Autotasked Performance of a NASA CFD Code
RND 90-003

Russell Carter

Computer Sciences Corporation
NASA Ames Research Center
Moffett Field, CA 94035, USA

1. Introduction

Current progress in the field of Computational Fluid Dynamics (CFD) is often limited by the speed that existing computers and algorithms compute solutions to various incarnations of the Navier-Stokes equations. One approach to increasing the rate of computation is to exploit algorithmic independence or parallelism in the calculation. Current generation supercomputers such as the Cray Y-MP 832 located at NASA AMES Numerical Aerodynamic Simulation Facility (NAS) provide several methods of exploiting parallelism. These methods differ in the amount of user sophistication required to achieve adequate results. In this report, the performance of an automatically parallelized, or autotasked, NASA CFD code is studied. Two aspects of the code's performance were assessed: unitasked vectorized performance, and performance when autotasked in dedicated time.

1.1 Code Description

The code chosen was SPARK, which is an implementation of a numerical model for supersonic reacting mixing layers as detailed in [1]. SPARK has been applied to problems associated with the propulsion systems of the National Aerospace Plane. The code used for this study solves the two-dimensional Navier-Stokes equations coupled to a two-species chemical reaction problem. The program was designed to consider the multicomponent diffusion and convection of important species in the chemical reactions, and the interactions between fluid mechanics, chemistry and thermodynamics. Specifically, the kinetics of the chemical reactions are incorporated into the simulation and in fact their computation constitutes the bulk of the computational expense. The configuration of SPARK used in this study solves a two-species model in the same fashion as the original code.

The governing equations are discretized using a temporally implicit scheme solved by a modified MacCormack technique. The FORTRAN source consists of approximately 5000 lines of code. Under UNICOS 5.0 the executable size of the unitasked job was 6.6 MW and the executable size of the autotasked job was 6.8 MW. Under UNICOS 5.0.12 the executable size of the unitasked job was 6.6 MW and the executable size of the autotasked job was 7.5 MW. Normal unitasked execution requires approximately 300 CPU

seconds. Since the algorithm is time stepping in nature, for the purposes of this study a shorter execution time of approximately 75 CPU seconds was chosen to allow reasonable turnaround time with the available batch queues.

1.2 CRAY Y-MP Computer System Description

The CRAY Y-MP, serial number 1002, is an eight CPU, multiple instruction multiple data stream (MIMD) computer system. The clock period (CP) is 6.33 nanoseconds. The peak computation rate based on one addition and one multiplication per clock cycle is 316 MFLOP/sec. The relatively fast 32 MW shared main memory is augmented by a slower 256 MW solid-state storage device (SSD)[2].

The operating system (UNICOS) is UNIX System V based. Support exists for multiprogramming, where different processors run different jobs, and *multitasking*. Multitasking allows multiple processors to execute two or more parts of a single program in parallel and is discussed in detail in Section 3.1. FORTRAN codes are by default run with full vectorization on a single processor, or *unitasked*. A *job* is a single program that may or may not spawn multiple processes. Jobs may be run in batch or dedicated mode. Batch mode jobs run in a multiprogramming environment concurrently with other batch jobs and interactive users. The performance of a code running in a multiprogramming, multiprocessing environment strongly depends on the availability of system resources such as processors, disks, or memory. Jobs run in batch mode on the Y-MP under normal system workloads compete for resources with other jobs and are subject to being swapped to disk which adversely affects wall clock execution time performance compared to programs run in dedicated time. Dedicated mode jobs run interactively with minimum competition from system and user activity and hence obtain the best performance possible on the CRAY Y-MP.

1.3 Performance Evaluation Tools.

Next, tools are discussed that may be used to facilitate the parallelization and evaluate the performance of a FORTRAN program on the Y-MP. Additional details may be found in [3].

The Y-MP hardware performance monitor (hpm) command *hpm* summarizes the machine performance of a program by reporting various hardware counter statistics. Statistics are arranged by groups. Each group reports a set of related statistics on aspects of the program's performance. Group 0 is an execution summary which reports such statistics as instructions issued, I/O and CPU references, and floating point additions, multiplications, and reciprocals. Group 1 reports on various hold issue conditions. Group 2 summarizes memory activity, while Group 3 reports vector events and an instruction summary. Preparation involves normal compilation of the program source which is then run with the *hpm* command on the command line in the same manner as the UNIX *time* command. *hpm* works with multitasked programs. A sample of *hpm* output is provided as Appendix 1.

The *perftrace* utility extends the *hpm* to individual program blocks such as subroutines and functions. Preparation involves compilation with the *flowtrace* option turned on and explicit linking in the *perftrace* libraries. *perftrace* fails on multitasked programs.

opc is a local utility that provides both floating point operation (FLOP) count and FLOP rate data at the program and subprogram level for serial Cray FORTRAN codes. The program explicitly counts the FLOPs in each line of source code and the number of times that line of source code would be executed. Since *opc* counts by inserting a unique indexing statement before each line of source, it works correctly only if the compiled code is executed as the source code is written. Thus, it is less reliable on poorly written source codes that can be significantly optimized by the cft77 compiler.

The *ja* command is a UNICOS utility that provides accounting information on program runs. Useful statistics include elapsed time, user CPU time, system CPU time, concurrent CPUs, and average concurrent CPUs.

2. Unitasked Performance

2.1 Vectorization Analysis

Output from *hpm* indicated that the SPARK code run with default vectorization, i.e., with the command *cf77 spark.f -o spark*, attained a sustained computation rate of 177 MFLOP/sec. This is about 56 percent of the peak single processor rate (316 MFLOP/sec). The unitasked performance of a CFD code on the Y-MP is strongly dependent on the amount of vectorizable work in the program. Hence the source code was examined for characteristics that enhance vectorization and thus performance. The version of SPARK studied performs no file input and and relatively little file output, so I/O effects were insignificant. Execution control flow consists of 200 time step iterations for the 75 CPU second version. During each iteration, subprograms are called between one and four times. No user subroutine is called more than four times an iteration, hence total subroutine call overhead is relatively low. The code's computational work consists of large numbers of iterations of DO loops which have no data dependencies. There are a total of 491 DO loops in the code. Of these, 216 are of the unnested form *Type 1*:

```
DO I = 1 , IXJ
     work
CONTINUE
```

where `IXJ` = 1118

Of the remaining loops, 79 are of the nested form *Type 2*:

```
DO K = 1 , NCS
     DO I = 1 , IXJ
          work
     CONTINUE
CONTINUE
```

where NCS = 9

Inspection of the cft77 listing file showed that these loops compile completely to long vector loops, indicating no dependency conflicts. *perftrace*, *flowtrace* and *opc* each reported that approximately 72% of the cpu time is concentrated in four code blocks, with none of the remaining blocks consuming more than 3.5% each of the total cpu time. Execution times of loops in the top four code blocks were timed by inserting calls to the *real time clock* . The times show that six loops account for 56.5% of the execution time of the code. These six loops were all of the forms listed above. Many of the long vector loops in the four most cpu time intensive subroutines consist of a reasonably balanced mix of vector multiplications and additions, which enhances vector chaining [4][5].

Output from the *hpm* was used to quantify the unitasked performance of the code. It is apparent from the output that SPARK is highly vectorized. For example, the ratio of vector to total floating point operations was 0.9994. The Group 1 hold issue summary indicates SPARK spent 38.84% of the clock periods waiting on vector registers and 51.48% of all clock periods waiting on vector functional units. The relatively high percentage of time waiting on both vector registers and functional units is an indicator of simultaneous use of functional units. The relatively low percentage of clock periods waiting on block memory references (9.01%) indicates there was a relatively low number of memory bank conflicts. The Group 2 memory activity summary shows a ratio of scalar memory references to block memory references of 5.9E-3. All these statistics indicate efficient, highly vectorized code.

2.2 Calculation of Percent Vectorization

Percent vectorization is defined as the percent of vector operations in the total of operations generated over the course of a program execution. Total operations is the sum of vector operations and scalar operations. The estimation of percent vectorization from hpm data is strongly dependent on the choice of what quantities to count as either vector or scalar operations. Three groups of performance data recorded by the hpm that can be classified as either vector or scalar are floating point operations (FLOPs), memory references, and instructions. The proportion of vector to scalar data in each group differs. For example, the proportion of vector FLOPs to scalar FLOPs differs markedly from the proportion of vector instructions to scalar instructions since vector instructions may result in as many as 64 times as many operations as scalar instructions. FLOPs can be subdivided into floating point additions, multiplications, and reciprocals.

Method 1 of estimating percent vectorization is to calculate the percent of vector FLOPs in the total FLOPs, as was reported in the previous section. Percent vectorization thus estimated neglects the memory and housekeeping operations present in all codes. Method 1 produces a percent vectorization of 99.94 percent. *Method 2* adds memory references plus integer and logical operations to the total number of operations. Details are provided in Appendix 2. The percent vectorization estimated by Method 2 is 97.0 percent.

3. Autotasked Performance in Dedicated Time

3.1 Parallelization Techniques on the Cray Y-MP

Three types of multitasking are implemented on the Y-MP: macrotasking, microtasking, and autotasking [6][7]. *Macrotasking* is the process of dividing a code or algorithm into sequential pieces (called tasks) that can be performed in parallel. This division may involve a simple partition of an existing serial algorithm's independent sequential components or it may involve a complete implementation of a purely parallel algorithm. Tasks are explicitly distributed among available processors. Explicit calls to the macrotasking libraries provide task initiation, synchronization, protection of critical code segments, and communication. Tasks typically have relatively high complexity (large granularity) and are usually structured as subroutines. FORTRAN code written to exploit macrotasking on the CRAY Y-MP is not portable. Since the design and implementation of the parallel solution to a problem is explicitly done by the programmer, programmer overhead is high.

Microtasking permits multiple processors to operate on an existing serial program without affecting portability. The level of parallelism may vary from large tasks involving the simultaneous execution in large subprograms to very small tasks executing single iterations of do loops. Task initiation, synchronization, and protection of critical code segments are provided for by compiler directives manually inserted into the FORTRAN source code. These directives appear as FORTRAN comment lines and do not affect the portability of the original code. However, the programmer overhead issues raised with macrotasking still occur if the data dependency analysis is complex. Subtle, difficult to trace errors can arise from the parallel execution of blocks of code originally written to be executed serially if the dependency analysis manually performed by the programmer is faulty. To some extent the dependency analysis may be automated by processing the modified source code with manually inserted microtasking directives through the FORTRAN preprocessor and allowing it to do the dependency analysis. This aspect was not examined in this report.

Very good performance has been obtained with macrotasked and microtasked FORTRAN codes on the NAS Y-MP [8].

Autotasking is a specialized and automated adaptation of microtasking. The basic idea is the automatic parallelization of independent DO loops.

FORTRAN source code is first processed through *fpp*, the FORTRAN preprocessor. *fpp* performs a data-dependency analysis and looks for parallelism within the code on a DO-loop level and inserts appropriate autotasking compiler directives into the FORTRAN source code. Autotasking compiler directives are similar in syntax and function to microtasking directives. The output of *fpp* is then processed by *fmp*, the FORTRAN mid processor, which uses the autotasking directives to restructure the code for parallel execution. The output of *fmp* is FORTRAN source code with machine-dependent library calls and compiler functions imbedded in the source. This output is translated by the *cft77* compiler to machine code. Loading with *segldr* produces an autotasking executable. Alternatively, the three step process may be automated by a single call to the *cf77* command which takes the original FORTRAN source input and produces an autotasking executable. Since the programmer potentially has minimal contact with the parallel aspects of the code, programming overhead is low. Versions 3.1 of *fpp*, *fmp*, and *cft77* were used in this study.

An autotasked job is structured as a set of sequential and parallel regions. Within sequential regions, a *master task* executes the sequential work while *slave tasks* associated with the job are idle, or *parked*. Within parallel regions, both master and slave tasks execute available work. The number of processors actually allocated to a parallel region is determined from the number of CPUs requested by the job initially, and the amount of demand on the system. The number of processors requested may be specified at runtime by setting the UNICOS shell variable *NCPUS*. The number of processors allocated will not exceed the number requested by the parallel job. On the other hand, normal demands on the system have the effect that the number of processors actually obtained is significantly less than the number requested. However, during dedicated time, system load is minimal, and actual numbers of processors obtained is close to the number requested.

3.2 UNICOS statistics on Parallel Jobs

The UNICOS operating system reports a number of statistics that summarize the runtime performance of the parallel code. The most important of these are the time statistics produced by *ja* and *hpm* .

Several interrelated time quantities are reported by the system for each UNICOS job . The following definitions have units of seconds. Connect time to *i* concurrent processors T_{ci} is the elapsed time *i* processors were concurrently attached to the job (either executing or idle) and is obtained from the *ja* summary report. Total connect time T_{ct} (also denoted total CPU time) is the sum of the T_{ci} weighted by the number of concurrent processors. Assuming *n* processors were requested for the autotasked job, T_{ct} is calculated from

$$T_{ct} = 1*T_{c1} + 2*T_{c2} + ... + n*T_{cn}$$

The average number of concurrent processors over the duration of the job *p*, is then computed from

$$p = \frac{T_{ct}}{(T_{c1} + T_{c2} + \ldots + T_{cn})}$$

T_p is the elapsed (wall clock) runtime for a job executing with *p* concurrent processors. Wait semaphore time T_{ws} is the total of all the individual CPU execution times that all connected CPUs spent waiting for a semaphore (waiting at a synchronization point). It is not simply related to total execution time since concurrent processors waiting for a semaphore are simultaneously accumulating wait semaphore time. Wait semaphore time is obtained from the hpm group 1 output.

User CPU time T_{usr} is the difference between total connect time T_{ct} and the wait semaphore time T_{ws}.

$$Tusr = Tct - Tws$$

System CPU time T_{sys} is essentially UNIX system kernel work attributable to a job. System CPU time is obtained from the *ja* command and summary reports. Since the kernel is shared by all processes, some work of the kernel is not attributable to a particular process and relevant charges are distributed among all active processes. Similarly, some system work attributable to a job is not charged to the job's system CPU time. Usually, for serial programs T_{sys} represents less than one percent of the total time.

3.3 Parallel Characteristics of the SPARK Code

Recall that the bulk of the SPARK code's computational expense consists of large numbers of iterations of independent vector loops. At the lowest (DO loop) level, parallelism can be introduced by simply computing iterations in parallel (DO ALL). An important consideration is the conflict of vectorization vs. parallelism. Highly vectorized code is typically ten to twenty times faster than scalar code. On the other hand, under ideal conditions, the speedup for a parallel code on an eight processor system such as the Y-MP is less than or equal to eight. Therefore, on a vector computer system with a small to moderate number of processors, overall vectorization should not decrease as a result of introducing algorithmic parallelism [7].

For the simple model of parallelism given above, the implementation of SPARK is ideal. Iterations of Type 1 DO loops can be divided into chunks of not less than 64 (with one chunk of size less than 64). Similarly, each outer loop of Type 2 loops can be considered a single task which executes a fully vectorized inner loop. In both cases, vectorization of the code is not decreased by the introduction of parallelism.

Parallelizing a code introduces system overheads that in total may exceed the amount of elapsed run time saved by adding more processors to the work, particularly if vectorization is decreased by the additional parallelism. In this case, the execution time for the code will increase. *fpp* by default analyzes the amount of computational work contained in a loop and performs a threshold analysis based on the number of loop iterations on whether or not to autotask a loop. Often, loop iteration bounds are determined dynamically at runtime. In this case *fpp* inserts runtime threshold tests of the loop iteration bounds so that a loop is autotasked for sets of iteration bounds that meet the criteria.

Loops in the four most computationally expensive subroutines were examined for failure of the threshold tests inserted by *fpp*. All of the loops passed the imposed test.

3.4 Dedicated Time Results

The SPARK code was first run with default autotasking (*cf77 -Zp*). Results were disappointing; the code ran at 199.3 MFLOPS with a speedup of 1.13. Speedup is defined to be the elapsed run time on one processor T_1 divided by the elapsed run time on p processors T_p, where p is slightly less than eight on the eight processor Y-MP. The preprocessors *fpp* and *fmp* have options that allow various modes and levels of optimization to be applied to the source code. Inspection of the listing file produced by *fpp* revealed that all of the Type 1 loops defined above were not multitasked and were computed solely as unitasked vector loops. *fpp* by default examines only nested sequences of loops. Outer loops are microtasked; inner loops are vectorized. Single loops are considered to be inner loops. After specification of the command line option to enable inner loop autotasking (*cf77 -Zp -Wd,-ei*), examination of the *fpp* output file showed that all Type 1 and Type 2 loops were correctly autotasked. *fpp* implements automatic inline expansion of called subroutines. Preprocessor options control the level of inlining of code. Level 6 is claimed to be always safe, level 7 is "rarely unsafe". Level 6 was used to test the code. The code was then run in dedicated time with fpp options to enable inner loop autotasking and level 6 subroutine inlining (*cf77 -Zp -Wd,-ei6*). With these options the code ran at 832.7 MFLOPS, a speedup of 5.0. These options were used to generate the remainder of the data presented in this report.

3.5 Theoretical Speedup Calculation

The relatively simple approach which Cray autotasking takes to the parallelization of FORTRAN codes, i.e., parallelization of DO loops, suggests that information about the level of vectorization of the original unitasked code may be used to predict the dedicated time performance of the autotasked code. Of interest is the calculation of the best theoretical speedup S_p. The speedup calculation uses a variant of Amdahl's law. The key assumptions are that all DO loops in the original source vectorize completely and *fpp*

transforms all vector loops to parallel vector loops. Manual inspection of the listing file produced by *fpp* from the SPARK FORTRAN source showed this was a reasonable approximation. Finally, it is assumed that no scalar code is executed in parallel, and that vector code executes R_v times faster than scalar code. Vector operations on a Cray are performed at the highest rate when the vector length is greater than about 64. Vectorized code is not executed 64 times faster than sequential code, however, due to various factors such as memory contention and vector startup overheads. Observed R_v's range from about 10 to 20 for the Y-MP.

The execution time for FORTRAN code run on a single processor that is to be run in parallel on multiple processors can be described as the amount of time spent in the serial portion plus the amount of time spent in the parallelizable portion of the code. Then the fraction *f* of the overall sequentially executed program CPU time (T_{usr}) spent in the parallelizable section (also sequentially executed) is given by:

$$f = \frac{P}{(P + S)} \qquad (1)$$

where

P = CPU time spent executing sequentially in the parallelizable section of the code,
S = CPU time spent executing in the serial section of the code.

Note that f, thus defined, is the fraction of parallelizable code in the program. The assumption is made that P and S are computed from

$$S = S_T * T_{sop}$$
$$P = V_T * T_{vop}$$

where

S_T = total number of scalar operations
V_T = total number of vector operations
T_{sop} = average time for one scalar operation
T_{vop} = average time for one vector operation.

The relationship between T_{sop} and T_{vop} is expressed as

$$T_{sop} = T_{vop} * R_v$$

Substitution into equation (1) gives

$$f = \frac{V_T}{(R_v * S_T + V_T)} \qquad (2)$$

Let η be the fraction of the total operations in the code that are vectorized, that is, η is the percent vectorization, described in Section 2.2, divided by 100:

$$\eta = \frac{V_T}{(S_T + V_T)}$$

Then (2) can be written equivalently as

$$f = \frac{\eta}{((1 - \eta) R_v + \eta)} \qquad (3)$$

Equation (3) directly relates the fraction of vectorized operations to the fraction of time in the parallelizable section of the code.

Both approximations, Method 1 and Method 2 (see Section 2.2) for percent vectorization assume operations are not overlapped. In practice, operations can be overlapped between functional units. For example, *chaining* a vector through the vector add, vector shift, and vector logical functional units results in simultaneous execution within the three functional units. The ratio of vector operation execution speed to scalar operation execution speed R_v varies widely, but is most often between 10 and 20 [9]. R_v is taken to be 10 in this calculation. Approximating percent vectorization by Method 1 produces an f of 0.994. Similarly, approximating with Method 2 yields an f of 0.76.

A variant of Amdahl's Law [7] is used to calculate the theoretical speedup:

$$S_p = \frac{1}{(1 - f) + \frac{f}{p}} \qquad (4)$$

Where

S_p = theoretical speedup,
p = number of concurrently executing processors, and f is defined in (2).

3.6 Comparison to Experimental Data

The validity of this model for parallelism was investigated by comparison of the results from dedicated time runs with the theoretical speedup. The theoretical speedup uses the fraction of parallelizable code f calculated using the percent vectorization approximated by Methods 1 and 2.

The number of concurrently executing processors p is taken to be the average number of concurrent CPUs as reported by *ja*. The dedicated time runs used the NCPUS variable to vary the number of processors requested from 2 to 8. The observed speedup S_{obs} was computed relative to a unitasked job run in dedicated time, i.e.,

$$S_{obs} = \frac{T_1}{T_p}$$

where

T1 = elapsed time on one processor
Tp = elapsed time on p processors.

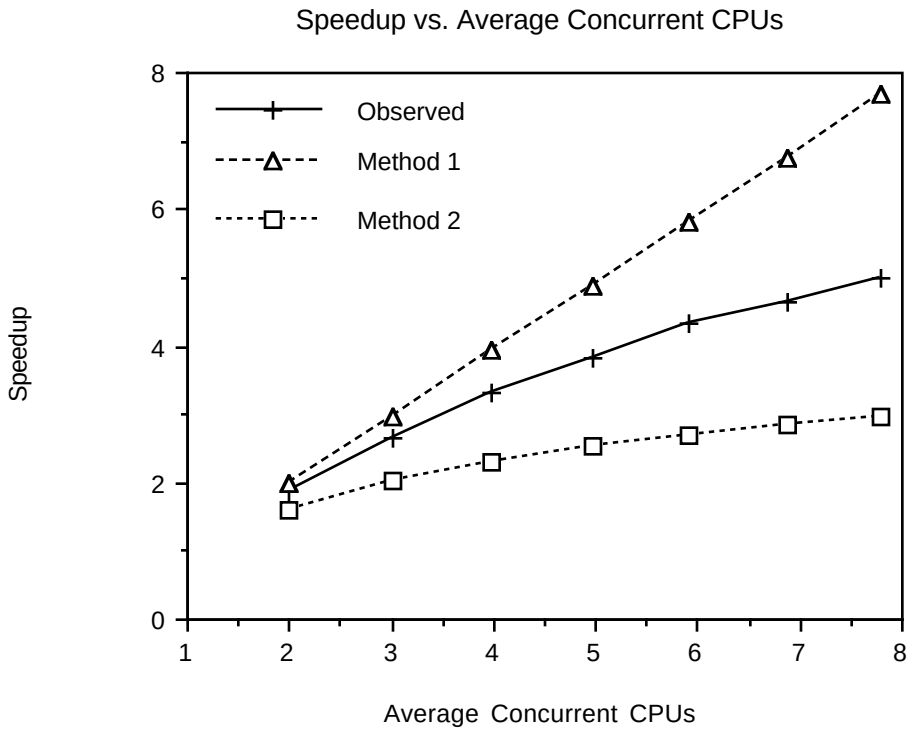
Results from the fastest individual runs in dedicated time are given below in Table 1. Data from *ja* and *hpm* for these runs may be obtained upon request from the author. The performance for a constant number of requested processors was observed to vary about five percent from run to run. The NCPUS column lists the number of concurrent CPUs requested for each run. The p column lists the number of concurrent CPUs actually obtained for each run. $S_p(1)$ and $S_p(2)$ correspond to the theoretical speedup calculated using the value of f obtained when the percent vectorization is approximated by Method 1 or 2, respectively. A plot of the speedups S_{obs}, $S_p(1)$ and $S_p(2)$ follows as Figure 1.

Table 1

Autotasked Speedups of the SPARK Code

NCPUS	p	S_{obs}	$S_p(1)$	$S_p(2)$
2	2.00	1.88	1.99	1.61
3	3.00	2.66	2.96	2.03
4	3.99	3.31	3.93	2.32
5	4.96	3.82	4.88	2.54
6	5.92	4.33	5.83	2.71
7	6.87	4.64	6.76	2.85
8	7.80	5.00	7.68	2.96

Figure 1



3.7 Discussion

The experimental data approximately bisects the region defined by the curves produced for the theoretical speedups. Both methods fail to predict the observed speedup accurately. Method 1 overestimates the speedup while Method 2 underestimates the speedup by a similar amount. Method 1 uses only FLOPs to determine the potential parallel fraction in the code, and so entirely neglects the effects of parallelization on the housekeeping functions required in any code. Hence Method 1 is likely to be a strong upper bound on the amount of speedup that can be obtained in a FORTRAN code by autotasking. The relatively large amount of difference between Method 1's speedup prediction and the speedup obtained suggests that more parallelism might be exploitable in the SPARK code.

Method 2 assigns the hpm scalar functional unit counts to the sequential fraction of the code. The scalar functional unit produces 78 times as many operations as scalar floating point for the SPARK code. These operations dominate the sequential fraction of the code. However, the scalar functional unit processes many of the integer and logical operations generated by housekeeping functions associated with the parallelizable fraction of the code, i.e., the DO loops. Hence many of the scalar functional unit counts, though not vectorizable, are incorrectly assigned to the

sequential fraction of the code. This implies that the predicted speedup should underestimate the observed speedup (as is observed).

Equation (4) may be used to compute an "observed f" f_{obs} for each p and S_{obs}. The average value of f_{obs} obtained from the seven data points in Table 1 was f_{obs}=0.927. This compares with f=0.994 from Method 1 and f=0.76 from Method 2.

3.8 Overhead

An autotasked code incurs unavoidable overheads that are manifested as increases to both user CPU time, T_{usr}, and system CPU time, T_{sys}. These overheads increase as the number of concurrent CPUs is increased. Table 2 lists the overheads measured for the SPARK code at differing numbers of concurrent CPUs. (The last entry in Table 2 differs from the last entry in Table 1 because the hpm group specified for the entry in Table 1 did not subtract wait semaphore time from the total execution time. The entries in Table 2 were run with hpm group 1 so that wait semaphore time is correctly deducted from T_{usr}.)

Table 2

Autotasking Overheads Relative to NCPUS=1
(Dedicated Time)

p	O_{usr} (percent)	O_{sys} (percent)	$O_{usr+sys}$ (percent)
1.00	0.00	0.00	0.00
2.00	1.21	56.71	1.27
3.00	1.97	122.60	2.10
3.99	2.53	217.12	2.75
4.96	3.13	1000.55	4.13
5.92	4.00	1323.01	5.32
6.87	4.74	1633.97	6.37
7.68	5.75	3129.59	8.87

The column headed by p is the average number of concurrent processors for the run as reported by *ja*. O_{usr} is the percent overhead of T_{usr} at the given number of concurrent CPUs, relative to the elapsed time for a run with one CPU. That is,

$$O_{usr} = \frac{T_{usr}(p)-T_{usr}(1)}{T_{usr}(1)} * 100$$

where

$T_{usr}(p) = T_{usr}$ with p concurrent CPUs
$T_{usr}(1) = T_{usr}$ with one CPU.

Similarly, O_{sys} is the percent overhead of T_{sys} at the given number of concurrent CPUs, relative to the elapsed time for a run with one CPU, i.e.,

$$O_{sys} = \frac{T_{sys}(p) - T_{sys}(1)}{T_{sys}(1)} * 100$$

where

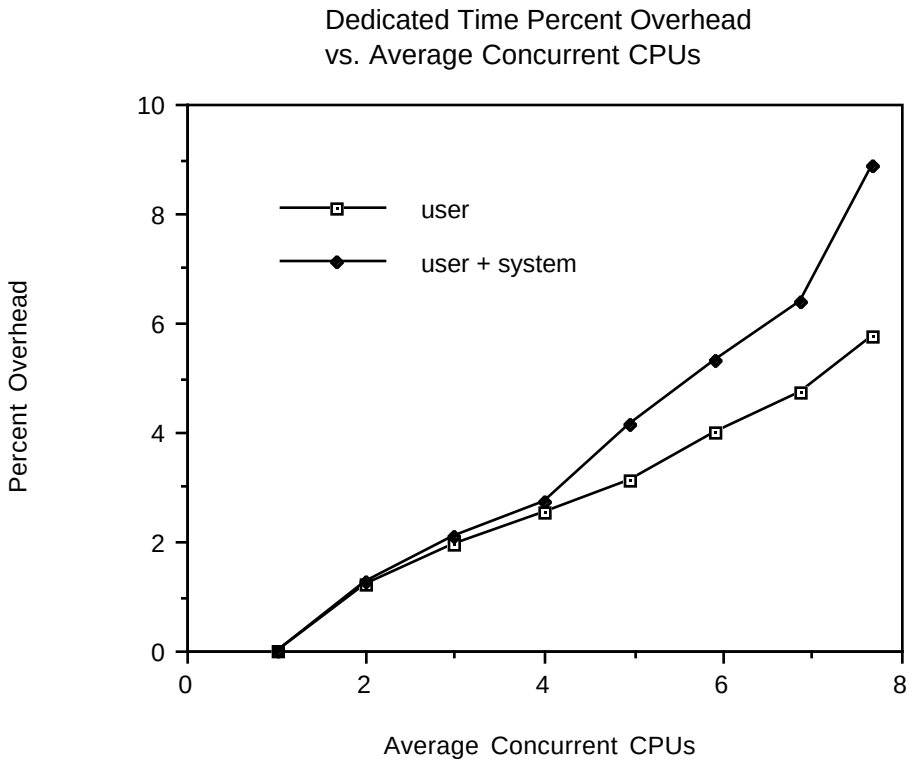
$T_{sys}(p) = T_{sys}$ with p concurrent CPUs.
$T_{sys}(1) = T_{sys}$ with 1 CPU.

The final column lists the total percent overhead of user and system time relative to the run with one CPU:

$$O_{usr+sys} = \frac{T_{usr}(p)+T_{sys}(p)-T_{usr}(1)-T_{sys}(1)}{T_{usr}(1)+T_{sys}(1)} * 100.$$

O_{usr} and $O_{usr+sys}$ were plotted versus the average number of concurrent CPUs in the following graph.

Figure 2.



System CPU time increases at a much greater rate than user CPU time. Although system CPU time is initially negligible (0.1%) when p=1, it comprises 3% of the total CPU time when p=7.68.

4 Summary

The performance of the single tasked, vectorized code was 177 MFLOPs, or about 56% of the Y-MP peak computation rate of 316 MFLOPs. The code was autotasked using the inner loop and obtained a maximum computation rate of 832.7 MFLOPs. The amount of data parallelism present in the code was expressed as a function of percent vectorization. Two methods of estimating the percent vectorization were used; each is an approximation which relies on different specific assumptions about the distribution of operations counted by the hardware performance monitor. The two methods produced percent vectorizations of 99.4 and 97.0 percent.

The percent vectorization was used to estimate the fraction of parallelizable code. The average fraction of parallelizable code observed was 0.927, which compares with 0.994 and 0.76 obtained from the two methods of percent vectorization estimation. These estimates coupled with a variant of Amdahl's law were used to predict the speedup. The predicted speedup was compared to actual speedups observed in dedicated time on the Cray Y-MP. Neither method adequately predicts the observed speedup, but it is important to note that the two methods predict speedups which bracket the observed speedups.

Overheads for running the code in parallel were measured and found to range from as little as 1.3% of the total work when running in parallel on two processors, to 8.9% when running on eight processors. Operating system overhead, insignificant at low numbers of processors, was found to increase at a very high rate as the number of processors increased. This suggests that for numbers of processors greater than ten, system overhead would become the dominant overhead, at least in a system configured like the Cray Y-MP.

5 Acknowledgment

The author wishes to thank Duane Carbon, Douglas Pase and Robert Bergeron for their constructive comments on this work.

References

[1] J.Philip Drummond, R.Clayton Rogers, and M. Yousuff Hussaini, "A Numerical Model For Supersonic Reacting Mixing Layers.", *Computer Methods in Applied Mechanics and Engineering*, 64 1987.

[2] Cray Research Inc., *CRAY Y-MP Computer Systems Functional Description Manual*, Pub No. HR-4001, 1988.

[3] Cray Research, Inc., *UNICOS Performance Utilities Reference Manual*, Pub. No. SR-2040 B, 1989.

[4] Hui Cheng, "Vector Pipelining, Chaining, and Speed on the IBM 3090 and Cray X-MP." *IEEE COMPUTER*, September, 1989, pp. 31-46.

[5 R. A. Fatoohi, "Vector Performance Analysis of Three Supercomputers: CRAY-2, CRAY Y-MP, and ETA10Q.", NASA Internal Report.

[6] Cray Research, Inc., *CRAY-YMP and CRAY-XMP Multitasking Programmer's Manual*, Pub. No. SR-022E, Cray Research, Inc.1988, pp. 2-15 - 2-17.

[7] Cray Research, Inc., *Autotasking Users Guide*, Pub. No. SN-2088, Cray Research, Inc.1988, pp. 2-15 - 2-17.

[8] R. A. Fatoohi, "Multitasking on the Cray Y-MP: An Experiment with A 2-D Navier Stokes Code.", NASA Internal Report.

[9] Cray Research, Inc., *Autotasking Users Guide*, Pub. No. SN-2088, Cray Research, Inc.1988, pp. 2-22.

Appendix 1
Hardware Performance Monitor Output

The following is the hpm groups 0, 1, 2, and 3 output for the unitasked SPARK code. The additional vector logical unit was turned off via the */etc/special -avloff* command so that the correct vector floating point operation counts were obtained.

Group 0: CPU seconds	:	72.20	CP executing	:	11408382236
Million inst/sec (MIPS)	:	16.22	Instructions	:	1171291071
Avg. clock periods/inst	:	9.74			
% CP holding issue	:	85.95	CP holding issue	:	9805871212
Inst.buffer fetches/sec	:	0.01M	Inst.buf. fetches	:	700748
Floating adds/sec	:	64.80M	F.P. adds	:	4678931278
Floating multiplies/sec	:	98.38M	F.P. multiplies	:	7103157910
Floating reciprocal/sec	:	13.38M	F.P. reciprocals	:	965757209
CPU mem. references/sec	:	107.76M	CPU references	:	7780825360
I/O mem. references/sec	:	0.24M	I/O references	:	17648903
Floating ops/CPU second	:	176.55M			
Floating ops/wall second	:	114.68M	CPU/wallclock time ratio:		0.65

Group 1: CPU seconds :	72.27937	CP executing:	11420345986
Hold issue condition		% of all CPs	actual # of CPs
Waiting on semaphores	:	0.00	475
Waiting on shared registers	:	0.00	0
Waiting on A-registers/funct. units	:	2.54	290135225
Waiting on S-registers/funct. units	:	2.25	256954420
Waiting on V-registers	:	38.84	4435353185
Waiting on vector functional units	:	51.48	5879202927
Waiting on scalar memory references	:	0.27	30621091
Waiting on block memory references	:	9.01	1028567088

Group 2: CPU seconds: 72.23268 CP executing: 11412969521

Inst. buffer fetches/sec	:	0.01M	total fetches	:	702585
			fetch conflicts	:	2384992
I/O memory refs/sec	:	1.42M	actual refs	:	102895681
% having conflicts 49.87	:		actual conflicts	:	51314578
Scalar memory refs/sec	:	0.64M	actual refs	:	45872184
Block memory refs/sec	:	107.08M	actual refs	:	7734953176
CPU memory refs/sec	:	107.72M	actual refs	:	7780825360
% having conflicts 6.56	:		actual conflicts	:	510748658
CPU memory writes/sec	:	35.33M	actual refs	:	2552186714
CPU memory reads/sec	:	72.39M	actual refs	:	5228638646

Group 3: CPU seconds : 72.18967 CP executing: 11406172771

(octal) type of instruction	inst./CPUsec	actual inst.	% of all inst
(000-017)jump/special	1.42M	102271667	8.73
(020-077)scalar functional unit	8.75M	631462036	53.91
(100-137)scalar memory	0.64M	45872184	3.92
(140-157,175)vector integer/log	0.97M	70335483	6.00
(160-174)vector floating point	2.77M	199796989	17.06
(176-177)vector load and store	1.68M	121552567	10.38

type of operation	ops/CPUsec	actual ops	avg. VL
Vector integer&logical	62.11M	4483931095	63.75
Vector floating point	176.48M	12739787062	63.76

	ops/wallsec		
Vector integer&logical :	46.68M	CPU/wallclock	
Vector floating point:	132.63M	time ratio:	0.75

Appendix 2
Percent Vectorization Calculation
Method 2

Percent vectorization was estimated from hpm groups 0,2, and 3 by two methods. The additional vector logical unit (AVL) was disabled by means of the */etc/cpu* command so that the hpm would report the correct number of vector floating point operations. The methods differ by the calculation of the number of vector memory references. Method 2a calculates vector memory references directly from the number of vector load and store operations, while Method 2b equates block memory references to vector memory references. Both methods yield identical results.

1. Method 2a

The percent vectorization of operations $P_V(2a)$ is given by

$$P_V(2a) = \frac{V_t}{V_t + S_t} * 100$$

where

V_t = Total vector operations
S_t = Total scalar operations.

The total vector operations V_t are given by the equation

$$V_t = V_{fp} + V_{il} + V_{mr}$$

where

V_{fp}	=	Vector floating point operations	(Group 3, line 11)
V_{il}	=	Vector integer and logical operations	(Group 3, line 10)
V_{mr}	=	Vector memory references.	

Vector memory references are the product of the number of vector load and store instructions and the average vector length

$$V_{mr} = V_{ls} * A_{vt}$$

where

V_{ls}	=	Vector load and store instructions	(Group 3, line 8)
A_{vt}	=	Average vector length (total).	

The average vector length is calculated by:

$$A_{vt} = \frac{V_{il} * A_{vil} + V_{fp} * A_{vfp}}{V_{il} + V_{fp}}$$

where

A_{vil}	=	Average vector length for integer and logical instructions	(Group 3, line 10)
A_{vfp}	=	Average vector length for floating point instructions	(Group 3, line 11)

The total scalar operations S_t are calculated from the equation

$$S_t = S_{mr} + S_{fu} + J_s$$

where

S_{mr}	=	Scalar memory references	(Group 3, line 5)
S_{fu}	=	Scalar functional unit	(Group 3, line 4)
J_s	=	Jump/special instructions	(Group 3, line 3)

Scalar integer and logical instructions are performed in the scalar functional unit.

2. Method 2b

The second method uses block memory references to determine the number of vector memory references. The percent vectorization $P_v(2b)$ is given by:

$$P_v(2b) = \frac{V_t}{V_t + S_t} * 100$$

where

V_t	=	Total vector operations
S_t	=	Total scalar operations.

The total scalar operations are calculated as in the previous method. The total vector operations V_t are given by the equation:

$$V_t = V_{fp} + V_{il} + V_{bmr}$$

where

V_{fp}	=	Vector floating point operations	(Group 3, line 11)
V_{il}	=	Vector integer and logical operations	(Group 3, line 10)
V_{bmr}	=	Block memory references	(Group 2, line 7)