MIMD PROCESSING AND THE DENELCOR HEP

By

David F. Snelling

And

Burton J. Smith

INTRODUCTION

The weather community has long provided the forefront of computational science in that their need for high-speed computation tends to grow as order n^3 on the size of the problem.

As a result the weather community first entered into the use of vector processors and were shortly followed by the rest of the scientific community. This has also been the case with the multi-processor industry; the European Centre for Medium Range Weather Forecasting has taken early membership in the community of multi-processors with their Cray XMP dual processor system.

With the increase in the number of multi-processors available on the market and the fast-growing need for high-speed computation in the MIMD industry, it is important to consider the issues involved in multi-processing such as synchronization, create processing, process termination, control of large numbers of parallel processors and questions as to the number of parallel streams versus size and computational power of the processors executing these streams. In view to addressing some of these issues this paper will discuss briefly the Heterogeneous Element Processor architecture from Denelcor as an example of a very efficient high-performance MIMD architecture. Based on the architecture of the HEP we will further discuss some aspects of parallel computation and how they are particularly related to the weather industry.

THE HEP ARCHITECTURE

The Denelcor HEP is a modular multi-processor mainframe with 3 types of modules: PEM (Process Execution Module) and Data Memory Module, MSS (Mass Storage System). These are connected to each other via a high-speed packet switching network called, appropriate enough, the Switch.

A HEP system may have 1 to 16 PEMs each capable of running at 10 million instructions per second. The PEM consists of a self-contained program memory of 1 to 8 magabytes. A 16 PEM HEP system therefore could have 128 megabytes of program space (this is independent of the data memory). Each PEM contains 2048 general purpose registers and 4096 system constants. Constant memories are used in the same way as registers except that they are read only to user programs. They contain frequently used values like the small integers, pi and other math library constants.

Up to 128 data memory modules may be included in the system allowing for a total memory of 1 Gigabyte. The memory is organized into 64 bit words with 8 error correction bits. The memory may be addressed as words (64 bits), half-words (32 bits), quarter-words (16 bits), and bytes. This memory is one continuous address space and may be interleaved or blocked by module.

The MSS is a solid state buffer that is addressable as part of the data memory and is used as a buffer for disk transfers, a swap space for rolled out programs, and/or a solid state disk. There may be up to four MSSs in a HEP system each with a maximum space of 128 megabytes.

All the above modules are interconnectedby the Switch. The Switch is composed of nodes (the number of which increases with the complexity of the system). Each node has 3 bidirectional ports with a bandwidth of 80 megabytes per second per port. To illustrate, when a load data memory instruction is executed in the PEM, a request message is inserted into the Switch; is passed from node to node; has data added to it at the data memory; is passed back node to node; and is returned to the PEM. At each node along the way a preset routing map determines which exit port will lead, with the shortest path, to the data memory. The optimal routing from any module to another is set at system initialization time, and hence a PEM may be partitioned out of the network for routine maintenance while other PEMs in the system continue to operate. The same holds for other modules in the system.

Each PEM is itself a pipelined parallel processor. Each operation is subdivided into 8 segments and separate parts of the hardware handle each segment. With the exception of divide and data memory accesses, all machine

instructions require eight stages. Unlike most pipelined machines, the HEP need not be executing the same kind of instruction in all segments of the pipeline as in SIMD (Single Instruction Multiple Data stream) computer architectures.

When an instruction stream or process is executing, an instruction from it enters the pipeline, passes through all the stages, and 8 cycles later the next instruction from that process enters the pipeline. This leaves 7 pipeline stages unused. These 7 stages are available to other processes which run in parallel with the original process. Control of these parallel processes is achieved by hardware queues. For each process in the machine there is a PSW (Process Status Word). A PSW is comparable to the program counter in a conventional Von-Neumann machine. It contains the address of the next instruction to execute and some other information pertaining to the process to which it belongs. Each PEM in a HEP system has 128 PSWs (processes). These PSWs are queued by the hardware for execution.

The final topic in this basic architecture discussion is that of process synchronization. If two processes are acting as producer and consumer, some form of synchronization is necessary to avoid one outrunning the other. In the HEP every location in data memory has an access bit associated with it. This bit tells whether the location is empty or full. The access state of a memory location may be tested and changed indivisibly by use of the special access modes in the memory addressing scheme. In the example below subroutine ABC is sending a stream of values to subroutine XYZ. XYZ is consuming them and the transfer is protected from either process overrunning the other. The example is written in Fortran 77 with the parallel routines provided with the HEP system.

```
      program doitforever
      external abc, xyz
      call create (abc,x)
      call xyz (x)
       :
       :
      subroutine abc (x)

10    continue
      t=bigfunction()          generate the value t
      call awrite (x,t)        write x only if empty and set full
      goto 10
      end

      subroutine xyz (x)
10    continue
      t=aread (x)              read x only if full and set empty
      call uset (t)            use the value t
      goto 10
      end
```

x, when used in this way, is called an asynchronous variable.

PARALLEL PROGRAMMING TECHNIQUES

In parallel processing there are two basic approaches to handling parallel streams: The first is called the fork-join in which parallel streams are created, perform their assigned work, and then are joined back together to form a serial stream of execution. The other alternative is up-front creation of many parallel streams. Once the streams are created work is passed around among the various executing streams throughout the execution of the program.

Both the schemes have their advantages. Fork-join first of all provides very clear and precise definition of where the parallelism is and also provides a more simplified mode of programming. On the other hand, if the particular machine in question has relatively inefficient create processing, the overheads involved can limit the size of parallel streams allowed. For example, if a machine is inefficient at parallel process creation then only very large segments of the program can be efficiently parallelised. The second alternative, that of up-front process creation, has advantages and disadvantages. When a process is created at the beginning it is not known what routines it may actually execute. As a result the parallelism is not apparent to the programmer or to someone debugging the program. A point of synchronization somewhere in the program, a barrier of some sort would indicate that at this point some form of synchronization is taking place and there would be no indication as to what other routines may be involved. The corresponding situation in a fork-join can only be a fork-join statement in an isolated place in the program. At this point all synchronization would be apparent to someone debugging or trying to understand the program. The key trade-off between using fork-join process creation and up-front process creation is the trade-off between efficiency and readability. As a result some form of measure of the overhead involved in synchronization is required in order for program designers to determine which form of process creation is most suited to the application.

One such measure has been put forward by Dr. Roger Hockney of Reading University with his $S-\frac{1}{2}$. $S-\frac{1}{2}$ is a measure of the amount of overhead required to perform a parallel process creation. To be more precise it is a measure of the amount of parallel computation required for a particular architecture to make use of half the number of instruction streams being considered.

For example, in a HEP1 program with two instruction streams, sixty floating point operations are required to guarantee half utilization of the two instruction streams. As a result $S-\frac{1}{2}$ is the overhead for creating another instruction stream. The $S-\frac{1}{2}$ for the HEP1 processor running 14 instruction

streams is 820. The S-½ for Cray XMP dual processor running two instruction streams is 5700. (Reference Dr. Hockney's paper in this publication).

Given the large difference in the efficiencies of the two machines mentioned above it will be of interest to describe how the two generate their parallel instruction streams.

In the Cray XMP there is a system routine called Task-Start or Task-Create that actually generates another virtual job within the Cray XMP dual processor, and then the scheduling between the two processors is determined by the operating system. On the HEP, however, the approach is much simpler. The execution stream that will be doing the create performs a score-boarded store into a demon location. This uses the empty/full facility of the HEP architecture. Concurrently with this stream is another stream already set out by the system at load time.

This stream then becomes the second instruction stream. As a result some of the create processing itself is running in parallel, and the number of processes that can be created is much greater, and the efficiency much higher than on a system that involves operating system control and operating system scheduling of parallel streams.

Once many instruction streams have been created it would be nice if all of the streams could execute independently throughout the execution of a job. This, however, is rarely the case and some form of synchronization between instruction streams is virtually always required. As a result various different schemes for controlling the process synchronization have been developed. Two will be discussed here:

One is pre-scheduling in which what work a particular stream does is determined prior to the creation of that stream. The other scheme is called self-scheduling in which processes are created at the on-set and then work allocated dynamically as a program runs.

In the pre-scheduling scheme it is ultimately the programmer who decides what instruction streams will execute what code. As a result issues such as load-balancing and efficiency of parallel processing are left entirely up to the user and not necessarily a function of the actual load on the machine at a given time. Whereas, in self-scheduling, when the process becomes available (i.e. it has just finished a previous assignment) it goes to a queue of work and picks up additional computation. In experience it has been found that self-scheduling tends to provide a more efficient load balancing of parallel process execution.

In general it is the feeling of the authors that self-scheduling with fork-join process creation is the most self-explanatory and efficient mode of parallel programming. This, however, does require that the architecture in consideration have an efficient way of first creating parallel instruction streams and second of allocating self-scheduled process modules.

OTHER MIMD ISSUES

There are two major MIMD issues that have become quite familiar to the industry as a result of experience in vector processing.

The first is Amdahl's Law which states that unless the entire program can be run in parallel, total performance cannot be achieved. To the extent that, if a program is 90% parallelizable or vectorizable the program can run a maximum of 10 times faster, in which case the 90% parallelizable section would be running in zero time.

It is well-known that Amdahl's Law cannot be changed or avoided. However, by investigating the contributing factors one can get an idea of how programming practice can avoid Amdahl's Law to some extent. The goal in avoiding Amdhal's Law is to have as much of the program parallelizable as possible. What typically happens in a program once it is parallelized is that different instruction streams wind up doing different amounts of work. The problem is typically called load-balancing, in which case a large segment of code has been allocated to one instruction stream whereas another instruction stream is dealing with a much smaller section. When a smaller section has completed, the longer section must continue running on only one processor. As a result Amdhal's Law comes into play and the program does not run at full efficiency. The load-balancing problem is particularly pronounced when pre-scheduling is used, as dynamic scheduling algorithm on a large number of parallel streams is not possible. The best solution to this problem is to have a large number of smaller parallel code segments executing in a self-scheduling manner. In this case busy processes continue executing while non-busy processes return to the queue and acquire more computation.

In a very large problem one would be able to achieve high degrees of parallelism by using a large number of very small pieces of parallel execution. The problem that arises here is that the synchronization associated with selecting a parallel segment is significant to the total performance of the system. As a result, for any given system, a balance between the number and size of executable grains, and the scheduling algorithm (either self-scheduling or pre-scheduling) must be chosen.

Another issue that contributes to Amdahl's Law is memory latency. As the systems are built with larger and larger memories the memory latency or access time to memory becomes significantly large. Even when facilities for masking the memory latency (i.e. overlapping of instructions and other techniques used by various different architectures) are employed, the memory reference time becomes a

significant factor in the performance of a parallel system. It becomes even more important as the number of parallel streams increases, as memory conflicts will begin to accrue between several different processes in the system.

The Heterogeneous Element Processor from Denelcor has a facility for handling this memory latency issue. The way it works is by ensuring that there is more work to do than there are actual processors in the system. The HEP, in a single processor version, is essentially a 12 instruction stream machine requiring typically 12 instruction streams to keep the processor busy.

By queueing up a larger number of instruction streams, say 20 to 30, the HEP is able to fill-in the time it takes to go to memory with more work. As a result when more processors are added to the HEP system, the increased conflicts and latency associated with a multi-processor system are also masked by creating more instruction streams in the overall system.

Based on a preliminary study of the weather model at ECMWF it appears to the authors that there are some 900,000 potential parallel streams in the standard weather model currently being run at the Weather Centre. It should be noted, of course, that each one of these streams has associated with it enough instructions that it would land within the efficiency range of a fork-join synchronization approach on the Heterogeneous Element Processor. Hence the potential is high for parallelizing weather models to a very high degree on a large large number of processors. As the weather community comes closer and closer to using parallel processing as a production tool there are still many questions that have to be considered.

One issue that would require a little closer examination would be an extension to Amdahl's Law. Amdahl's Law has typically been stated for one or all processors executing. It would be interesting to investigate what happens to Amdahl's Law when one considers the performance achieved when a certain percentage of the code is running with full parallelism, a smaller percentage is running at 50% parallelism and then a smaller section running in serial or scalar mode.

Such a study is currently being carried out by Denelcor Limited in the United Kingdom along with a study into a parallel run time load-balancing package in which, at execution time, the load is balanced dynamically at a very fine grain level.

One issue being addressed by Denelcor Inc. in the United States with regard to their second generation processor, is the development of a parallel memory management system for programs executing many parallel streams simultaneously.

Studies underway at Los Alamos National Laboratory in conjunction with University of California at San Diego are providing tools for global data dependency analysis and a simplified package of portable parallel programming tools.

These tools provide a collection of macro facilities and synchronization modes that allow them to generate code for HEP processors, Cray XMP processors and multiple VAX processors, all using one standard syntax.

Parallel processing provides a rich and efficient mode of achievinghigh degrees of performance in various different applications, the weather industry in particular. However, many questions remain to be considered and one hopes that academic activity in the area continues and that manufacturers and industry continue to co-operate towards that end.