JOURNAL OF PARALLEL AND DISTRIBUTED COMPUTING 1, 22–31 (1984)

Multiprocessing Linear Algebra Algorithms on the CRAY X-MP-2: Experiences with Small Granularity

STEVEN S. CHEN

CRAY Research Inc., Chippewa Falls, Wisconsin 54729

JACK J. DONGARRA*

Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, Illinois 60439

AND

CHRISTOPHER C. HSIUNG

CRAY Research Inc., Chippewa Falls, Wisconsin 54729

This paper gives a brief overview of the CRAY X-MP-2 general-purpose multiprocessor system and discusses how it can be used effectively to solve problems that have small granularity. An implementation is described for linear algebra algorithms that solve systems of linear equations when the matrix is general and when the matrix is symmetric and positive definite.

OVERVIEW OF THE SYSTEM

"Multiprocessor" is a term that has been used for years. Our definition follows those of [8], [9], and [10].

The CRAY X-MP is a follow-up to the CRAY-1S system offered by CRAY Research, Inc. The CRAY X-MP family is a general-purpose multiprocessor system. It inherits the basic vector functions of CRAY-1S, with major architectural improvements for each individual processor. The interprocessor communication mechanism and the provision of Solid-State Disk device(SSD) are new designs that create tremendous potential in the realm of high-speed computing.

The CRAY X-MP-2 system is the first product of the CRAY X-MP family.

*Work supported in part by the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research of the U.S. Department of Energy under Contract W-31-109-Eng-38.

0743-7315/84 $3.00

It is a dual processor model that is housed in a physical chassis identical to that of the CRAY-1S. Each processor occupies half of the space of the original CRAY-1S. This is achieved through larger IC integration, denser packaging, and much improved cooling capacity.

The system is designed specifically to handle computation-intensive and I/O-intensive jobs in an efficient way. It can be used to perform simultaneous scalar and vector processing of either independent job streams or independent tasks within one job. Hardware in the X-MP enables multiple processors to be applied to a single Fortran program in a timely and coordinated manner. (See Fig. 1.)

All processors share a central bipolar memory (of up to 4 million words), organized in 32 interleaved memory banks. Each processor has four memory ports: two for vector fetches, one for vector store, and one for independent I/O operations. In other words, the total memory bandwidth of the two processors is up to eight times that of the CRAY-1S system. The added memory bandwidth provides a balanced architecture for memory-to-memory vector operations as typified by scientific Fortran codes.

Other features of the machine include hardware automatic "flexible chaining" [1]. This feature allows each individual processor to have simultaneous memory fetches, arithmetic, and memory store operations in a series of related vector operations. The elimination of "chain slot time" guarantees "supervector" speed in all vector operations. It contrasts with the "fixed chaining" and unidirectional vector fetch/store of the CRAY-1S [7]. The balance between CPU speed and memory bandwidth makes the CRAY X-MP more friendly to Fortran codes. The need to resort to assembly level hand

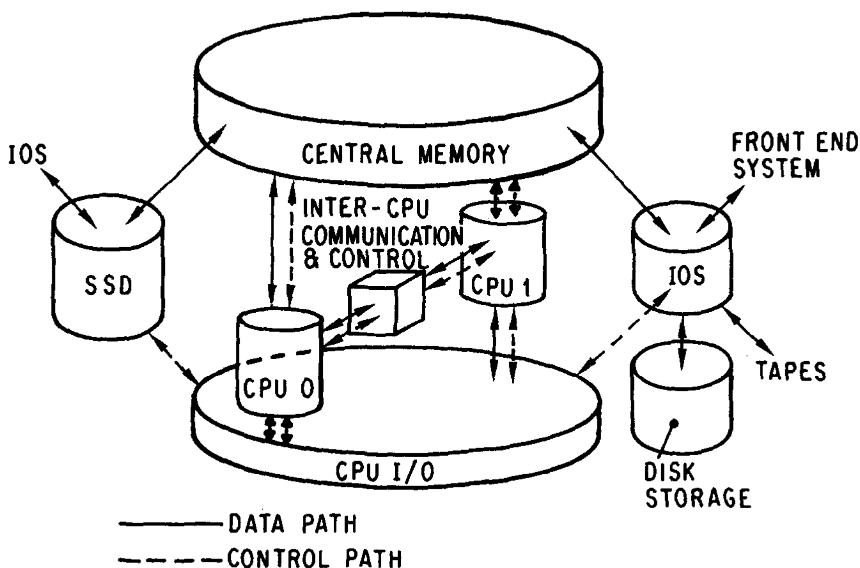


FIG. 1. Cray X-MP-2 system organization.

coding is drastically reduced. Other improved features of the CPU include multiple data paths and increased instruction buffer size. The machine has a faster central clock with improved cycle time, 9.5 nsec as compared with 12.5 nsec on the CRAY-1. The result is a much improved scalar and vector speed over the CRAY-1S.

In addition, a new, optional, CPU-driven Solid-State Storage Device (SSD) offers an extended central memory, an important element to buffer between fast central memory and slow disk devices. The SSD can be used as a fast-access device for large prestage or intermediate files generated and manipulated repetitively by user programs, or it can be used by the system for job swapping space and temporary storage of frequently used system programs.

While multiprocessing is not a new concept, the CRAY X-MP multiprocessor design is unique in many ways. It differs from most other conventional multiprocessors (see, for example, [10], [13]) in its multilevel parallelism (vectorization in the inner most loop and multiprocessing in outer loops) and its tightly coupled interprocessor communication control (sharing of registers, for example).

Additional hardware in the X-MP enables efficient and coordianted application of multiple processors to a single user program. All processors assigned to a task share a unique set of synchronization and communication registers. There are three kinds of shared registers: a set of binary semaphores, a set of index registers, and a set of data registers. These registers, in cooperating with the shared central memory, allow processors to signal each other, wait for each other, and transfer data between each other. Processors can also interrupt each other through the interprocessor interrupt. These basic hardware functions provide a basic mechanism for efficient communication and synchronization between processors.

Other than hardware instructions, software support for multitasking is at the library level, where the user makes calls to ask the system for multitasking functions. Three categories of routines provide different mechanisms for parallel processing [5]. First, a task can be created to be scheduled for execution through the TSKSTART call. The calling task may wait for the termination of a created task with the TSKWAIT routine. Task is a schedulable unit that the user expects to be executed in a serial manner. It is a software entity that the programmer deals with, as the physical processor is concealed from him.

Second, tasks may need to communicate or synchronize with one another as they execute concurrently. Producing tasks may signal consuming tasks through an EVPOST routine. A consuming task may wait for the signal through an EVWAIT routine. As the signal is consumed, it may be reset through an EVCLEAR call.

In the third category, the LOCKON and LOCKOFF routines are used to protect the integrity of code segment or shared resources (e.g., data) among tasks.

Based on these three categories, other synchronization and communication mechanisms can be developed. Since the multitasking library employs a queuing mechanism (among others) to handle general situations, it is very flexible. Of course, the user always has the option of hand coding his own synchronization routines through the use of hardware instructions.

GRANULARITY OF TASKS

A number of factors influence the performance of an algorithm in multiprocessing. These include the degree of parallelism, process synchronization overhead, load balancing, interprocessor memory contention, and modifications needed to separate the parallel parts of an algorithm.

The size of the work performed in parallel, the granularity of the task, is the first critical factor in matching a parallel algorithm to an architecture which should be addressed. By "granularity," we mean the amount of time the cooperating tasks execute concurrently on related codes in between synchronization points. The need to synchronize and to communicate before and after parallel work will greatly impact the overall execution time of the program. Since the processors have to wait for one another instead of doing useful computation, it is obviously better to minimize that overhead. In the situation where segments of parallel code are executing in vector mode, typically at ten to twenty times the speed of scalar mode, granularity becomes an even more important issue, since communication mechanisms are implemented in scalar mode.

Granularity is also closely related to the degree of parallelism, which is defined to be the percentage of time spent in the parallel portion of the code. Typically, a small granularity job means that parallelism occurs in an inner loop level (although not necessarily the innermost loop). In this case, even the loop setup time in outer loops will become significant without even mentioning frequent task synchronization needs.

For the CRAY X-MP family, granularity on the order of milliseconds is considered large. Many reports [1, 4, 5] have shown significant speedups for multitasking of large granularity problems on the CRAY X-MP-2. For example, speedups of 1.8 to 1.9 were seen when two processors were used instead of one to run a particle-in-cell code, a weather forecasting model, and a three-dimensional seismic migration code. For these problems, the multitasking library is used to implement parallel tasks. The reported successes indicate that, for large granularity codes that have high degrees of parallelism, the payoff of doing multitasking on the CRAY X-MP-2 is very significant. The use of the CRAY multitasking library to handle intertask communications proves to be very powerful. It will be interesting to see, however, how far we can push the granularity down and still get a descent speedup. As will be shown later, even when the library appears to be too coarse for small

granularity tasks, the hardware capability still allows efficient handling of them. For our purposes, we would like to use matrix vector operation to test the machine behavior for small granularity tasks.

THE ALGORITHMS

Employment of automatic compilation techniques to identify parallel work [11], [12] is an approach with general applications. In linear algebra code, it may have greater payoff if we can change the algorithm to exploit structures with bigger parallel blocks.

We have chosen matrix vector operation for two reasons. First, we can easily construct the standard algorithms in linear algebra out of these types of modules. Second, matrix vector operations provide simple modules for parallel execution [2, 3]. We will describe an implementation for standard LU factorization with partial pivoting for a general square matrix.

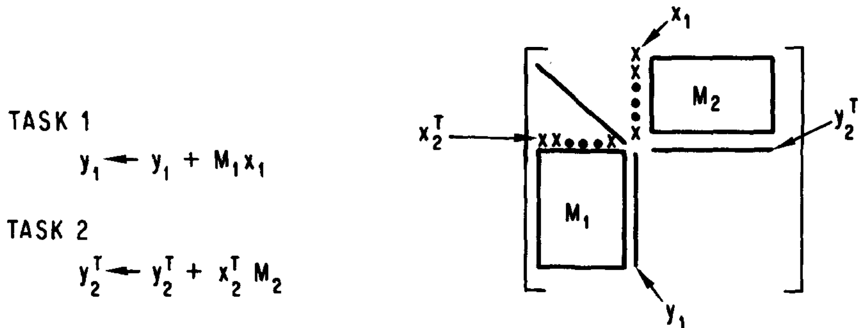
The algorithm can be described as having basically three distinct parts within a loop:

do $i = 1, n$
  perform the ith matrix vector product (forms part of L)
  search for a pivot and interchange
  perform the ith vector matrix product (forms part of U)
end

The algorithm organized storage so that the original matrix is overwritten with the factorization. The amount of work required to perform the factorization is approximately $2/3n^3$ floating point operations (here we count both additions and multiplications as operations). The factored matrix can then be used to solve systems of equations. In order to maintain stability in the algorithm, a partial pivoting scheme is used. This helps in avoiding problems with small divisors which can cause inaccurate computations. The pivot is chosen to be the element of largest absolute value in a column. The method used is a simple variant of Crout reduction [6], but the algorithm has been expressed in terms of modules that are easy both to understand and to implement.

The algorithm described above allows for a number of alternatives for parallel implementation. That is, the available processors can be assigned to each of the three steps, allowing for multiprocessing within each. This avenue has not been investigated since the pivoting does not take as much time as the other steps. Instead, each processor will be given the task of performing one of the matrix vector operations concurrently. The algorithm can be easily modified so each matrix vector operation is independent and can proceed in parallel. The pivoting is handled by one of the two processors in a sequential fashion.

Depicted graphically, the processing of the matrix by the algorithm at the ith stage would look like the following (the matrix factors overwrite the original matrix):



The algorithm is modified slightly to allow independent operations to be performed in parallel. The modified algorithm in no way increases the number of operations or complexity over the original algorithm. The modified algorithm just rearranges the computation within the loop to expose independent operations. The resulting modified algorithm is of the following form:

perform the 1st matrix vector product
  do $i = 1, n-1$.
  search for a pivot and interchange
  perform the ith vector matrix product
  perform the $(i+1)$st matrix vector product
end
perform the nth vector matrix product

It is now easy to see how to partition the work between two processors: each matrix vector operation within an iteration is independent of the other. The two cooperating tasks need to synchronize $2(n - 1)$ times during the course of the calculation (once before a task is started and once after). There is, however, a slight imbalance in the work of each task. At the ith step, the amount of work for each task is $O((n - i + 1) * i)$ and $O((n-i) * i)$ operations.

To perform the synchronization between tasks, we have used task control (TSKSTART) to start a second task before the LU decomposition routine is entered. This task waits until it is directed to start up. Event control (EVPOST, EVWAIT, and EVCLEAR) is used to start and synchronize the work within the algorithm.

Table I describes the performance for this implementation of LU decomposition. The column labeled "Degradation from code change" reflects the loss of performance when the sequential algorithm was restructured to separate independent tasks. The numbers are obtained by running the original algorithm and the modified algorithm using no multitasking features on a single processor and taking the percentage difference. Measurements were also made of the total time spent in the parallel portion of the algorithm. As expected, small-order matrices consume a greater percentage of time in nonparallel parts than do larger matrices. Even for matrices of order 50, however, over 75% of the time is used in the parallelizable portim.

TABLE I
PARALLEL VERSION OF LU DECOMPOSITION

n	Degradation from code change (%)	Percentage of parallel code (%)
50	6.5	75.1
100	4.9	82.0
300	2.0	87.5
600	0.7	97.0

We also measured the speedup of the algorithm when two processors were used. Table II shows the comparison between the modified (but without multitasking mechanism) one-processor version and the multitasked two-processor version. The column labeled "Mean granularity" is the average time spent in each of the matrix vector calls for that particular order problem. In other words, it is the average time in between synchronization calls.

The "Optimal" column is an attempt to filter out the time required to synchronize the processors and the time introduced by memory contention caused by multitasking on the two processors. These numbers were generated by running the program twice. The first run was a sequential process, using no multitasking constructs. In the second run the call to the shorter of the two parallel tasks was removed; thus, in some sense, this is the best situation (the calculation, of course, does not produce the correct results, but it provides a good measure of the overhead). The third column is the result of using the standard multitasking routines in the Multitasking (MT) library. As an alternative, there are assembly language (CAL) routines that use hardware instruction directly. These CAL routines perform the minimum synchronization function required by the code and are faster than the general-purpose multitasking library routines.[1]

TABLE II
SPEEDUP OF ALGORITHM ON 2 PROCESSORS VS 1 PROCESSOR

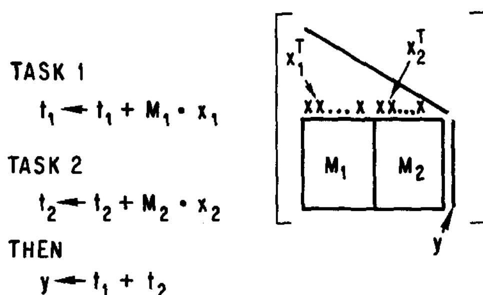
n	Optimal	MT library calls	MT CAL calls	Mean granularity (μsec)
50	1.44	0.86	1.39	40
100	1.57	1.05	1.54	66
300	1.77	1.60	1.79	250
600	1.91	1.80	1.86	800

[1]The MT library routines keep track of additional information on the activities of other tasks. They are better suited for larger and more tasks and are quite flexible for different programming styles. For this particular small granularity job that information is not needed; hence the CAL mechanism is more amenable. The difference in the implementation is O(1) clocks for the CAL version and O(100) to O(1000) clocks for the multitasking library, depending on mechanism used.

With this algorithm the work partition is well matched for two processors. The overhead in multitasking is essentially wiped out as the problem size increases, and for small problems it is not a great penalty. The implementation comes very close in the limit to attaining the optimum performance.

We now focus on another algorithm for dealing with a system of equations where the matrix is symmetric and positive definite: Cholesky factorization. The algorithm can again be described in terms of a matrix vector operation, but in this case because of symmetry only half the matrix is referenced.

As before, we can graphically describe the algorithm at the ith stage as follows:



In this case, the algorithm does not divide naturally into two parts as in the previous algorithm. To distribute the work between the processors, we take the naive approach of just splitting the matrix vector operation in half, letting one processor take the left half and the other processor the right half, and then put the two parts back in a sequential part. Table III shows the results.

As before, the percentage given here is for the modified code before the multitasking mechanism is put in. The degradation in code performance is the result of the additional subroutine calls to perform the matrix vector product and the fact that one more work array has to be initialized and added to the other half in each iteration. In Table IV, the modified (but without multitasking mechanism) one-processor version is compared against the multitasked two-processor version.

TABLE III
PARALLEL VERSION OF CHOLESKY DECOMPOSITION

n	Degradation from code change (%)	Percentage of parallel code (%)
50	33.1	80.2
100	24.2	85.4
300	8.0	91.9
600	3.5	96.1

TABLE IV
SPEEDUP OF ALGORITHM ON 2 PROCESSORS VS 1 PROCESSOR

n	Optimal	MT Library calls	MT CAL calls	Mean granularity (μsec)
50	1.67	0.76	1.56	30
100	1.74	0.94	1.54	45
300	1.85	1.52	1.85	130
600	1.93	1.80	1.92	400

As in the previous example, the improvement is substantial when two processors are used to partition the work and perform the task. For large orders, the parallel program reaches the optimal rate of speedup.

This experiment is relevant to the case when there are more than two processors. The matrix vector operation will then be split across the matrix in a fashion similar to that followed here, perhaps going to a block matrix scheme to achieve the desired number of parallel tasks. We expect to observe the same trend when we can perform a similar experiment with more processors.

Note that there is certain amount of fluctuation in between runs on the CRAY X-MP depending on background activities. The numbers we present here should be given a 1–3% tolerance.

CONCLUSIONS

The multitasking concept on the CRAY X-MP-2 has been shown to be advantageous in solving problems with relatively large granularities (that is, when there is more than one millisecond of computation that can be performed in parallel between synchronization points).

For problems with small granularity with a reasonable degree of parallelism that can be exploited, at least from the standpoint of linear algebra solvers where the granularity is in the order of microseconds, the situation can be handled just as efficiently. In general, the main sources of overhead (other than synchronization, load imbalance, and code change) are memory contention and operating system service. The speedup figures of the examples presented here show that the interference from these two factors is insignificant. Our experience demonstrates that multitasking with small granularity jobs is very promising on the CRAY X-MP-2.

In the anticipation of more processors, it will be interesting to see the performance speedup for these small granularity problems through the use of more processors. As we pointed out earlier, the overhead incurred by synchronization, especially by using hardware instruction directly, is minimal in this size of problems. The deciding factor in performance will eventually be the degree of parallelism of the code.

For the LU factorization code, the degree of parallelism is about 61.1% for $n = 50$ and 95.3% for $n = 600$. The anticipated speedup by using four processors should be approximately 1.7 for $n = 50$ and 3.5 for $n = 600$.

For the Cholesky code, the degree of parallelism is about 80.2% for $n = 50$ and 96.4% for $n = 600$. The anticipated speedup by using four processors should then be 2.4 for $n = 50$ and 3.6 for $n = 600$.

REFERENCES

1. Chen, S. C. Large-scale and high-speed multiprocessor system for scientific applications—CRAY-X-MP-2 series. NATO Advanced Research Workshop on High Speed Computation, Nuclear Research Center, Julich, West Germany, June 1983.
2. Dongarra, J. J., and Hiromoto, R. A collection of parallel linear equations routines for the denelcor HEP. ANL/MCS-TM-15, Sept. 1983.
3. Dongarra, J. J., and Eisenstat, S. C. Squeezing the most out of an algorithm in Cray Fortran. ANL/MCS-TM-9, May 1983.
4. Hsiung, C. C., and Butscher, W. A new numerical seismic 3-D migration model on the CRAY X-MP, SIAM Conference on Parallel Processing and Scientific Computing, Norfolk, Va., 1983.
5. Larson, J. L. An introduction to multitasking on the CRAY X-MP-2 multiprocessor. *IEEE Comput.*, in press.
6. Stewart, G. W. *Introduction to Matrix Computations*. Academic Press, New York, 1973.
7. Johnson, P. M. An introduction to vector processing. *Comput. Design* **17**, 2 (Feb. 1978), 289–197.
8. Baer, J. L. A survey of some theoretical aspects of multiprocessing. *ACM Computing Surveys* **5**, 1 (March 1973), 31–80.
9. Enslow, P. H., ed. *Multiprocessors and Parallel Processing*. Wiley-Interscience, New York, 1974.
10. Enslow, P. H. Multiprocessor organization. *ACM Computing Surveys* **9**, 1 (March 1977), 103–129.
11. Padua, D. A., Kuck, D. J., and Lawrie, D. H. High-speed multiprocessors and compilation techniques. *IEEE Trans on Comp.* **C-29** (Sept. 1980), 763–776.
12. Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B. and Wolfe, M. Dependence graphs and compiler optimizations. Proc. 8th ACM Symp Principles Programming Languages, Jan. 1981, pp. 207–218.
13. Hwang, K., and Briggs, F. A. *Computer Adventure and Parallel Processing*. McGraw-Hill, New York, 1984.