

# Programming Models and Memory Systems

Ian Watson

University of Manchester, UK

## *Introduction*

With multi-core computing now in the mainstream, we finally need to take general purpose parallel computing seriously. There are two fundamental problems:

**General Purpose Parallel Programming** – Writing parallel programs has, until now, been the preserve of the specialist. In areas where it was required (usually HPC) cost was not the most important consideration. Correctness was not unimportant but mission critical software was unlikely to be highly parallel. Complexity was often regarded as a challenge rather than a problem so that the issue of simplifying parallel programming was not a priority. In addition to all this, many HPC programs are too regular to be considered truly general purpose and, as a consequence, are relatively easy to parallelize.

Now we need to think differently. We must take general purpose problems, which may not contain significant amounts of array computation, and work out how to turn them into parallel programs. We want to do this cheaply using less expert programmers and we probably need to be fairly concerned about correctness and reliability. In short, we need some new ideas.

**Extensible Memory Architecture** – Current multi-core systems have only a small number of cores. They usually support a coherent view of shared memory using a bus based protocol which relies on all cores having an instantaneous view of all memory transactions occurring on the bus. Unfortunately this solution does not scale beyond a few tens of cores at the very most. Is there a way to build extensible systems which do not rely on the use of a bus?

These two problems may seem significantly different, but they are related. Programming models are supported by memory models and memory models are supported by hardware structures. It is not necessarily true that simple hardware leads to simple programming models or vice versa, but they need to be examined together.

## *Parallel Programming Approaches*

There are a number of approaches to parallel programming which are strongly advocated by supporters. It is likely that they all have something worthwhile to say but it is unlikely that any of them are the complete answer. Let us examine some of the candidates and some possible claims for them.

**Thread Based Parallelism** – “is well established with real world implementations”. This is true, but there are significant doubts about the complexity of programs using the conventional locks and/or barriers approach. In addition the implementations rely heavily on a coherent view of shared memory which is difficult to scale.

**Current HPC Approaches** – “are well understood and just need a bit of work to make them more general purpose”. Unfortunately most HPC approaches are largely limited to array based data parallelism. Handling complex control parallelism is not the same thing and neither is handling dynamic data structures. Obviously there is a lot of important expertise in this area, but it is not the whole answer.

**Message Passing** – “is easily understood, can be implemented easily on extensible hardware and has formal models to help with correctness”. All of this is true. Unfortunately there is a lot of history which should help us to conclude that this is not the way to program general purpose applications. It either becomes too complex, it does not scale or it is not efficient. There are, of course, problems which are expressed naturally using message passing but again it is not the answer. (remember the Transputer).

**Declarative Programming** – “is easy to understand and easy to parallelize”. This hides a myriad of issues. Simple declarative programming is easy to understand but enthusiasts delight in complex higher order constructs which deter many. Purists also insist that laziness is essential to the approach and this introduces implementation issues which can both kill parallelism and reduce efficiency. However, the biggest impediment to acceptance is the attempt to make declarative programs express things that they are not good at. Shared updateable state, synchronization and I/O are not declarative!

**Transactional Memory** – “makes the use of shared state much easier to understand than the equivalent locking solution”. This is true, but proponents of the approach are complicating it rapidly. Open nested transactions with associated constructs are not easy to understand. Is there a danger here of falling into the same trap as the declarative camp by extending a simple idea to situations to which it is not suited?

What should we make of all this? It is too easy just to say that we should devise a hybrid approach which integrates the best of each. Nevertheless, there are important issues. We should clearly recognize opportunities for data parallelism when it exists. In applications which are naturally message passing (hardware description?) it makes sense to be able to express this. If computations can be described simply and without loss of efficiency using declarative approaches then it makes sense to use them rather than make unnecessary use of state.

Ultimately, the issue comes down to the handling of shared state. This is where both program complexity and implementation problems originate. Maybe we can make progress by considering the memory issues.

## ***Memory Architecture***

As already discussed, current multi-core systems use bus based coherent shared memory and this is used to support thread based parallel programming models which use locks and/or barriers to control shared state.

There is general agreement that this approach is not scalable. One possible solution is to devise other coherent memory implementations possibly using hierarchical busses and/or directory based coherence protocols. However, it may be more profitable to ask the question whether the sort of coherence maintained by current approaches is necessary. Do we really need to ensure that every individual memory write is seen to be coherent across the whole of a parallel memory system? The answer is that we almost certainly do not. Although this approach provides a simple picture of a parallel computation, this is rapidly complicated by the use of locks to control access to this global memory and we have lost the advantage.

The truth is that it is perfectly possible to develop parallel programming models where it is necessary only to establish coherence at less frequent intervals. The Bulk Synchronous Parallel (BSP) model is an example of this as is Transactional Memory. And, of course, there are plenty of cases where updateable shared memory is not required, particularly if we are more aware of not using inappropriate programming styles when they are unnecessary.

Although BSP is a well understood programming model, it has not been widely studied from an implementation viewpoint. There have been many studies of hardware to support Transactional Memory although most of them have used bus based mechanisms, to detect transactional conflict, which may still limit extensibility. Others, which use lazy conflict detection, rely on global broadcasts which may strain the bandwidth capabilities of larger networks. There is much still to do.

## ***Conclusions***

There certainly needs to be a major push in the development of programming styles, tools and probably languages which will significantly ease the task of parallel programming. It is essential that we examine underlying programming models, taking note of research which has been done in the past. But it is also important that we do not get diverted by the purists view that there is only one true story!

This may all seem like a big problem, but it is also an opportunity! We should examine carefully how the properties of programming models influence the requirements of the underlying hardware, particularly memory architecture. It may be that, not only can we ease the task of programming parallel systems, but we can also learn how to make them extensible.