Summary on Shared Memory MIMD computers: Understanding, Impressions and Interpretation

Manh Huynh, Rohan Dawson, Gita Alaghband
Department of Computer Science and Engineering
University of Colorado Denver

The first thing needed to clarify is in the definitions of different terms: multiprocessor, multiprogramming and multiprocessing. Without understanding on these, there is no way to go further. With that in mind, **multiprocessor** is an integrated system consisting multiple processors (2 or more) on the same die, each of them has ability to run different stream of instructions, usually have their own data and instruction cache, and also able to share data with each other. **Multiprogramming** is running separate jobs independently, and share only the resources of the system, such as I/O and storage devices. This must be done through OS and can be done with a single processor. However, this is not what we are interested in the course. Thing is more interesting is to use multiprocessors to an application and contribute to a unified solution. This we call multiprocessing. In multiprocessing, we need all processors share not only system resources but also problem data and synchronization must be done. Other important terms needed to keep in mind is **NUMA** (nonuniform memory access) and **UMA**(uniform memory access) machines, which are about how long does it take you to read/write a memory location. For some locations take longer to access refer to NUMA, same for all refer to UMA.

A **cluster** is a group of shared memory multiprocessor connected with each other, as a combination somehow of distributed (message passing) architecture and shared memory. The real system is also the combination of these type under certain level of consideration. The PDP-11 computer is an example of this.
The connection between nodes should be done in an efficient way to reduce the latency. There many type of internetwork connection, but it seems like hypercube is the best configuration between them because the step to reach other nodes is just $\log_2 N$, where N is the number of nodes.
The **synchronization mechanism** is the most important part for programming in shared MIMD computers. But always, what we want is to maximize the parallelism by reducing the synchronization as much as possible since it is expensive. There are 4 definitions needs to be clear at this point: **software solution, hardware solution, control-oriented and data-oriented**. Software solution uses some of operations that atomic at machine level such as load/store, while hardware solution uses very low level of hardware such as modifying bits in hardware. Control oriented is mostly done using software solution (implementation of critical section) since it needs to control the flow of program, while data-oriented is controlled by status of data. Thus, both software and hardware solutions can be used to implement data oriented synchronization such as produce/consume (software) and lock/unlock(hardware). Another thing we should care about when implementing synchronization is problem of overhead, the lower implementation of software synchronization, busy waiting. From our point of view, there are always overhead in implementation of synchronization, however using hardware or software with support from hardware may keep busy waiting time smaller. Now, another method such as **semaphore** tries to avoid busy waiting by placing them in sleep and waking them up when need. The available threads can be used in other computations. However, the cost of waking up thread is much more expensive. All these need to put into consideration before implementation.

The barrier is an extremely import synchronisation mechanism – without it, writing any parallel procedure would be almost impossible. Whereas the critical section mandates only one thread can execute the area at one time, the barrier has the power to stop all threads that have reached a current point and wait for the threads that are yet to arrive. It is especially important if the designer wishes to be efficient with process creation and minimise the number of forks and joins. In fact, if the program creates n processes when it first starts, and only ever joins them all at the very end – any form of data dependency will make barriers the absolute most important structure in the program.

When it comes to the actual mechanics of process creation, there are a few options based on what platform the code is running on. The most basic way to create a process would be to use the unix fork command – this effectively duplicates the process in memory and starts execution of the child process at the same point as the parent process. I believe this I quite a crude method of doing things, especially because in many cases the child process is far more heavy (has more information) than is necessary for the situation. A much nicer option available on some other systems is to create another process whose only job is to execute a given subroutine. This allows for a much smaller process image in memory and such flexibility can only be a good thing.

This method is quite similar to how SPMD is designed to work. That is many instances of a process or subroutine are executed at the start, and left to compute themselves usually without too much oversight. Although SIMD can look attractive from a performance perspective, practicalities often push us towards the less powerful but more practical SPMD. This is similar to the trade off with regards to assembly level programming allowing for better use of the underlying hardware, but high level language being far more practical for portability reasons.

Inside the code, parallel regions can be either prescheduled or self-scheduled. Like most decisions in parallel computing this is a trade-off – prescheduled requires less overhead but can easily fall victim to poorly balances workloads, whereas a self-scheduled region requires some reasonable overhead, but is able to load balance any workload. In the end, there is no right answer and the designer needs to make a decision based on how they feel about the data.

The two parallel languages we talked about in class are OpenMP and FORCE. Both of these manage to parallelise code in a slick, lightweight manner that avoids heavy forking process creation. Both also use relatively similar constructs in order to get a reasonable fine grained control of the data using relatively high level commands. As high level programmers, we should be thankful to the creators of these languages for saving us from the potential horror that parallel programming could have otherwise become. Looking forward, perhaps writing parallel code in a succinct and easily readable way could become the job of a visual/graphical programming language?