THE IMPACT OF PARALLELISM ON SOFTWARE

DR. GARY W. COBB
TEXAS INSTRUMENTS INCORPORATED
AUSTIN, TEXAS

There seems to be a tug-of-war raging between computer procurement technical evaluation committees, computer designers and scholars of computer science and numerical analysis over the issue of parallelism in computations. Caught in the middle seems to be the user community and the compiler writers. For the scope of this paper, the term "user community" will be assumed to be Fortran programmers who are involved in solving problems that require large computer resources, e.g. plasma research, weather prediction, ray tracing, seismic analysis, econometric modeling, weapons research, reactor calculation, etc.

Parallelism within computer systems has appeared in almost all major components of the more recent computers, such as the ASC, B6700, B7700, CDC 7600, CRAY 1, HITACHI 8800, IBM 195, ILLIAC IV, MU5, STAR, STARAN, etc. The impetus for this movement comes from the emphasis placed by this user community on the fast throughput of single large-resource jobs versus the relatively slow increase in the speed of computer logic and components. Of course, the first signs of parallelisms manifested themselves in I/O channels, interleaved memory systems, cache memories, as well as dedicated processors for operating system functions, separate from the user's processor. With the possible exception of parallel I/O channels, these parallelisms were installed in most Fourth Generation computers in a manner that was "transparent to the user", and so, for the most part, their existence could be ignored. However, the parallelisms found in the Fifth Generation's array processing computer systems are by no stretch of the imagination "transparent to the user" and cannot be ignored without significant impact upon performance. The subject of the remainder of this paper deals with the reflections of array processors' parallelisms in software and the scope of the array processing computers will be limited to the ASC, STAR, CRAY 1 and ILLIAC IV computers.

The technical evaluation committees representing the user communities that are searching for large-scale computers continue to use the rather classical approach of measuring the cost/performance ratio of prospective array processing computer systems by requiring that the computer vendor compile-link-and-execute a set of Fortran benchmarks which is typical of the committee's installation's job mix. Usually, several time-burning kernels are selected from the installation's production codes, along with a typical I/O sweep program that exercises one or more levels of heirarchical memory. This selection is executed in standalone mode and in batch mode to measure the throughput capabilities of the new computer.

The vendors of the array processing computers usually perform program conversion in order to warp the kernels and I/O sweep programs into moderately optimized solutions to the same problems. This often involves the substitution of parallel algorithms for pointwise (scalar) algorithms. Judgement is them made by the technical evaluation committees as to the depth of the recoding done by the computer vendors on the kernels in order to measure how much of their installation's Fortran code will need to be recoded to exploit the parallelisms available in each computer being measured.

This brings up a very important point. That is, the dynamic range of execution speeds of the array processing computers is much larger than for the scalar computers of previous generations. A pointwise algorithm may be sped up by an order of magnitude or more by converting the algorithm to one that exploits array processing. Furthermore, the conventional measures of performance, such as MIPS (millions of instructions per second) and MOPS (millions of operations per second, excluding loop control and logic) are being replaced by measures of parallelism. In his paper entitled "Multioperation Machine Computational Complexity", David J. Kuck has defined a measure he calls "utilization", U_p, to be $O_p/(PT_p)$ where p is the number of processors that an algorithm is compiled for, T_p is the time required to execute the algorithm in p processors and O_p is the count of the operations that may be spread out among all p processors. It becomes the task of the hardware and the compiler to maximize U_p while still insuring hazard-free execution of the algorithm. Figure 1 illustrates this point.

```
LINE NO.      FORTRAN STATEMENT

   1          COMMON/NAME/A(100,9,7), B(100,9,7),
             1  C(100,9,7), X(100,9,7), Y(100,9,7)
   2          DO 1 L=1, 7
   3          DO 1 K=1, 9
   4          DO 1 I=1, 100
   5          X(I,K,L) = 1.0/X(I,K,L)
   6          A(I,K,L) = X(I,K,L) + Y(I,K,L)
   7          B (I,K,L) = X(I,K,L) - Y(I,K,L)
   8    1     C(I,K,L) = X(I,K,L)* Y(I,K,L)
   9          END
```

FIGURE 1

Let p denote the number of processors available, assuming that each processor is capable of executing any instruction or any array operation, as in the ASC and the ILLIAC IV. The ASC's optimizing Fortran compiler maximizes utilization by determining as many equal-instruction-time, nonhazardous array operations as possible, and, failing that, to split up one or more operations into disjoint array operations of equal length to balance the processing across the processors. Note that lines 6, 7, 8 are independent from one another, but all are dependent on line 5's output, X. Due to the ASC's three-loop hardware-indexing capability for its vector operations, partitioning may be done on any one of the three loops, so that the compiler's selection algorithm involves all compile-time constant lengths for the outermost one that is as closely divisible by p as possible. Figure 2 shows the results of this scheme for p=1,2,3,4 for single and double precision variables. Note that +, - and * are equal-stream-rate instructions in single precision, but in double precision + and - are equal stream-rate and * is about half as fast.

P	Precision	Resulting Array Operations
1	Single or Double	Four vector operations result to be executed sequentially.
2	Single or Double	Six vector operations result; line 5 is partitioned two ways on I, lines 6 and 7 are executed in parallel and line 8 is partitioned two ways on I.
3	Single	Six vector operations result; line 5 is partitioned three ways on K and lines 6,7,8 are executed in parallel.
3	Double	Twelve vector operations result; each of lines 5, 6, 7, 8 is partitioned three ways on K.
4	Single Double	Twelve vector operations result; line 5 is partitioned four ways on I, lines 6 and 7 are partitioned two ways each on I and line 8 is partitioned four ways on I.

FIGURE 2

The STAR also has a stream unit having two independent processors, each of which is partitionable for like instructions. Therefore, for the Figure 1 code, the maximizing algorithm for U_p requires 6 vectors for 32-bit precision and 8 vectors for 64 bit precision, since all arrays are stored continguously in memory. However, if line 4 were changed to read "DO 1 I = 1, 10", a timing trade-off must be made between executing 252 vectors of length 10 or processing and then not storing 90% of the results by use of a control vector on 6 or 8 vectors, as described above. Both cases diminish U_p.

For the CRAY 1 machine, the maximizing algorithm for U_p is obtained by executing 686 vector instructions each of length 64, followed by 7 vector instructions each of length 28. A trade-off similar to the STAR algorithm would need to be made if line 4 were altered to read "DO 1 I = 1, 10", diminishing U_p. For the 64 processing element ILLIAC IV the maximizing algorithm for U_p is obtained by executing 392 array operations each utilizing all 64 PE, followed by 4 operations each utilizing 28 PEs.

The major importance of this analysis on Figure 1 is the understanding it brings up the variety of compiler optimization algorithms that are required to maximize multiprocessor and vector utilization, if some degree of "transparency to the user" is to be maintained. However, the compliment of this problem is precisely what the individual array processing computer users face - that of knowing the central processor architecture, the hierarchical memory system and it's buffering characteristics and the optimization capabilities of the Fortran compiler.

Figure 3 shows some source code which inhibits compiler optimization dramatically. Since compilers classically compile one source module at a time, then there is no hope of collapsing the code in Figure 3, the code "hides" from the compiler the length information so that length tests and array operation setup code, usually scalar code, will be required to be executed each time through the subroutines. IF - tests like in line 4 of SUB1 have plagued array processor compiler writers in the past, but recently some general approaches to generating array instructions for DO-loops with IF-tests in their range have been developed. To recognize parallelism from Fortran, some array

```
LINE NO.          FORTRAN STATEMENT

  1        COMMON/NAME/A(100,9,7), B(100,9,7),
            C(100,9,7), E(100,9,7), F(100,9,7)
  2        L = 0
  3   10   L=L+1
  4        DO 1 K=1, 9
  5        CALL SUB1 (X(1,K,L),100)
  6        CALL SUB2 (A(1,K,L), X(1,K,L),
            Y(1,K,6),100)
  7        DO 1 I = 1,100
  8        C(I,K,L) = X(I,K,L) * Y (I,K,L)
  9    1   CALL SUB2 (B(I,K,L), X(I,K,L),-Y
            II,K,L),1)
 10        IF (L-7)10,10,11
 11   11   STOP
 12        END

  1        SUBROUTINE SUB1 (R,LEN)
  2        DIMENSION R(1)
  3        DO 1000 I=1, LEN
  4        IF (R(I).EQ.0.0) R(I) = 1.E-50
  5 1000   CONTINUE
  6        DO 1001 I=1,LEN
  7 1001   R(I) = 1.0/R(I)
  8        RETURN
  9        END

  1        SUBROUTINE SUB2(R,S,T,LEN)
  2        DIMENSION R(1),S(1),T(1)
  3        DO 2000 I=1,LEN
  4 2000   R(I) = S(I) + T(I)
  5        RETURN
  6        END
```

FIGURE 3

processor compilers key only off the DO statement, so that the non-DO-loop evident in lines 2, 3, 10, 11 of the first source module usually evades detection.

The purpose of the example given in Figure 3 is to demonstrate the "fragilness" of Fortran source code that has been written for an array processing computer, as in Figure 1. An unknowing programmer might replace the Figure 1 code by Figure 3 code, say to avoid a divide check, and time out at ten hours on a job that previously completed in one hour! Of course, a case should be made now that a source code that has been revamped to present a maximum degree of parallelism to an array processor's compiler, will usually execute at a rate equal to or greater than it's previous generation ancestor code on a previous generation scalar processing computer. Hence, the revamped code represents an intrically equal, if not superior, algorithm, regardless of which array processor is used.

A higher level of parallelism has been carried over from the Fourth Generation computers to those of the Fifth Generation - that is asynchronous I/O and other operating system services. The Control Data system's concept of the "station", IBM system's concept of asynchronous data channels and the ASC system's concept of distributed tasks among the virtual processors of the peripheral processor are three examples of these parallelisms. As one might expect, in the absence of a Fortran standard for asynchronous I/O, the Fortran syntax required to invoke this feature is quite varied from CDC's BUFFER IN/OUT to IBM's FORTBDAM to ASC's QDAM. Technical evaluation committees are again faced with subjective decisions to make in evaluating the flexibility, useability and performance

performance of these parallel operating system services.

In summary, it has become clear over the past decade that two major factors influence the execution speed of a large Fortran program; they are the selection of the algorithm and the interface between the user's coding technique and the compiler's optimization characteristics. Many scholars of Numerical Analysis have begun to rethink some of the algorithms for solving some of the more classical problems of science, and a solid base of literature is taking form to address the parallelism of these algorithms. This research along with the new array processing computers should form an adequate stepping stone to the succeeding generation of computers. And, to complete the tug-of-war, the technical evaluation committies have a difficult, and often times very subjective, job of selecting a new large-scale computer, whose cost/performance ratio may be low for existing small-memory-scalar-algorithm models, but which will increase in its performance as knowledgeable users begin new models, using parallel algorithms.

Acknowledgement

The author wishes to thank Al Riccomi, Dan Sifferd and Dick Roth for their discussions and, of course, the typist Mary Baugh.

References

Butler, Margaret K. "Prospective Capabilities in Hardware", ERDA-wide Conference on Computer Support of Environmental Science and Analysis, July 9-11, 1975, Albuquerque, N. M.

Cobb, Gary W. "What a Vector Machine Can Do For a Meteorological Problem", presented at the Symposium on Complexity of Sequential and Parallel Numerical Algorithms, ONR, Carnegie-Mellon Univ., Pittsburgh, Penn., May 16-18, 1973.

Owens, Jerry L. "The Influence of Machine Organization on Algorithms", presented at the Symposium on Complexity of Sequential and Parallel Numerical Algorithms, ONR, Carnegie-Melon Univ., Pittsburgh, Penn., May 16-18, 1973.

Wedel, Dorothy. "Fortran For the Texas Instruments ASC System", Proc. of Conf. on Programming Languages and Compilers for Parallel and Vector Machines, SIGPLAN Notices, Vol. 10, No. 3, PP 119-132, March 1972.