USING AN IBM MULTIPROCESSOR SYSTEM

A.L. LIM and D.B. SOLL

IBM Corporation, Data Systems Division

Kingston, New York, U.S.A.

ABSTRACT

This paper presents an overview of the evolution of IBM multiprocessor products and current product offerings. Programming language solutions for parallel program execution are discussed and their suitability for the implementation of scientific and engineering applications is assessed.

A set of groundrules governing the provision of parallel programming capabilities is proposed. Results from applying these groundrules to real scientific application codes executing on a multiprocessor are provided.

Open questions for future investigation are posed.

SECTION 1: OVERVIEW OF IBM MULTIPROCESSING

The trend in large computer systems has been driven by desires for higher performance, increased reliability and lower cost per unit of computation. Much has been written about advances in computing technology from the mid 1940's to today. Figure 1 highlights some of the major accomplishments in large system developments.

In fact, parallelism, exploited in machine design, has been a significant contributor to these improvements. For example, even uniprocessor configurations may incorporate parallelism at various levels such as interleaving of storage, signal multiplexing, pipelining of functional units and high speed cache. Today, large, high end systems consist of multiple processor complexes.

Multiple processor configurations may be utilized in various ways, depending on the manner in which the processors are connected. Loosely coupled processors, that is, processors which are connected to each other by channel-to-channel attachments, for example, need not be identical, since there is no sharing of storage, channels, or controls. Loosely coupled processor systems have existed since the 1960's.

Tightly coupled processors, on the other hand, have a much greater connectivity and generally share memory, channels and other functional control. This arrangement also implies a high degree of interaction between the processors, which can be utilized to achieve either optimal throughput or optimal turn-around. Tightly coupled processors such as the IBM 370/168 MP

HISTORICAL REVIEW

FACILITY	TIME
◆ FIRST ELECTRONIC COMPUTER	1944-46
◆ FIRST STORED PROGRAM	1949
◆ CONTROL UNITS AND CHANNELS	1958
◆ INTERLEAVED STORAGE	1961
◆ PARALLEL/PIPELINE PROCESSORS	1961
◆ LOOSELY COUPLED SYSTEMS	1964
◆ HIGH SPEED BUFFERS (CACHE)	1969
◆ TIGHTLY COUPLED SYSTEMS	1969

FIGURE 1. HISTORICAL REVIEW

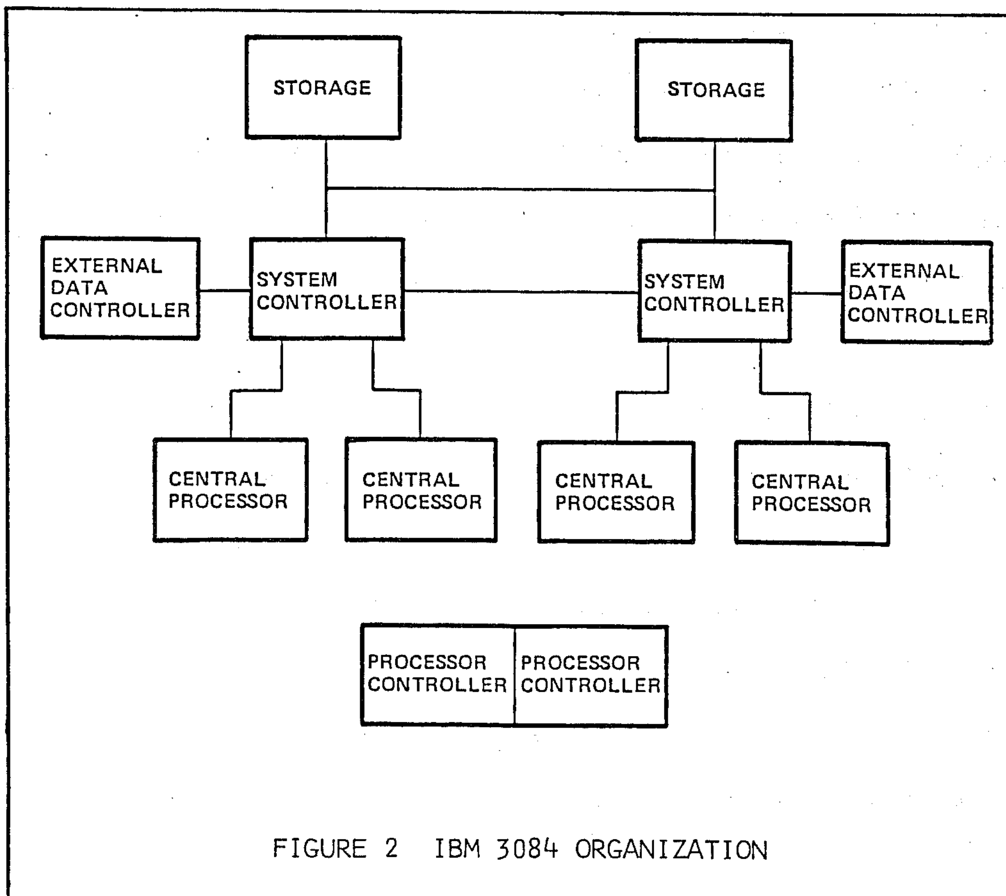

FIGURE 2 IBM 3084 ORGANIZATION

were available in 1975. Today, the IBM 3084 is a 4-way, dyadic based multiprocessor complex, as shown in Figure 2, featuring 48 to 128 Mbytes of real storage and 48 channels.

Dividing the work between processors at, say, the JOB level, means that the configuration is utilized as a multiprocessor, in which throughput is the primary consideration. Allocating the work within a single job to multiple processors, then, indicates that the configuration is used as a parallel processor, with job turn-around as the primary concern. In fact, the IBM 3084 can be used in both modes of operation.

Concurrently with the development of hardware, there has been a similar evolution of operating system architecture. Operating systems evolved from simple batch systems to sophisticated, multi-purpose systems supporting mixed interactive and batch environments with virtual storage and multiple address space management. The IBM 3084 is supported by MVS (Multiple Virtual Storage) and MVS/XA (Extended Architecture) operating systems. MVS/XA gives the user virtual and real addressing to 2 Gbytes via 31-bit addressing, whereas MVS provides 16 Mbytes of virtual addressing.

Both MVS and MVS/XA contain integrated support for loosely and tightly coupled multiprocessing. Their libraries include macro instructions for multi-task management such as ATTACH and DETACH of Tasks, POST and WAIT of Events, and ENQUEUE and DEQUEUE of Resources. Further information is contained in Arnold et al (1974). A higher level user interface that is callable from FORTRAN is available as a program offering. This facility is called the Application Program Performance Extender, see IBM (1984).

It is the purpose of this paper to report on a series of experiments in using an IBM tightly coupled multiple processor configuration such as the IBM 3084 as a parallel processor. Prior to this discussion, a review of the history of programming language support for parallelism is appropriate.

SECTION 2: PROGRAMMING LANGUAGE SOLUTIONS FOR PARALLELISM

Programming language features for expressing parallel execution in programs were actively proposed twenty years ago and actual implementations were realized in widely used compilers. The concept of FORK and JOIN was introduced in the early 1960's. One of the earliest proponents of parallel processing was M.E. Conway. In Conway (1963), he said, "Parallel processing is not so mysterious a concept as the dearth of algorithms which explicitly use it might suggest." He proposed the following instructions to permit an adequate specification of parallelism:

FORK A	Fork to location A and the next statement.
FORK A,J	as above, in addition set counter J to 2, denoting 2 parallel tasks.
FORK A,J,N	as above, except set counter at location J to N, where N denotes N parallel tasks.
JOIN J	Decrement counter at J by 1. If result is zero jump to location J+1; else detach task.

```
JOIN J,B          If counter at J equals 1, set it to zero and
                  jump to location J+1; else jump to location B.
```

A. Opler (1965), proposed extensions to high level languages such as FORTRAN to facilitate parallel processing. The two new statements were "DO TOGETHER" and "HOLD" and they are used as shown in the following example:

```
START    DO TOGETHER   TASK1, TASK2, TASK3 (MEET)

TASK1    statement
         statement        PATH 1
         statement
           . . .
TASK2    statement
         statement        PATH 2
         statement
           . . .
TASK3    statement        PATH 3
           . . .
MEET     HOLD
```

Fairly explicit rules for DO TOGETHER and HOLD were defined covering notions of sharing variables and nesting.

J.P. Anderson (1966), proposed extensions to ALGOL60 based on the work of Conway and Opler. He introduced three additional statements, namely, TERMINATE to explicitly deactivate tasks, OBTAIN to have exclusive use of selected variables, freeing them from interference by other tasks, and RELEASE to free variables previously locked onto by an OBTAIN.

Similar constructs were proposed by Dennis and Van Horn (1966) as shown in their example program which evaluates the product of two vectors, A and B:

```
Begin   Real Array A(/1:n/),B(/1:n/)
        Boolean w; Real s; Integer t;
        Private Integer i;
        t: = n ;
        For i: = 1 Step 1 Until n Do          Create n processes with
           FORK e;                            instructions at label e
        Quit;
     e: Begin Private Real x ;
        x: = A(/i/) * B(/i/) ;
        LOCK w;                               For each e-process, add x
        s: = s + x;                           to s only when no other
        UNLOCK w;                             e-process is accessing s
        JOIN t,r;                             When t tasks complete,
        Quit;                                 branch to r
        End;
     r:
 End;
```

The LOCK and UNLOCK pair basically solves the 'critical section' problem by providing protection for common data from simultaneous access and change by two or more processes. Similar solutions were provided by Dijkstra (1965) namely, the semaphore concept; by IBM OS/360 with ENQUEUE and DEQUEUE macro-instructions for resource sharing and WAIT and POST macro-instructions for process synchronization.

Programming languages that have practical implementations of parallel programming constructs include Burroughs' extended ALGOL, PL/I, Concurrent PASCAL and ADA. PL/I provides task initiation via a CALL statement of the form:

```
CALL P(a1,a2,...,an) TASK(taskname) EVENT(eventname) PRIORITY(N);
```

The call initiates a new task, whose execution is represented by the procedure P. The status of the task may be interrogated via the event variable. This task has an associated priority of N which provides explicit scheduling of task execution when there are more tasks than processors.

The WAIT statement is the PL/I version of a JOIN. It has the form:

```
WAIT (E1,E2,...,En) K;
```

where Ei are event names and K is an integer number or null. This statement will hold program execution at this point until any K events signal completion or until all events signal completion, if K is null.

In summary, programming language features for parallel processing have been in existence for twenty years. Practical implementations have also been available, although they are mostly for ALGOL-based languages.

SECTION 3: GROUNDRULES FOR PARALLEL PROGRAMMING

Robert E. Kahn of the Defense Advanced Research Project Agency, in his paper "A New Generation in Computing", (see Kahn (1983)), said,

"...It is difficult to estimate how much of the existence of sequential machines has affected the way we think about problems. It clearly affects the way we implement them. A given algorithm implemented on a multiprocessor system will surely bear little resemblance to its counterpart on a sequential machine. And multiprocessor-based LISP systems need bear no relation to a single processor version. Conversely, we still know very little about multiprocessor architectures, concurrent programming, or parallelism. Our understanding of task decomposition strategies is limited and current languages, both natural and computer based, are inadequate to represent concurrency. ....Some problems do not lend themselves to much parallelism and cannot make good use of a multiprocessor system. An outstanding generic research question is how to determine the amount of performance speedup (or parallelism) possible in an application..."

Although Dr. Kahn was primarily addressing the world of artificial intelligence, his comments are appropriate for scientific computing in general.

Parallel programming approaches and concepts that have existed for two decades have had little impact. Specifically, the scientific/engineering programmer or analyst is concerned with the problem at hand, and views the intricacies of detailed control and communication required to allow the desired parallel execution as a burden. Furthermore, while the trend is towards higher level descriptions of problems, these approaches are actually proceeding toward low level programming techniques involving functions that are part of operating system technology.

It would certainly be an immense task to re-solve and re-implement most of the scientific programs that are in use today. How do we make progress toward answering the question of how much parallelism is possible in applications? The approach proposed here is to experiment with converting existing serial applications to run parallel on multiprocessors. For this to be successful, some groundrules have to be established. They are:

- Control of parallelism should be accomplished through a minimal number of new commands or statements.

- The new commands or statements should conform to or be compatible with the current syntax and semantics of FORTRAN.

- The algorithm structure and execution flow of the application should be preserved as much as possible.

- System parameters and multitasking control complexity should be hidden from the end user.

The above groundrules were applied in constructing an experimental environment to permit parallel execution of FORTRAN application programs using currently available hardware and software, including operating system and compiler. Results for converting three VS FORTRAN programs for the IBM 3084 running MVS/XA are discussed below.

SECTION 4: APPLICATION CASE STUDIES

Three scientific application programs were analyzed and modified to execute in parallel. These applications were adapted for parallelism following the groundrules described above. The modified versions of these programs were measured on an IBM 3084 to determine the elapsed time speed-up achieved when executed on a 4 processor system.

APPLICATIONS MODIFIED

The first application program, TBLADE, is a turbine blade analysis application. The application uses a finite difference algorithm to solve for the three dimensional air flow between the blades of a rotating turbine stage. The application requires a 38 megabyte private region to execute, and runs for approximately 24 hours when executed on an IBM 3084 using a single processor.

The second application program, VA3D is a fluid dynamics application from a national laboratory using methods developed by Pulliam (1978) and Beam (1978).

The program simulates the subsonic or supersonic flow field around a hemi-spherical-nosed body. It solves the Euler equations or the Navier-Stokes equations with a thin-layer approximation using a three dimensional, implicit, approximate-factored algorithm. This is a form of the Alternating Direction, Implicit (ADI) technique as reported in Douglas (1964).

The third application program, BOAST, is a three phase black oil reservoir simulation program developed by the U.S. Department of Energy and reported in Keplinger (1982). The program can simulate both the primary depletion and the secondary recovery operations in a two- or three-dimensional black oil reservoir. The solution uses the Implicit Pressure, Explicit Saturation (IMPES) method to solve for the fluid flow in a reservoir. The pressure and saturation equations for the oil/gas/water systems are approximated using a finite difference method. The resulting system of linear equations are solved using an iterative Line Successive Over-Relaxation (LSOR) technique.

REORGANIZATION EFFORT

Each of the applications described above was modified for parallel execution. The major portion of the reorganization time was spent understanding the original application, locating the data and functional independence, and introducing the appropriate forks and joins in the application. Once this was accomplished, the process of creating parallel subroutines was straight-forward and mainly mechanical. Code sequences that could execute in parallel were packaged as separate subroutines to be scheduled for dispatching by the operating system.

The level of effort required to perform these modifications is indicated by Table 1. For each of the three application programs considered, the size, in lines of code, (LOC) of the original source program, the amount of code which actually performed the operations which could be executed in parallel, the percentage represented, and the resulting number of newly created parallel subroutines are listed. The modifications of these three application programs was performed without changing the original algorithm, rather, the inherent parallelism of each application was exploited.

Application	Size (LOC)	Parallel LOC	% LOC PARALLEL	# Parallel Routines
TBLADE	3500	677	19	13
VA3D	3800	1350	36	15
BOAST	4200	1021	24	5

Table 1: Application Code Modified for Parallelism

RESULTS

The results of executing the modified application programs on a 4 processor IBM 3084 are summarized in Table 2. All of the measurement runs were performed on a dedicated system with no other jobs executing concurrently with the measurement jobs.

For each application program, both the original version and the parallel version of the program (using 4 subtasks) were executed, and the elapsed time for the program was measured. The speed-up ratio is the ratio between the elapsed time for the original serial run and the elapsed time for the 4-way parallel run for each application.

Application	Serial Time (Min)	Parallel Time (Min)	Speedup Ratio	Percent Parallel	Scope of Run
TBLADE	80.39	31.84	2.52	81	200 Time Steps
VA3D	70.18	21.24	3.30	93	200 Time Steps
BOAST	50.23	15.50	3.24	92	3000 Simulated Days

Table 2: Application Measurement Results

The percentage of parallelism for each application was calculated from the serial and parallel elapsed times using the following simple model:

$$\text{Time serial}/\text{Time parallel} = 1/(\,(1-P) + (P/N)\,)$$

or

$$P = (N/N-1) * (1 - \text{Time parallel}/\text{Time serial})$$

where P is the effective parallelism and N is the number of processors being used.

This simple model provides a rough estimate of the amount of work that was done with some degree of parallelism and takes into account the overhead that was introduced.

SECTION 5: CONCLUDING REMARKS

The results as shown in Table 2 are significant with respect to performance improvement for the applications concerned. More importantly, they were achieved with little investment in program modification. In fact, between 1 and 2 person months were spent in each case with program behavior analysis time included.

Other experiments with parallelism for scientific applications have been conducted within IBM. An example is the work of Blaine and Wang (1976) on the IBM 370/168, at the IBM Palo Alto Scientific Center. Another effort is by Meck (1984) on the IBM 308X.

The experiences above provide considerable motivation to further investigate the issues in advancing the state of the art in the exploitation of parallelism. Some questions deserving exploration are:

- Should general purpose scientific processing be limited to low levels of parallelism or conversely, are highly parallel machines inherently application specific?

- How is the effective cost for running a program in parallel to be determined?

- Is automatic detection of parallelism possible and can efficient tools be constructed?

- What extensions should be made to FORTRAN to facilitate parallel algorithm construction?

- How do we complement the efforts to exploit parallelism and vectorization?

Though it is premature to conclude that our groundrules are valid, the results do provide evidence of their effectiveness. Much remains to be done to advance the exploitation of parallel or multiprocessing systems by scientific applications.

ACKNOWLEDGEMENT

The authors wish to acknowledge the contributions of J.M. Gdaniec and R.J. Sahulka for making the above experimental work possible.

REFERENCES

Anderson, J.P. (1966), "Program Structures for Parallel Processing", Comm. A.C.M., Vol.8, No. 12, December 1965.

Arnold, J.S., Casey, D.P., and McKinstry, R.H. (1974), "Design of Tightly Coupled Multiprocessing Programming", IBM System Journal, No. 1, 1974.

Beam, R.M. and Warming, R.F. (1978), "An Implicit Factored Scheme for the Compressible Navier-Stokes Equations.", AIAA Journal, Vol. 16, No. 4, April 1978.

Blain, R.A. and Wang, H.H. (1976), Private Communication, IBM Palo Alto Scientific Center, California 94304.

Conway, M.E. (1963), "A Multiprocessor System Design.", Proceedings Fall Joint Computer Conference 24, Spartan Books, Baltimore, 1963.

Dennis, J.B. and Van Horn, E.C. (1966), "Programming Semantics for Multiprogrammed Computations.", Comm. A.C.M., Vol.9, No. 3, March 1966.

Dijkstra, E.W. (1965), "Solution of a Problem in Concurrent Programming Control.", Comm. A.C.M., Vol. 8, No. 9, September 1965.

Douglas, J. and Gunn, J. (1964), "A General Formulation of Alternating Direction Methods.", Numerical Mathematics, Vol. 6, No. 5, 1964.

IBM Documentation (1984), "Application Program Performance Extender, 5798-DNL".

Keplinger and Associates (1982), "BOAST: A Three Dimensional Three Phase Black Oil Applied Simulation Tool (Vers. 1.1).", DOE/BC/10033-3.

Kahn, R.E. (1983), "A New Generation in Computing", IEEE Spectrum, Vol. 20, No. 11, November 1983.

Meck, D.L. (1984), "Parallelism in Executing Fortran Programs on the 308X: Systems Considerations and Application Examples", IBM Kingston, NY, Laboratory Technical Report, TR-21.942.

Opler, A. (1965), "Procedure-Oriented Language Statements to Facilitate Parallel Processing.", Comm. A.C.M., Vol.8, No. 5, May 1965.

Pulliam, T.H. and Steger, J.L. (1978), "On Implicit Finite-Difference Simulations of Three Dimensional Flow.", AIAA paper 78-10, January 1978.