PRACTICAL CONCERNS IN MULTITASKING ON THE CRAY X-MP

John L. Larson
Cray Research, Inc.
Chippewa Falls, WI

Summary: Multitasking and vectorization are both optimizations which exploit program and machine parallelism. Many of the goals for obtaining increased performance through multitasking are identical to those for vectorization, only the terminology and underlying structures are changed. High performance on the CRAY X-MP is achieved by addressing familiar concerns in a new setting. Practical objectives in programming as well as limitations in performance are described for a multitasking environment.

1. INTRODUCTION

The exploitation of parallelism for increased performance is commonplace. Architectural features such as multibanked memory, instruction execution overlap, multiple functional units, and asynchronous I/O are but a few examples of techniques which use parallelism to decrease program execution time. The CRAY-1, with its segmented functional units, exploits pipeline parallelism to increase performance through vectorization, Johnson(1978). The factors which influence vector performance have been studied in detail. Relationships between performance and quantities such as percentage of time vectorized, vector length, and balanced resource utilization seem well understood. See, for example, Hockney and Jesshope(1981).

With the introduction of the vector multiprocessor CRAY X-MP (2 processors in 1982, 4 processors in 1984) simultaneous exploitation of vectorization and multitasking became possible, Hwang and Briggs(1984). At first

glance, many of the concerns for increased performance through multitasking seem new. However, closer inspection reveals that the goals in multitasking are identical to those in vectorization, only the terminology and underlying structures are different. This paper investigates the issues in obtaining optimal performance in a multitasking environment and relates them to familiar concerns in vectorization optimization.

2. CRAY-X-MP/48 Hardware Overview

Figures 1 and 2 illustrate, respectively, the overall system organization and typical data flow of the CRAY X-MP/48. The mainframe consists of 4 identical processors which share 8 million 64-bit words of ECL bipolar memory. The clock cycle time is 9.5 nanoseconds, and the memory bank cycle time is 38 nanoseconds. Processors may be configured to operate independently on separate jobs, or assigned in any combination to operate jointly on a single job. For a multitasked job, the processors may use a shared set (cluster) of additional registers for synchronization and communication.

The computational capabilities of the mainframe are complimented by the I/O capabilities of the DD-49 disk, the I/O Subsystem (IOS) and the Solid-state Storage Device (SSD). The DD-49 disk has a capacity of 1200 megabytes, a 10 megabytes/second transfer rate, and an access time of 20 milliseconds. The IOS contains up to 8 million words of buffer memory, and is connected to the mainframe and SSD by several 40 megabytes/second channels. The SSD provides up to 128 megawords of secondary storage, and is connected to the mainframe by 2 very high speed channels with an aggregate transfer rate of 2000 megabytes/seconds. The rates given here represent actual, rather than peak performance.

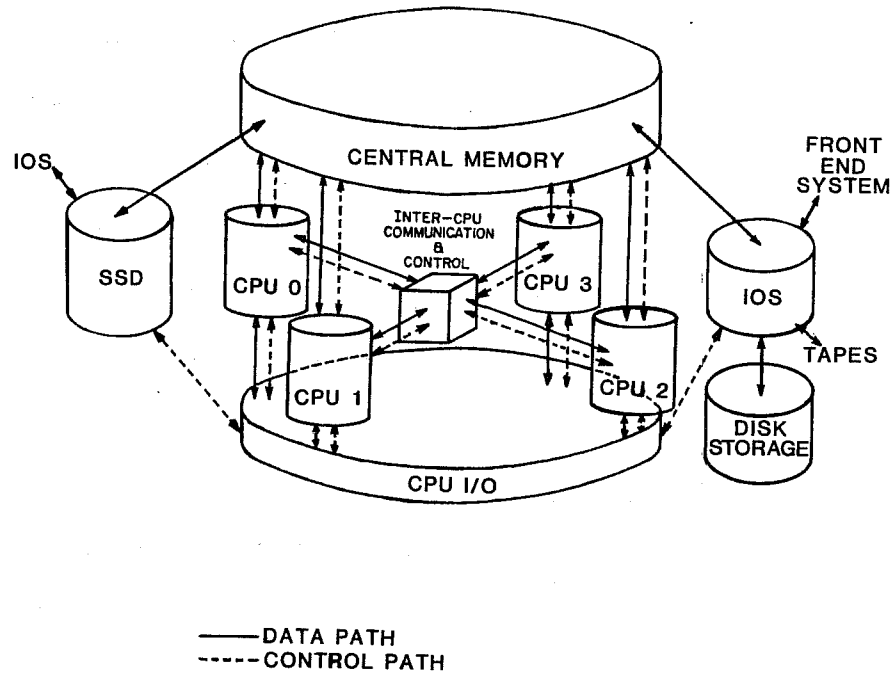


Figure 1. CRAY X-MP-4 System Organization

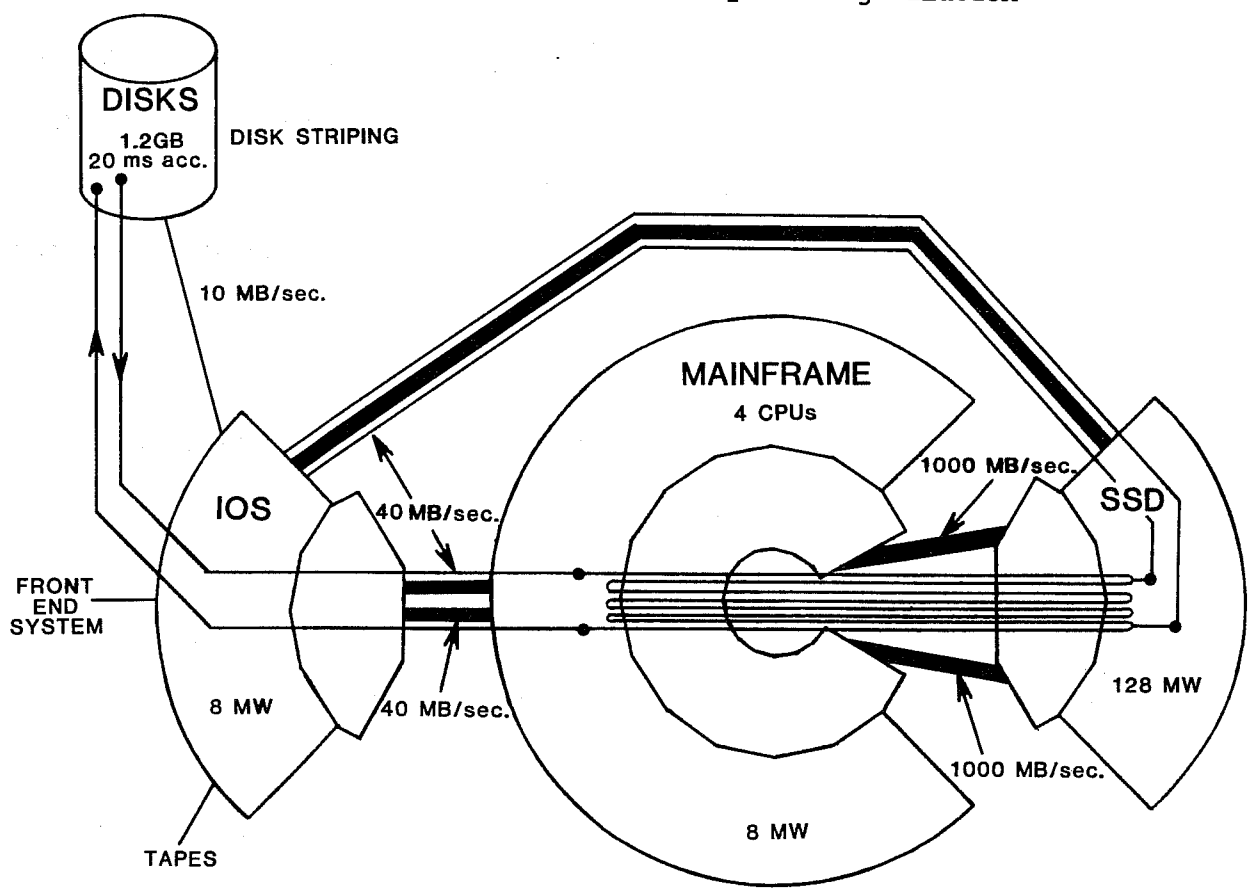


Figure 2. CRAY X-MP-4 Data Flow

Data flow for a typical job moves input data files from disk to central memory or SSD. Disk transfers may use the parallel disk striping capabilities of the IOS to enhance the disk transfer rate. Temporary files may be created and repetitively manipulated on the SSD during program execution. Output data files are migrated from central memory or SSD to disk at the end of the job.

3. CRAY X-MP/48 SOFTWARE OVERVIEW

Software support for multitasking programming on the CRAY X-MP/48 is available at the FORTRAN level, where basic multitasking operations are implemented as user callable subroutines. Table 1 contains the key multitasking subroutines in the multitasking library. The capability of starting an asynchronous task to be executed on another processor is provided by the TSKSTART routine. Synchronization, communication and mutual exclusion functions are provided by the other routines using various programming styles. See Larson(1984), and CRAY(1985) for more details.

Category	Subroutine	Function
TASK CONTROL	CALL TSKSTART(TASKID, SUBNAME,ARGS)	Creates a task with identification TASKID and entry point SUBNAME.
	CALL TSKWAIT(TASKID)	Suspends the calling task until the task with identification TASKID has completed.
EVENT CONTROL	CALL EVPOST(EVENT)	Changes EVENT to 'posted'.
	CALL EVWAIT(EVENT)	Suspends the calling task until the status of EVENT is 'posted'.
	CALL EVCLEAR(EVENT)	Changes EVENT to 'cleared'.
LOCK CONTROL	CALL LOCKON(LOCK)	Suspends calling task until LOCK is 'unlocked', then changes LOCK to 'locked'.
	CALL LOCKOFF(LOCK)	Changes LOCK to 'unlocked'.

Table 1. Key multitasking subroutines

The FORTRAN multitasking environment is also supported by enhancements in

other system software. Code reentrancy is provided by CFT, the FORTRAN compiler, through the generation of stack-based object code and a compatible calling sequence. The operating system, COS, allows a job to create new tasks belonging to the job, and handles tasks as the fundamental unit of schedulable work. The multitasking library also performs task scheduling at the user level.

4. MULTITASKING PERFORMANCE ISSUES

In this section we investigate several concerns pertaining to multitasking performance. Upper limits of achievable speedup and lower limits of applicability are studied. Programming techniques are given to maximize load balance and minimize memory contention. Finally, the distinction between time and work is clarified.

4.1 Theoretical Speedup and Amdahl's Law

Multitasking is an optimization which changes the apparent execution time of code segments to which it is applied. The result of having two different execution speeds is that the overall execution time of a multitasked program behaves according to Amdahl's Law based on the percentage of time which is multitasked, Amdahl(1967).

Let T1 be the execution time of a non-multitasked program. If multitasking is applied to a fraction, f, of the original execution time, then the theoretical execution time (assuming no overhead or delays) is the time to do the sequential portion, Ts, plus the time to perform the multitasked part, Tm. These quantities are a function of the original execution time, the number of processors (p), and f.

T1 = original execution time

f = fraction of T1 multitasked

p = number of processors

Tp = wall clock time of multitasked execution

Ts = (1-f)*T1 = wall clock time of sequential part

Tm = (f/p)*T1 = wall clock time of multitasked part

The theoretical speedup attainable with p processors, S(p,f), is a ratio of the original execution time to the total execution time of the multitasked program.

$$S(p,f) = \frac{T1}{Tp} = \frac{T1}{Ts + Tm} = \frac{T1}{T1 * ((1-f) + (f/p))}$$

For a given, fixed number of processors, a plot of speedup versus f produces the familiar Amdahl's Law curve in figure 3.

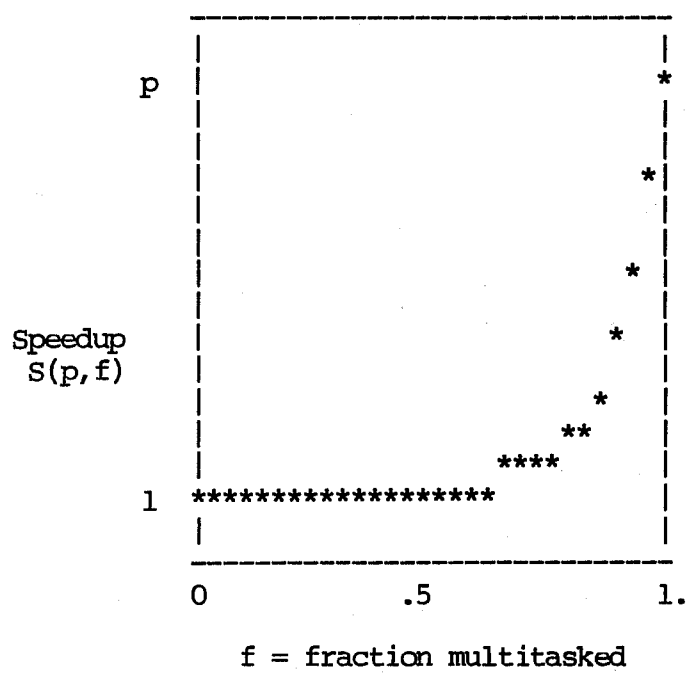


Figure 3. Amdahl's Law

It is interesting to look at the theoretical speedup for different values of p and f. See Table 2. Several important observations can be made from this table. The small entries in the lower part of the table show that significant speedups are not possible unless significant portions of the program are multitasked. For example, if 50 percent of the time in a code is multitasked on 4 processors, it should be no surprise that the actual speedup attained is, say, "only 1.5". The best possible speedup which could have been achieved is 1.6. The tendency to think that with p processors, speedups should always be p, is a myth.

fraction of time multitasked	number of processors							
f	p=1	p=2	p=4	p=8	p=16	p=32	p=64	p=infinity
1.00	1.00	2.00	4.00	8.00	16.00	32.00	64.00	infinity
.99	1.00	1.98	3.88	7.48	13.91	24.43	39.26	100.00
.98	1.00	1.96	3.77	7.02	12.31	19.75	28.32	50.00
.97	1.00	1.94	3.67	6.61	11.03	16.58	22.14	33.33
.96	1.00	1.92	3.57	6.25	10.00	14.29	18.18	25.00
.95	1.00	1.90	3.48	5.93	9.14	12.55	15.42	20.00
.94	1.00	1.89	3.39	5.63	8.42	11.19	13.39	16.67
.93	1.00	1.87	3.31	5.37	7.80	10.09	11.83	14.28
.92	1.00	1.85	3.23	5.13	7.27	9.19	10.60	12.50
.91	1.00	1.83	3.15	4.91	6.81	8.44	9.59	11.11
.90	1.00	1.82	3.08	4.71	6.40	7.80	8.77	10.00
.75	1.00	1.60	2.28	2.91	3.37	3.66	3.82	4.00
.50	1.00	1.33	1.60	1.78	1.88	1.94	1.97	2.00
.25	1.00	1.14	1.23	1.28	1.31	1.32	1.33	1.33
.10	1.00	1.05	1.08	1.09	1.10	1.11	1.11	1.11
.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 2. Theoretical speedup

Another message contained in the table comes from the upper rows. For a fixed percentage of multitasked execution time, the speedup does not increase as fast as the number of processors. Indeed, the speedup converges to a finite limit for p=infinity. The cause of this phenomenon is the percentage of time not multitasked. For a large number of processors, the

execution wall clock time is dominated by the non-multitasked code. Speedups approaching the number of processors are possible, but for larger number of processors, much more of the code must be multitasked. This encourages future application development to strive, using a top-down approach, for high level parallelism.

The use of Amdahl's law in explaining general speedup limitations for vectorization is commonplace. In this environment f represents the fraction of the original sequential execution time which is vectorizable, and p denotes the ratio of vector to scalar speed. An alternative formulation would relate (1-f) to the ratio of vector startup time to total vector execution time, and p would represent the number of pipeline stages.

Multitasking speedup performance is often compared to the "obvious" speedup maximum (the number of processors), although the correct comparison should be against the theoretical speedup given by Amdahl's Law. Ironically, actual vectorization speedup performance is seldom compared in the same way against any theoretical maximum.

4.2 Granularity and Overhead

At one end, Amdahl's Law limits the maximum performance possible for a multitasked program. At another extreme, the overhead incurred in multitasking limits the smallest granularity which can be profitably exploited. By granularity we mean the length of time to execute a multitaskable segment of the original program on a single CPU. Granularity is measured in time units, rather than floating point operations, since the amount of work that can be done in a given time depends on whether the computation is in scalar or vector mode.

On the CRAY X-MP, a granularity of one millisecond is considered large. For this granularity size, the overhead of the FORTRAN multitasking calls is often negligible. Very small granularity (on the order of microseconds) can also be profitably exploited through assembly language coding for synchronization which directly accesses the hardware supporting multitasking. See Chen, Dongarra, and Hsiung(1984).

The effect of overhead on performance can be modeled in a simple way. Let T1 = X be the time to execute a multitaskable program segment on one processor. Then for a given overhead, OH, the time to execute the code on p CPUs will be Tp = OH + X/p. The ratio T1/Tp gives the speedup, Sp, of the multitasked code over the original program. Figure 4 shows the relationship of speedup to granularity for a fixed overhead.

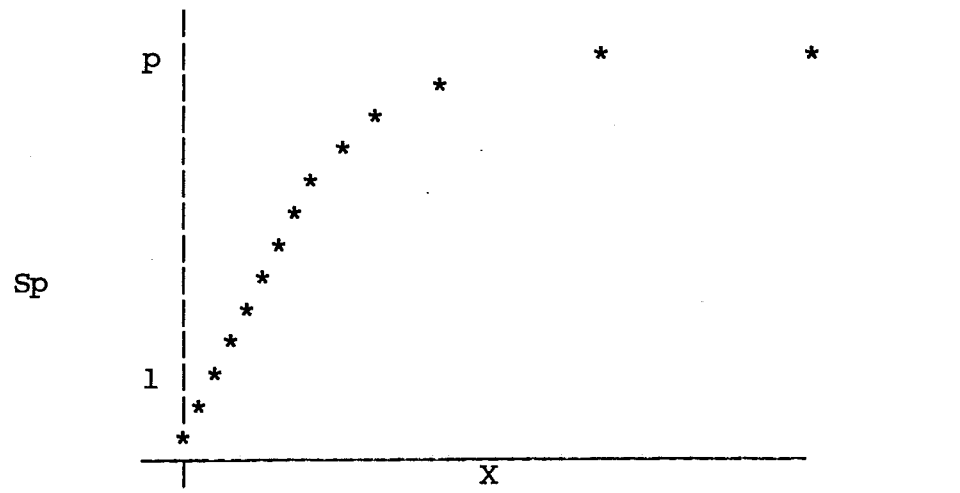


Figure 4. Speedup vs. granularity for fixed overhead

By simple manipulation of the formula for speedup, one obtains a relationship for the minimum granularity required to obtain a given speedup in the presence of overhead.

$$X = (Sp * p * OH) / (p-Sp)$$

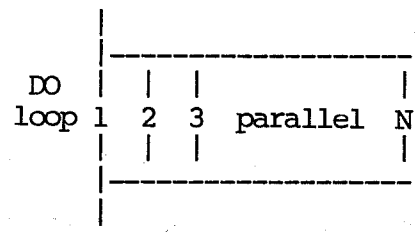
For a given OH on p processors, a speedup Sp can be obtained only if the granularity is at least X.

The curve in figure 4 is identical to one produced in vectorization analysis. For this environment, granularity is measured in terms of vector length, (VL), and the overhead is denoted as vector startup time. As Kuck(1978) describes, a scalar computation may take time T1 = k(VL) where k is the clock time for the operation. For a pipelined computation using a functional unit with k stages, the vector time is Tk = (k-1) + VL, where the overhead of filling the pipe is (k-1). The speedup curve of Sk = T1/Tk vs. VL shows the minimum vector length required for profitable vectorization speedups.

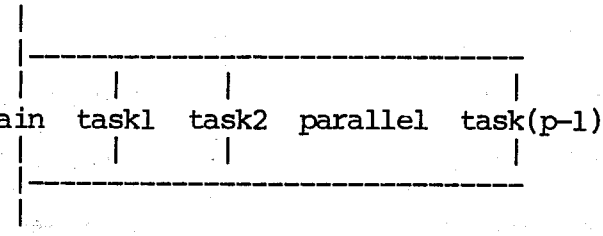
4.3 Load Balancing

Multitasking is most often applied to parallel work found in the independent iterations of DO loops. If the loop has N iterations, we call N the extent of the parallelism of the loop. Load balancing is the technique of mapping N onto p processors or tasks so that each task has the same amount of work to do. See figure 5.

There are two major cases to consider: static and dynamic partitioning. Static partitioning is used when the time for each iteration of the loop is approximately the same. This technique assigns a fixed subset of iterations to each task. The iterations which a task is to perform can be computed from the task, or processor number. One strategy assigns a subset of contiguous iterations to each processor. The jth processor computes iterations (j*N)/p+1 to ((j+1)*N)/p. Another static strategy assigns the iterations in an interleaved fashion. The jth processor computes every pth iteration from j+1 to N. See figure 6.



N = extent of parallelism



multitasked code on p processors

Figure 5. Mapping parallelism to processors

Processor	Assigned iterations		
p0	I = 1, N/4		I = 1, N, P
p1	I = N/4+1, 2*N/4	or	I = 2, N, P
p2	I = 2*N/4+1, 3*N/4		I = 3, N, P
p3	I = 3*N/4+1, 4*N/4		I = 4, N, P

Figure 6. Static partitioning with p = 4

If the time for an iteration of the loop varies significantly then a dynamic partitioning of work will tend to balance the load on each processor. This technique maintains a shared counter which indicates the next unprocessed iteration. Each processor accesses and updates the counter to commit itself to one or more iterations. Those processors which commit to short iterations will return more often to find more work, while processors committed to long iterations will look for new work less frequently. This technique incurs an overhead to protect the counter during the update process. If the average granularity of the iterations is large compared to this overhead, then the iterations may be distributed one at a time. If the average granularity of the iterations is small compared

to the overhead of protecting the iteration counter, then the iterations can be distributed to processors in chunks. Each chunk contains K iterations. See figure 7, where I, K, and N are shared, and L is private to each task.

```
      COMMON /TASKS/ I, K, N
      I = 1

Each processor executes

 10   CONTINUE
      CALL LOCKON ( LOCKI )
      L = I
      I = L + K
      CALL LOCKOFF ( LOCKI )
      IF ( L.GT.N ) GO TO 20
      compute interations L through min(L+K-1,N)
      GO TO 10
 20   CONTINUE
```

Figure 7. Dynamic partitioning for large (K=1) or small (K>1) granularity iterations

Heuristics can be devised for choosing the chunking factor, K. On the one hand, too small a value of K will not allow the lock overhead to be amortized over enough iterations. Also, too large a value of K will not produce enough chunks to allow load balancing to take place. See figure 8.

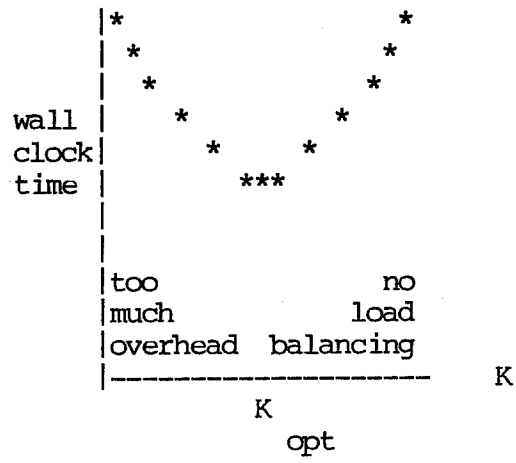


Figure 8. Choosing the optimal chunking factor

To reduce the relative overhead, K should be at least Kmin. To facilitate

load balancing, K should be less than Kmax. The value of K should be chosen in the range (Kmin, Kmax), where

$$\text{Kmin} = \text{smallest K such that } \frac{(\text{lock overhead})}{K*(\text{ave.iter.time})} < .01$$

$$\text{Kmax} = \text{largest K such that } \frac{N}{K} > 10*p$$

The balancing of computational work on several processors in multitasking has the same goal as overlapping asynchronous I/O with computation in a uniprocessing environment, namely, to maximize the simultaneous utilization of available resources.

4.4 Minimizing Memory Contention

One of the factors influencing performance in a shared memory architecture is memory contention. Memory references made with the three computational ports of each processor may result in resource conflicts. These conflicts are resolved by having some references wait until the required resource becomes available. This waiting can cause a computation on a single CPU to take longer in a system with all processors busy, than in an otherwise quiet environment.

The user has control over the intensity of memory references as reflected in the programming style. The intensity of vector memory references can be measured in units of memory references per floating point operation (memrefs/flop). Also important is the layout of the memory references across the interleaved memory banks.

Programming styles which reduce the memory contention produce an increased performance both in vector uniprocessing and multitasking environments.

The following is a list of program optimizations which reduce the memory contention for the current compiler. A complete description can be found in CRAY(1985).

1. Vertical Inner Loop Unrolling
2. Horizontal Inner Loop Unrolling
3. Vertical Outer Loop Unrolling
4. Horizontal Outer Loop Unrolling

   For example, (Dongarra(1984)),

```
        DO 40 J = 1, 4*M+1                3 MEMREFS / 2 FLOPS
          DO 40 I = 1, N
40          Y(I) = Y(I) + X(J) * M(I,J)
        DO 41 J = 1, 4*M+1, 4             6 MEMREFS / 8 FLOPS
          DO 41 I = 1, N
41          Y(I) = (((Y(I) + X(J  ) * M(I,J  ))
     $                   + X(J+1) * M(I,J+1))
     $                   + X(J+2) * M(I,J+2))
     $                   + X(J+3) * M(I,J+3)
```

5. Dimensioning of Arrays

   For example,

```
        DIMENSION X(64,100), XX(65,100)
        DO 50 I = 1, 64           X(I,*) IN 1 BANK
          DO 50 J = 1, 100
50          X(I,J) = X(I,J) + 1.0
        DO 51 I = 1, 64           XX(I,*) IN 64 BANKS
          DO 51 J = 1, 100
51          XX(I,J) = XX(I,J) + 1.0
```

6. Loop Interchange
7. Padding Between Common Block Arrays

These techniques are commonly considered vectorization optimizations since they enhance vector performance in a uniprocessing environment. However, they are also multitasking optimizations which minimize inter-CPU memory contention by minimizing the intra-CPU conflicts.

4.5 Measuring Time and Work

For a single CPU system used in a dedicated mode, measuring work is often done with CPU charges in time units. For computational jobs, the CPU time and the wall clock time are frequently quite close. No distinction between time and work is necessary. The ability of several CPUs to perform work at the same time creates a distinction between work and the time to complete the work. In a multitasked job, several CPUs may perform more work in less real time than an equivalent program executed on a single processor. The presence of multitasking overhead raises the question of what work is to be credited and measured.

Kuck(1978) defines some basic quantities which clarify the situation. See figure 9. From an external viewpoint, we observe the work (CPU charges), O1, and the wall clock time, T1, for a single processor execution of the program. Often these quantities are defined to be equal. We also record the wall clock time, Tp, for the execution on a p processor system. The ratio of T1 to Tp gives the speedup, Sp. A comparison of Sp to p quantifies the efficiency, Ep. This is an equitable comparison only when the theoretical speedup is p (the program is 100% multitaskable).

From an internal viewpoint, we observe the work (CPU charges), Op, and the wall clock time, Tp, of the multitasked job. Naturally, Tp has the same value as was recorded externally. However, additional work may be done in the parallel execution. Such work may include charges for multitasking

calls, synchronization delays, and memory contention. The redundancy, Rp, measures the extra work performed by the multitasked program. The internal speedup, "Sp", gives credit for all the work performed. Finally, the utilization, Up, shows how busy the system really is.

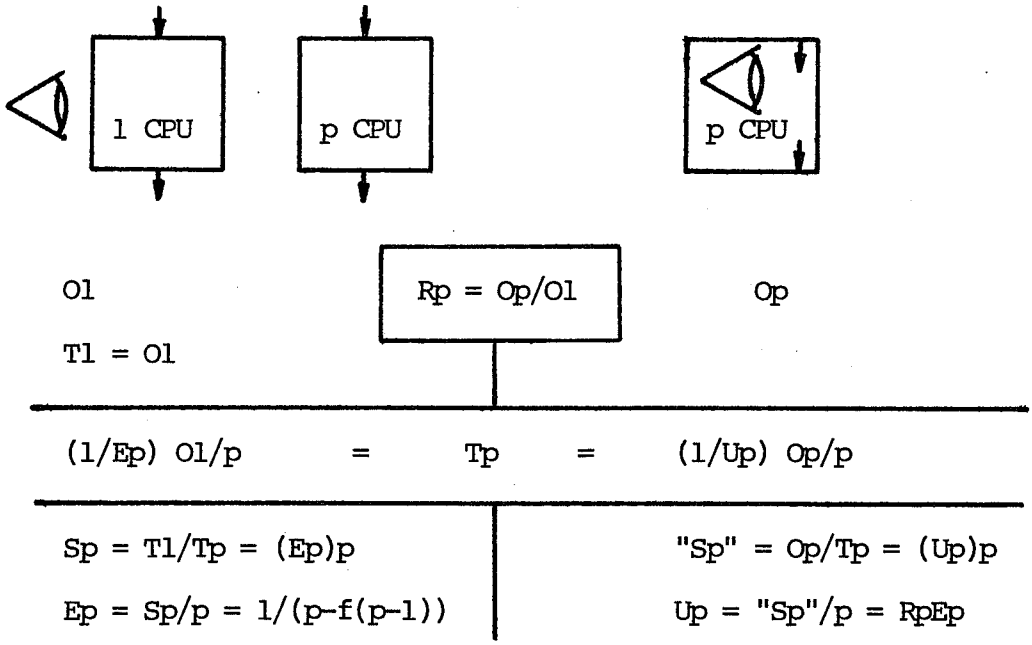


Figure 9. Measuring time and work

Some algebraic manipulation shows that the two viewpoints define similar quantities within a factor of Rp. The redundancy factor Rp is the link which relates and separates the two perspectives.

The situation is illustrated by an example code. A lattice gauge program, which is 100% multitaskable, requires 3.28 hours on a single X-MP-4 processor. When executed on four CPUs, the CPU charges are 3.45 hours, and the run completes in 0.87 hours. We have O1 = T1 = 3.28, Op = 3.45, and Tp = 0.87. From these quantities all others may be computed. The realized speedup is Sp = 3.77 with an efficiency Ep = 0.94. Internally, the system speedup is "Sp" = 3.96 and utilization is 0.99, where more work (Rp = 1.05) was performed than was in the original program.

One way to interpret these results is in the spirit of a backward error analysis for assessing roundoff error. Here we relate the performance of the multitasked job with overhead to a hypothetical multitasked job with no overhead, but with a smaller fraction of time multitasked. By doing this, all of the overhead can be accounted for in a single, simple way. From the formula, $Ep = 1/(p-f(p-1))$, we can see that our 100% multitaskable job with overhead gave the same performance as an equivalent program with no overhead, but which was 98% multitaskable. Amdahl's Law shows that for $p=4$ and $f=0.98$ the theoretical speedup with no overhead is $Sp=3.77$, which was the speedup recorded for our actual program.

The quantities presented here can be defined in a vectorization environment. For example, the value of p may be taken as the vector speed to scalar speed ratio. The vectorized work, Op, may account for vector set up or for redundant operations such as those which occur in masked operations. Efficiency, Ep, may take into account the fraction of time which was vectorized in computing the theoretical speedup possible.

5. APPLICATION PERFORMANCE

Multitasking is utilized as a performance optimization for a variety of application codes on the CRAY X-MP. The results are summarized in table 3. Comparisons are made against runs executed on the same system, but employing only one processor. The percent multitaskable column shows the fraction of single processor execution time which was multitasked. This percentage limited the theoretical speedup, based on Amdahl's Law, which could be obtained. These multitasking percentage values are greater than typical vectorization percentages, illustrating the common presense of extensive high level parallelism in many application areas. The nearness of the actual speedups to the theoretical speedups shows the high efficiency delivered by the CRAY X-MP.

	percent multitaskable	(4 CPUs) theoretical speedup	actual speedup
1. Particle-in Cell	97 %	3.67	3.48
2. Weather Forecast	98 %	3.77	3.55
3. Seismic Migration	98.7%	3.85	3.45
4. Monte Carlo	99 %	3.88	3.75
5. Lattice Gauge	100 %	4.00	3.77
6. Seismic Forward Modelling	98.2%	3.80	3.50
7. Aerodynamics	98.8%	3.86	3.22
8. Chess	variable	n/a	3.15*

* CPU time divided by wall clock time

Table 3. Application performance

6. CONCLUSION

In a sense, multitasking is not new. It is only the exploitation of parallelism along a different dimension than vectorization. Multitasking urges us to open our eyes wider, beyond the innermost loop, to see parallelism at a higher level, where often a natural independence exists along geometric or other problem attributes. Even at this higher level, however, the concerns in the exploitation of multitasking parallelism are basically the same as for vectorization, only the details are different. Additionally, multitasking offers an optimization alternative, often being applicable when vectorization is not possible. When applied on top of vectorization, multitasking gives the performance benefits of both worlds.

7. REFERENCES

Amdahl, G., 1967: The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. AFIPS Conference Proceedings, SJCC, 3, pp. 483-485.

Chen, Steven S., Jack J. Dongarra, and Christopher C. Hsiung, 1984: Multitasking Linear Algebra Algorithms on the CRAY X-MP-2: Experiences with Small Granularity. Journal of Parallel and Distributed Computing, 1, pp. 22-31.

Cray Research, Inc., 1985: Multitasking User Guide. CRI internal technical note, SN-0222, second printing.

Dongarra, Jack J., and Stanley C. Eisenstat, 1984: Squeezing the Most Out of an Algorithm in CRAY FORTRAN. ACM TOMS, 10, pp. 221-230.

Hockney, R. W., and C. R. Jesshope, 1981: Parallel Computers. Adam Hilger Ltd., Bristol, England, pp. 47-95.

Hwang, Kai, and Faye A. Briggs, 1984: Computer Architecture and Parallel Processing. McGraw-Hill, New York, pp. 714-728.

Johnson, Paul M., 1978: An Introduction to Vector Processing. Computer Design, 17, pp. 89-97.

Kuck, David J., 1978: The Structure of Computers and Computations. John Wiley and Sons, New York, pp. 100 and 255.

Larson, John L., 1984: Multitasking on the CRAY X-MP-2 Multiprocessor. IEEE Computer, 17, pp. 62-69.