction of parallel code is equal to 1-f where f, the serial fraction, is defined implicitly as (Karp and Flatt, 1989):

T(p) = T(1)*f + \(T(1)*(1-f),p)

where T(p) denotes the elapsed time on p processors and T(1) denotes the elapsed time on one processor. This report will provide measured single and multiple processor elapsed times and the above formula will allow calculation of the parallel code fraction.

A predictive tool, ATEXPERT (Cray, 1991) provides estimates of autotasking performance gains at the program, subroutine, and loop level. This tool records the distribution of parallel work to various numbers of processors and employs a simple model to extrapolate wall-clock times. The tool can run in the normal workload to estimate the effects of FPP directives upon parallel performance during dedicated time. The report shows some tables and graphs from this tool.

Section 2 of this report briefly discusses the physical problem modelled by STAGE-2 and Section 3 briefly explains the numerical approach employed in the code to solve the governing equations. Section 4 provides singletasked performance data while Section 5 provides the parallel performance obtained by autotasking. Section 6 discusses features in the code design which promote and inhibit parallelization and also projects some characteristics of a more parallel version of STAGE-2. Section 7 provides concluding remarks.

2.0 The Physical Problem

STAGE-2 describes the two-dimensional fluid flow around rotor/stator elements in a multistage compressor. The design generally consists of rows of multiple closely-packed rotor/stator airfoil pairs with each rotor moving relative to its stator. Although measured Mach numbers of 0.15 indicate an almost incompressible flow, the geometry leads to strong viscous effects including wake interference, vortex shedding, and flow separation. The relative motion between rotor and stator promotes an unsteady flow.

3.0 The Numerical Approach

A single grid treatment of the flow field in the rotor/stator geometry would require a highly skewed mesh since the close packing of the airfoils produces many regions where the dependent variables and their gradients change rapidly. Modern CFD codes employ a system of overlaid and patched grids of varying sizes to provide accurate multiregion solutions within a reasonable period of time. Such codes generally visit each grid serially, solving the relevant equations and exchanging the appropriate boundary conditions at the end of each iteration. STAGE-2 employs this approach, treating the compressor as two regions, an inner zone where viscous effects are important and an outer zone where viscous effects can be neglected. In the inner zones near the airfoil surfaces, STAGE-2 uses thin-layer Navier-Stokes equations. In the outer zones corresponding to regions far away from surfaces, the code neglects viscous effects and employs the Euler equations.

Modelling of the above flow system involves the motion of one grid relative to another with the attendant requirement that information be passed across the interface in a time-accurate manner. The discussion below will show that the STAGE-2 formulation avoids much of the indirect addressing that can characterize other systems, e. g., unstructured grids. Simplicity of array subscripting facilitates autotasking dependency analysis.

Application of a fully implicit, finite-difference method to the governing equations produces a nonlinear set of equations which would require expensive iterative techniques for solution. Linearization of the fluxes yields a system which can be solved, albeit expensively, with a block matrix solver. Further refinement of the system through an approximate factorization technique results in a system much more amenable to block matrix solution. However, the factorization introduces an error which requires a small number of iterations during each timestep to ensure convergence.

4.0 The Computational Implementation

The main loop of STAGE-2 consists of an outer loop performing the Newton-Raphson iteration to obtain convergence during the time step. The inner loop visits each of the two-dimensional grids to provide explicit and implicit contributions to the fluxes. For outer grids which model regions far away from surfaces, the code computes only inviscid contributions to the fluxes. For inner grids, the code computes viscous and inviscid contributions to the fluxes. After assembling all flux terms for a given grid, the code calls the block-solver. Upon conclusion of the inner loop, the code applies the proper boundary equations for each grid and begins another pass of the Newton-Raphson iteration. This version of the code performs three iterations before advancing the timestep.

The sample problem models 12 grids, 6 inner and 6 outer, for a total of 850,768 points. The problem executes 10 timesteps, which is typically sufficient to obtain an initial convergence of time-averaged quantities; final convergence may require a factor of 10 more computation. The 10-timestep problem requires 456 Y-MP CPU seconds.

The table below provides a Cray PERFTRACE breakdown for the sample problem. The table shows subroutine name, CPU time spent in the routine, percentage of CPU time, subroutine MFLOPS, and comments. The table list subroutines in order of decreasing CPU time.

TABLE 1. STAGE-2 Subroutine Performance

ROUTINE	CPU Time	Percent	MFLOPS	Comments
FLUX	1.28E+02	28.2	199	Osher fluxes
BTRI	9.48E+01	20.8	202	Block solver
SMATRX	8.03E+01	17.6	214	Calculate matrix terms
LHSO	2.29E+01	5.0	60	Outer zone implicit terms
LHSI	1.83E+01	4.0	59	Inner zone implicit terms
MUTUR	1.48E+01	3.3	103	Eddy viscosity
VMAT	1.46E+01	3.2	117	Add viscous terms
RHSO	1.33E+01	2.9	169	Explicit terms
CONTROL	1.23E+01	2.7	85	Main timestep iteration
GETQ	1.21E+01	2.7	0	Array-filler
RHSI	1.17E+01	2.6	156	Explicit terms
VRHS	6.14E+00	1.3	185	Viscous terms explicit
GETOLD	5.17E+00	1.1	0	Array-filler
MUKIN	4.64E+00	1.0	186	Viscous terms
EIGEN	3.58E+00	0.8	191	Timestep calculation
PUTQ	3.29E+00	0.7	0	Array-filler
CONVRG	2.84E+00	0.6	6	Convergence check
METRIC	2.11E+00	0.5	227	Transformation metrics
DATA	1.13E+00	0.2	2	Data readin
GETXY	1.09E+00	0.2	0	Array-filler
PUTOLD	1.09E+00	0.2	0	Array-filler
PATCH	4.85E-01	0.1	32	Grid condition transfer
CORRECT	2.89E-01	0.1	43	Boundary conditions
INITIA	2.11E-01	0.0	43	Initialization of variables
INTRP	5.62E-02	0.0	99	Transfer grid conditions
PUTXY	2.48E-02	0.0	0	Array-filler
HISAVG	1.46E-02	0.0	186	Save history

OUTPUT	1.42E-02	0.0	9	Provides output
HISTIM	2.82E-03	0.0	191	Save history
GETSTR	1.04E-04	0.0	0	File verification
MATCH	8.75E-05	0.0	39	Inter-zone geometry
STAGE-2	5.23E-05	0.0	0	Main program
REDWRT	2.38E-05	0.0	0	I/O for plotting
CHECK	8.15E-06	0.0	4	Input checker
TOTALS	4.56E+02	100	167	

The code executes the 73 MW sample problem at 165 MFLOPS with 96% of its operations being vectorized. The high performance makes STAGE-2 a good candidate for a performance gain through parallelism since the compiler is able to vectorize many of the computationally intensive loops. The Cray autotasker should be able to distribute DO loop iterations to available processors to decrease elapsed time. If 96% of the operations (all of the vector operations) can be executed in parallel, application of Amdahl's law for parallelism (Section 1) indicates a maximum 8-CPU speedup of 6.25

STAGE-2 spends 92% of its time in 12 subroutines which each exceed 1% of the total CPU time. This coarseness assists in identifying the routines important for parallelization.

5.0 Autotasking the code

Autotasking the code involved examining each subroutine which consumed more than 1% of the CPU time, as provided by the PERFTRACE output. The first step involved obtaining an understanding of parallel performance with no user-supplied directives. Previous experience with autotasking had indicated that automatic inlining and inclusion of inner loops for autotasking provided the most effective options.

The CRAY utility, ATEXPERT, can predict dedicated performance speedups for various numbers of CPUs. Briefly, ATEXPERT inserts timing statements to record the wall-clock time spent in serial and parallel portions of the code. For a subroutine, the fraction of parallel code is the ratio of the total time required to execute the parallel portion in singletasked mode divided by the time required to execute the entire subroutine (serial and parallel) in singletasked mode. This estimate is the maximum fraction since it excludes any overhead due to parallel execution. Dedicated time speedups for parallel regions are based upon the maximum parallel fraction and minimum wall-clock times measured for these regions in the batch runs. Since ATEXPERT employs the wall-clock time in a loaded system to estimate the wall-clock time in a dedicated system, variations in the system load can lead to variations in ATEXPERT-predicted speedups. ATEXPERT predicted the following dedicated speedups for STAGE-2:

TABLE 2. Initial ATEXPERT-predicted Speedups (with -ei6 option)

SUBROUTINE	SPEEDUP
FLUX	7.94
BTRI	3.74
SMATRX	7.97
LHSO	4.65
LHSI	5.78
MUTUR	1.01
VMAT	1.77
RHSO	2.25
CONTROL	2.22
GETQ	7.13
RHSI	1.65
VRHS	1.59
GETOLD	7.38
MUKIN	7.92
EIGEN	4.22
PUTQ	7.84
CONVRG	1.00
METRIC	7.26
DATA	1.00
GETXY	7.86
PUTOLD	7.92
PATCH	1.40
CORRECT	1.37
INITIA	7.78
INTRP	1.00
PUTXY	1.09
HISAVG	2.84

OUTPUT	1.00
HISTIM	1.50
GETSTR	1.00
MATCH	1.35
STAGE-2	1.00
REDWRT	0.19
CHECK	1.00
ENTIRE PROGRAM	3.09

The 8-CPU speedup of 3.09 is measured relative to execution to ATEXPERT's estimate of the singletasked execution in dedicated mode.
Amdahl's law for parallelism indicates that the autotasker was able to create a code which was 70% parallel.

The initial results seem promising because the autotasker had achieved a high speedup with default options; the obvious candidate for increased attention is the block solver BTRI. Other routines meriting attention for increased parallel performance are LHSO, MUTUR, RHSO, RHSI, VMAT, and VRHS.

5.1 Autotasking the Solver

Approximate factorization of the STAGE-2 implicit formulation of the Navier-Stokes equations leads to a block tridiagonal system, composed of 4-by-4 matrices. The solver applies a lower-upper decomposition to each of the 4-by-4 matrices within the general framework of a block-tridiagonal algorithm (Pulliam and Chausee, 1981). The factorization requires two sweeps per region, i.e., one for each direction, and STAGE-2 applies this technique to each of the 12 two-dimensional regions comprising the entire grid.

The block tridiagonal algorithm is a recursive block Gaussian elimination algorithm which economizes on both the number of operations required to solve the system and array storage requirements (Isaacson and Keller, 1969). Since this algorithm is recursive in only one dimension, the block Gaussian elimination will vectorize in the opposite dimension. With typical loop lengths for the STAGE-2 sample problem ranging between 200 and 300, the block solver BTRI will obtain 200 MFLOPS on the Y-MP. The recursive nature of the solver, combined with FPP's constraint to maintain vectorization, allowed FPP to create a solver which, while vectorizable, was only moderately parallel. The report will subsequently refer to this version of STAGE-2 as the Moderately Parallel Vector (MPV) version.

Manual restructuring of the FORTRAN code in BTRI produced a highly parallel algorithm, but the recursive nature of the block elimination algorithm prevented vectorization of the inner loops when restructured for complete parallelism. Moreover, the Cray autotasker simply could not create a parallel version of the highly parallel solver despite various directives requesting that it examine only a single outer loop for parallelism and ignore inner loops. On highly parallel versions of the solver, FPP first optimized the code for vectorization and then could not parallelize the resulting code. This vectorization bias will force careful examination of autotasker-produced code on multiprocessor machines where parallel versions of scalar algorithms may outperform singletasked vector versions of the same algorithm. An alternative Cray utility ATSCOPE (Cray, 1991) cast the solver into parallel form in its first attempt; this utility inserts the directives for parallel execution without performing optimization of user code. Singletasked test versions of the highly parallel block solver performed at scalar speeds, but the parallel version of this algorithm displayed a speedup of 7.25. The restructured algorithm has the potential for high performance on highly parallel systems. The report will subsequently refer to this version of STAGE-2 as the Highly Parallel Scalar (HPS) version.

In general, the autotasker attempts to distribute work performed by each iteration of the outermost DO loop to each of the processors. High performance requires that the distributed work be vector in nature. Thus, a structure promoting efficient autotasking would be:

```
cmic$ do all parallel...
     do 200 no=1,nouter
     .
     .
     .
c
cc  vector work in the next loop
c
          do 100 ni=1,ninner
          .
          .
          .
 100      continue
     .
     .
     .
 200 continue
 cmic$ end parallel
```

Three-dimensional codes which apply the block tridiagonal solver to the entire mesh (as opposed to the STAGE-2 zonal approach) can have another loop in the structure as in:

```
cmic$ do all parallel...
     do 300 np=1,nplane
     .
     .
     .
c
cc  vector work in the next loop
c
          do 200 no=1,nouter
          .
          .
          .
c
cc scalar work in the next loop
c
               do 100 ni=1,ninner
               .
               .
               .
  100     continue
          .
          .
          .
  200continue
     .
     .
     .
  300  continue
cmic$ end parallel
```

This structure permits efficient parallel performance (Bailey et al., 1991) since the work distributed to the processors is vector.

Section 5.4 will describe STAGE-2 parallel performance with the two solvers described above, the moderately parallel vector (MPV) version created by FPP and the highly parallel scalar (HPS) version created by rearranging the loops in the block solver and parallelizing with ATSCOPE.

5.2 Autotasking the Remaining Subroutines

The other subroutines in STAGE-2 required the insertion of preprocessor directives to allow the autotasker to scope the variable properly or to enable the autotasker to parallelize a loop in spite of the data relationships it had recognized as dependencies. The table lists these actions.

TABLE 3. Actions Required to Parallelize STAGE-2

SUBROUTINE	INTERVENTION
BTRI	Modify solver for parallelism and employ FMP directives (used for the MPV version of STAGE-2).
MUTUR	Employ FMP directive on one loop
RHSI	Employ FPP directive for scoping and dependency local variables
	Employ FPP directive to assist in dependency analysis for two loops
	Employ FMP directive on one loop
RHSO	Employ FPP directive for scoping local variables
	Employ FPP directive to assist in dependency analysis for three loops
VMAT	Employ FPP directive for scoping local variables
VRHS	Employ FPP directive for scoping local variables

Several of these insertions simply consisted of informing the autotasker that an array referenced in a parallel structure used values generated only in that structure. The conservatism required in dependency analyses frequently necessitates such user directives.

An FMP directive was required to override an FPP error in subroutine MUTUR. FPP-generated code did not allow proper initialization of a private array.

About 10% of the parallel structures created by FPP involved run-time scheduling, i.e., parallel execution of a block of code required satisfaction of an IF test during program execution. The simple subscripting allowed by the zonal decomposition allows this fraction to be much lower than the 48% quoted in a test of parallel compilers (Shen et al., 1990).

5.3 Final Predicted Performance

Insertion of the directives described in the previous section led to improved dedicated speedups predicted by ATEXPERT. Table 4 lists the predictions and the asterisks denote those routines which received FPP or FMP directives.

TABLE 4. Final ATEXPERT-predicted Speedups (with -ei6 option)

SUBROUTINE	SPEEDUP
FLUX	7.89
BTRI	3.50
SMATRX	7.64
LHSO	4.51
LHSI	5.86
MUTUR*	6.90
VMAT*	7.31
RHSO*	7.43
CONTROL	2.36
GETQ	7.85
RHSI*	7.27
VRHS*	7.49
GETOLD	7.92
MUKIN	7.89
EIGEN	4.21
PUTQ	7.85
CONVRG	1.00
METRIC	7.41
DATA	1.00
GETXY	7.84
PUTOLD	7.41
PATCH	1.36
CORRECT	1.37
INITIA	7.87
INTRP	1.00
PUTXY	1.09
HISAVG	2.81

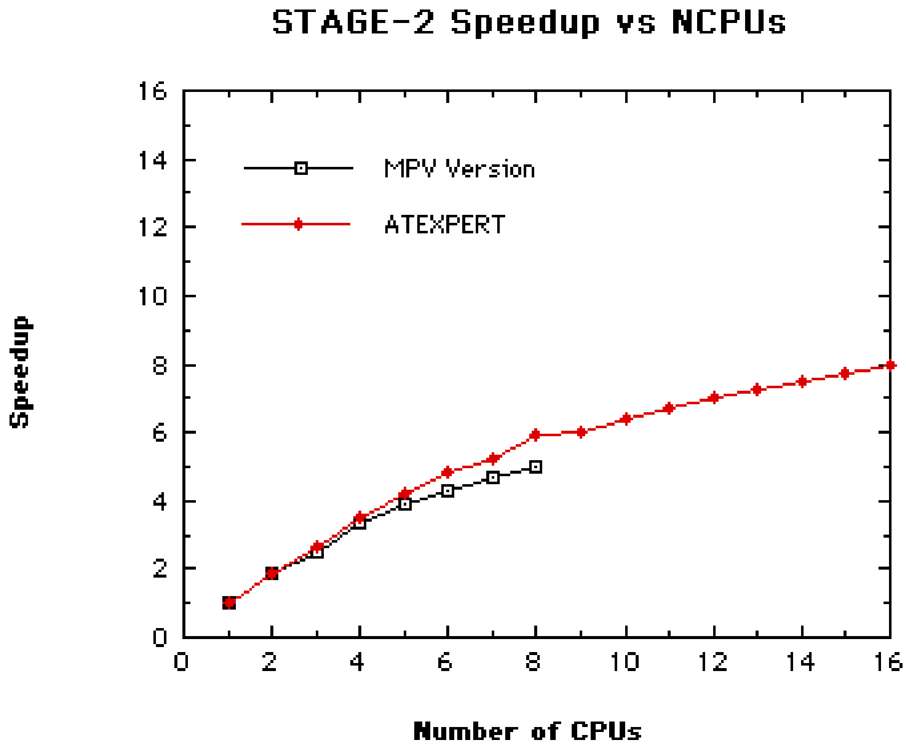
OUTPUT	1.00
HISTIM	1.48
GETSTR	1.00
MATCH	1.28
STAGE-2	1.00
REDWRT	0.10
CHECK	1.00
ENTIRE PROGRAM	5.97

All subroutines receiving increased attention displayed significant increases in parallel performance. Parallel speedups for nonasterisked subroutines in Table 4 display speedups which differ from the values in Table 2. ATEXPERT measures, on a loaded system, the elapsed time required to complete the parallel regions with various numbers of CPUs; ATEXPERT then applies an extrapolation technique to predict dedicated time speedups. The system load influences both the number of CPUs given to a parallel job and the elapsed time required to execute the parallel regions. Since the system load varies strongly throughout the day, the timing data used by the ATEXPERT algorithm will vary from one run to another. Thus, the speedups predicted for various parallel regions will change even though the parallel code remains unchanged. Additional comments regarding the variation in ATEXPERT-predicted speedups are provided in another analysis (Stockdale, 1992).

The 8-CPU estimated speedup of 5.97 is measured relative to execution to ATEXPERT's estimate of the singletasked execution time in dedicated mode. Amdahl's law for parallelism indicates that the user directives allowed the autotasker to create a code which was 85% parallel.

5.4 Final Measured Performance

Parallel performance for execution on 1 through 8 CPUs was measured in dedicated time. The figure below compares the speedup of STAGE-2 using the vector version as the base. The speedup is the ratio of the elapsed time for the code executing on 1 CPU to the elapsed time for the code executing on a specific number of CPUs. The figure also provides a plot of the ATEXPERT-predicted speedups for 1 through 8 CPUs and the speedups extrapolated by ATEXPERT to 16 CPUs.



The Moderately Parallel Vector (MPV) Version curve gives the true speedup for STAGE-2 because it is measured relative to the original singletasked version of the code. Data generated for this curve employ the FPP-produced parallel version of the block tridiagonal solver to define a 1-CPU elapsed time.

Some of the DO loops in parallel regions of the block tridiagonal solver BTRI have a natural parallelism of 4, i.e., a significant amount of the parallel calculation occurs in DO loops whose upper index is 4 and this value limits the number of processors which these sections of code can utilize efficiently. This limitation contributes to the decrease in slope (efficiency) as the number of CPUs increases beyond 4. Requests for more than 4 CPUs in BTRI tend to increase execution time due to the extra computation and synchronization; this situation is a load imbalance. BTRI's natural parallelism of 4 occurs because STAGE-2 simultaneously solves 4 equations: conservation of mass, conservation of momentum (in two dimensions) and conservation of energy.

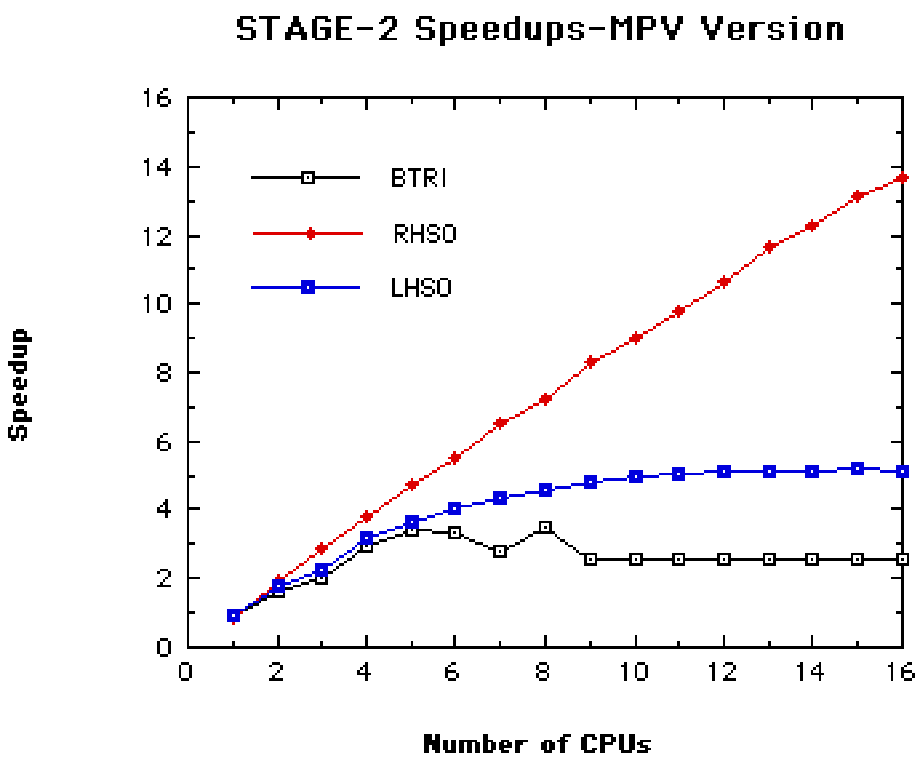
The measured STAGE-2 speedups were 3.36 for four processors and 5.00 for eight processors; the corresponding efficiencies were 84%% and 62%. Thus, execution on a smaller number of additional processors makes better use of CPU resources. The five-fold speedup on 8 CPUs reflects a code which is 91% parallel and such performance is quite reasonable for the level of effort expended (Kohn, 1989).

Task granularity varies between 0.01 and 150 milliseconds; these values are sufficiently large to amortize synchronization overhead.

For the parallel version, the Cray Hardware Performance Monitor (HPM) reported that memory conflicts increase linearly with additional CPUs, up to a value of 16% for execution on 8 CPUs. For comparison, the 8-CPU benchmark parallel version of ARC3D displays memory contention of 14%.

The figure shows that ATEXPERT-predicted speedups are slightly larger than those measured for the parallel version. ATEXPERT uses minimum parallel loop elapsed times to obtain its dedicated time speedup predictions. STAGE-2 DO loops generally use grid dimensions for loop lengths and since these values remain constant throughout program execution, the constant amount of work per iteration permits an accurate extrapolation to 8 CPUs.

ATEXPERT allows a closer examination of the sources of poor parallel performance in STAGE-2. The figure below displays speedups for three key subroutines of the MPV version of STAGE-2: BTRI, LHSO, and RHSO, as predicted by ATEXPERT.



BTRI is the FPP-produced MPV version of the block tridiagonal solver. The speedup increases through 5 processors, decreases for 6 and 7 CPUs and increases to a maximum of 3.5 on 8 CPUs. ATEXPERT predicts decreasing performance beyond 8 CPUs. This complex behavior is due to construction of the parallel regions, some having a natural parallelism of 4 and some having a much larger parallelism. While the actual performance is a blend of the performance of both types of regions, the BTRI 8-CPU speedup of 3.5 retards the STAGE-2 parallel performance.

LHSO performs the calculation of the left hand (implicit) side of the governing equations for the outer grids and RHSO performs the calculation of the right hand (explicit) side for the outer grids. LHSI and RHSI perform similar calculations for the inner grids and their performance curves are similar to LHSO and RHSO.

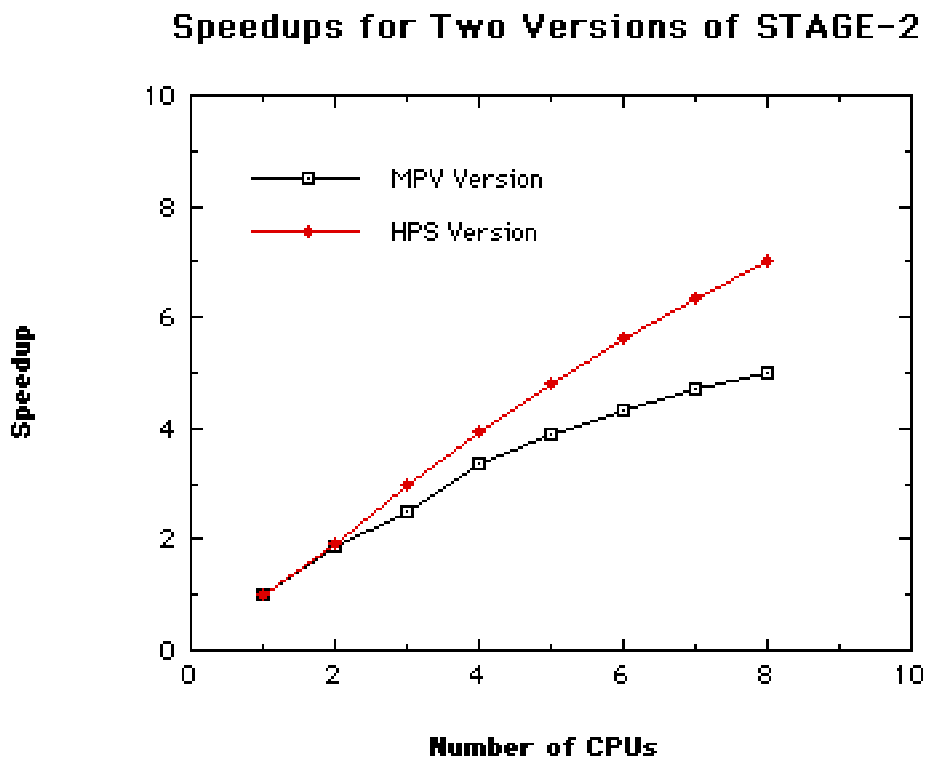
RHSO displays a near-linear speedup, a 1.27 FLOPS to memory-reference ratio, and performs at 169 MFLOPS on a single processor. RHSO computes flux contributions. ATEXPERT indicates that RHSO execution on increasing numbers of CPUs will suffer from memory contention and such contention is a source of RHSO's small departure from linear speedup.

LHSO displays an asymptotic speedup of 5.0, a FLOPS to memory-reference ratio of 0.31, and a performance of 60 MFLOPS on a single CPU. LHSO performs the data movement required to cast the left-hand side of the equations into block tridiagonal form for BTRI and such data transfer makes LHSO highly memory-intensive. HPM examination of the DO loops executing in singletasked mode indicated that 40% of the clock periods (CPs) involve memory-related instruction hold issues, with an equivalent amount of CPs spent waiting for the vector registers. The HPM further indicated that most of the memory traffic involves writing and only a small percentage of this traffic experienced memory conflicts. The HPM results mean that memory instruction holds occurred because memory ports were busy or vector registers were unable to clear quickly enough to allow transmission of new data from the memory. The limited amount of computation contained in these loops prevents calculation-related amortization of the memory delay. The autotasker-produced parallel version of LHSO consists of processes which execute identical code. The intensity of the memory traffic will produce CPU hold issues for all the processors, as ATEXPERT predicted. Moreover, the similarity of the memory access patterns should produce additional degradation due to memory bank conflicts.

This examination of STAGE-2 parallel performance indicates that two factors limit the current version: the load imbalance of the parallel solver and memory access patterns in the implicit solution subroutines.

6.0 A More Parallel STAGE-2

Improvement of the parallel performance of the block solver would strongly improve the parallel performance of STAGE-2. Section 5.1 discussed the creation of a highly parallel scalar (HPS) version of the solver with ATSCOPE. The figure below compares the speedup of STAGE-2 using the highly parallel solver.



The HPS version used the highly parallel scalar version of the solver to define a 1-CPU elapsed time. The 1-CPU parallel version of STAGE-2 requires more time to complete the problem than the vector version of STAGE-2 because the solver performs in scalar mode. Thus, speedups measured relative to the parallel version appear larger than those for the vector version. Relative to its 1-CPU parallel version, the 8-CPU HPS version displayed a 7.02 speedup with 98% of the code execution in parallel. However, execution of the HPS version on 8-CPUs required 130 seconds while execution of the MPV version on 8-CPUs required only 90 seconds. The results suggests that the block solver limitations must be overcome with a different algorithm.

A block cyclic reduction algorithm (Kumar, 1989) should generate improved parallel performance, albeit at the cost of increased memory, increased overhead for data rearrangements, and a variation in parallelism between the reduction steps. However, the code could maintain its current serial format and let the Cray autotasker generate the parallel code.

The STAGE-2 zonal approach could achieve a level of parallelism much coarser than that of the DO loop. The partition of the STAGE-2 problem into regions with varying degrees of mesh refinement resembles a Domain Decomposition (DD). Classical DD methods divide a problem, usually involving an elliptic operator, into smaller regions and paste the results together. The regions can have overlapped and/or non-overlapped boundaries and much of the DD effort emphasizes the treatment of boundaries. Domain decomposition has been quite successful in attaining high levels of parallelism, but STAGE-2 has several significant differences from typical DD applications which could strongly affect the parallel performance.

The STAGE-2 solution employs the Navier-Stokes operator in the viscous regions and the Euler operator in the inviscid regions and the nonlinearities introduced by these operators are not present in the usual DD linear operators. The STAGE-2 transfer of boundary conditions via linear interpolation differs from the more established procedures involving iterative boundary condition transfers. The STAGE-2 partition involves zones with differing numbers of mesh points and differing operator characteristics. Table 1 shows that the subroutines performing calculations for the 6 viscous zones (LHSI, MUTUR, VMAT, RHSI, VRHS, and MUKIN) require almost twice as much CPU time as the subroutines performing calculations for the 6 inviscid zones (LHSO and RHSO). This decomposition could lead to load-balancing problems for high-level parallelism. Moreover, STAGE-2 employs this decomposition to economize on central memory and promote job turnaround, as opposed to utilizing multiple processors to reduce elapsed time. Parallelism at the highest level, i.e., simultaneous solution of all grids, requires that each process have its own copy of the block solver. Conservation of scarce central memory would require redimensioning of arrays in the solver and perhaps other subroutines as well.

The clean decomposition inherent in the current version of STAGE-2 would make it a good test of high-level automatic parallelizing tool.

7.0 Conclusion

Application of the Cray Autotasker to the two-dimensional CFD code, STAGE-2, has produced a parallel version which displays an 8-CPU speedup of 5.0 relative to the vector version. The Cray Autotasker was able to generate a version of the code which was 91% parallel. However, examination of code performance with a CRAY utility, ATEXPERT, indicated two limiting factors inherent in the code structure; an automatic parallelizing tool could not be expected to reduce the impact of these factors.

One factor limiting the parallel performance of the code is the block tridiagonal solver. The moderately parallel vector version created by FPP displayed a severe load imbalance and the poor utilization of the CPUs by this routine degraded STAGE-2 performance. Creation of a highly parallel scalar (HPS) version of this solver with ATSCOPE allowed STAGE-2 to obtain a speedup of 7.02 on 8 CPUs. While data dependencies forced the HPS solver to operate in scalar mode and required an increased execution time for the highly parallel STAGE-2, this example does illustrate the potential benefits of a solution algorithm (such as block cyclic reduction) designed for parallelism.

A second factor limiting code parallel performance is the high intensity memory traffic in subroutines performing the implicit calculation.This traffic occurs as these routines prepare arrays for the block solver.

Manual intervention was required to assist the autotasker in scoping variables and recognizing dependencies. Such required action suggests that this code is a good test of the analytical capability of a low-level parallel preprocessor. The high level zonal decomposition of the problem led to a simple subscripting of array references and facilitated FPP dependency analysis. This decomposition also enabled much of the computation to be performed in easily parallelizable double DO loops suggests that STAGE-2 would be an excellent test program for a high-level automatic parallelization tool.

8.0 References

Bailey, D. H., et al. 1991. "The NAS Parallel Benchmarks-Summaries and Preliminary Results,"
In Proc. Supercomputer '91, pp 158-165.

Cray Research Inc. 1988. *Cray Y-MP and Cray X-MP Multitasking Programmer's Manual*, Pub. No. SR-022E, Cray Research Inc.,1988.

Cray Research Inc. 1991. *UNICOS Performance Utilities Reference Manual*, Pub. No. SR-2040 6.0, Cray Research Inc.,1991.

Gundy, K., et al. 1989. "Two-dimensional Computations of Multistage Compressor Flows Using a Zonal Approach," AIAA-89-2452, Monterey CA.

Isaacson, I. and Keller, H. B. 1969. *Analysis of Numerical Methods*, p. 58, Wiley, New York/London/Sydney, 1969.

Karp, A. and Flatt, H. P. 1989. "Measuring Parallel Processor Performance," Communications of the ACM, 33, 5 (May 1990) 539-543.

Kumar, S.P. 1989. "Solving Tridiagonal Systems on the Butterfly Parallel Computer,". International Journal of Supercomputing Applications, 3 (1) Spring 1989, pp 75-81.

Kohn, J. 1989. "Autotasking Performance Expectations,". Proceedings of the Cray User Group, October, 1989, Bologna.

Pulliam, T. H. and Chausee, D. S. 1981. "A Diagonal Form of the Implicit Factorization Algorithm," Journal of Computational Physics, 39 (1981) 347-363.

Stockdale, I. E., 1992. "Conversion of Program GAIL2D into a Parallel Code," forthcoming NAS Technical Report RND-92, Ames Research Center, Moffett Field, CA, 1992.

Shen Z., et al. 1990. "An Empirical Study of Fortran Programs for Parallelizing Compilers," IEEE Transactions on Parallel and Distributed Systems, 1,3 (1990) pp 356-364.