Chip Multiprocessors

Todd Davies

February 11, 2016

Overview

Due to technological limitations, it is proving increasingly difficult to maintain a continual increase in the performance of individual processors. Therefore, the current trend is to integrate multiple processors on to a single chip and exploit the resulting parallel resources to achieve higher computing power. However, this may require significantly different approaches to both hardware and software particularly for general purpose applications. This course will explore these issues in detail.

Syllabus

Introduction Trends in technology, limitations and consequences. The move to multi-core parallelism in programs, ILP, Thread Level, Data Parallelism.

Parallel Architectures SIMD, MIMD, Shared Memory, Distributed Memory, strengths and weaknesses.

Parallel Programming Multithreaded programming, Data parallel programming, Explicit vs Implicit parallelism, automatic parallelisation. The Case for Shared Memory. When to share?

Shared Memory Multiprocessors Basic structures, the cache coherence problem. The MESI protocol. Limitations. Directory based coherence.

Programming with Locks and Barriers The need for synchronisation. Problems with explicit synchronisation

Other Parallel Programming Approaches MPI and OpenMP

Speculation The easy route to automatic parallelisation?

Transactional Memory Principles. Hardware and Software approaches

Memory Issues Memory system design. Memory consistency

Other Architectures and Programming Approaches GPGPUs, CUDA

Data Driven Parallelism Dataflow principles and Functional Programming

Attribution

These notes are based off of both the course notes (`http://studentnet.cs.manchester.ac.uk/ugt/2015/COMP35112/`). Thanks to John Gurd for such a good course! If you find any errors, then I'd love to hear about them!

Contribution

Pull requests are very welcome: `https://github.com/Todd-Davies/third-year-notes`

Contents

1 The need for parallelism

Even though we've been unable to increase the clock speed of processors since around 2005, we have seen the 'power' of processors roughly double every 18-24 months since then in line with Moore's Law. The reason why, is that we have been able to increase the amount of transistors in chips (due to the feature size decreasing), and use the extra ones to provide more processing cores, which are able to process data in parallel. The degree of parallelism is increasing as time progresses.

In an ideal world providing a greater degree of parallelism would merely entail chip designers copy and pasting multiple processor cores onto the silicon, and programmers getting linear performance increases. In practice, there are lots of architectural issues (such as how processors are connected and how they're organised) as well as software issues (how do we make our app run on multiple cores).

As transistors are becoming smaller, we can also make them switch faster. The switching speed is determined by $R * C$, where R is the resistance and C is the capacitance. When we reduce the area of the transistor, C decreases, so in doing so, we make the circuit able to compute faster.

This was fine until 2005, at which point we started to have problems with things like *interconnect capacitance* (the capacitance between neighbouring wires). Also, the power density increased (smaller transistors means more per unit area), which has serious cooling implications. As we approach the theoretical limit of one atom per transistor, any impurity in the silicon becomes a major issue.

We have tried using extra transistors to build more complex single core processors (using Instruction Level Parallelism (ILP)) and by adding bigger caches so that they exhibit lower miss rates, however both of these techniques suffer from diminishing returns. Control statements such as `branches` make us have to periodically throw away all the partially completed instructions in a pipeline, and caches already have hit-rates in the high 90's.

Though we might be able to increase the number of cores on a chip, how does a programmer utilise this extra power? It is relatively easy for an operating system to schedule programs so that they can run on different cores and therefore have true multi-tasking (process level parallelism), but what if we want to make one program run faster by running it over many cores?

1.1 ILP vs TLP

Instruction Level Parallelism and Thread Level Parallelism are two different approaches to utilising parallel hardware, and both can be used at the same time.

In ILP, the processor is able to execute instructions out of order and in parallel, meaning that fewer clock cycles are needed to execute the same number of instructions. This form of parallelism is very limited, and can only be used in certain situations. Vector parallelism is similar, and lets you do things like do four 8-bit additions in one instruction (by splitting a 32-bit word into four 8-bit parts).

In TLP, a program can be composed of separate threads, each being its own sequence of instructions. Many threads can be executing in parallel and since their instructions are independent of each other, can be interleaved on the processor (or run on multiple cores). The output of threaded programs must be deterministic, i.e. for all possible sequences of execution, the output must be the same.

TLP is far more general purpose than ILP (and much more so than vector parallelism, which is only really useful in simple operations like array addition). However, it relies on the programmer finding a way to express an algorithm in a parallel manner.

1.2 Data and instruction parallelism

Flynn classified parallelism as **instruction stream** and **data stream** parallelism.

Data parallelism is when the same computation is carried out on multiple elements of some dataset, usually an array. Vector parallelism is an example of this. Only certain problems amenable to this type of parallelism can be sped up in this manner.

Instruction parallelism is when multiple instructions can be executed in parallel (with no side effects that cause problems to each other).

Different combinations of these types of parallelism have different names:

SISD: Single Instruction Single Data is like a normal program. A serial sequence of executions working on a stream of data, one word at a time.

MIMD: Multiple Instruction Multiple Data is the most parallel one, where there are multiple instruction streams and they can all operate on their own independent stream of data. This is most normal computer chips.

SIMD: Single Instruction Multiple Data is stuff like when you use instructions like `ADD8` to process multiple data elements at once with the same instruction, or a GPU processing lots of pixels at once with the same filter. If it's not a vector instruction, then it's one or more processors working in lockstep to process elements of an array or something.

SPMD: Single Program Multiple Data is a generalisation of SIMD, where different processors execute the same code but don't need to be in lockstep. This is most often how parallel programs are written for CPU's.

1.3 Connecting processors

In order to make them work in parallel, multiple cores of a processor need to be connected to each other, and to memory. There are lots of different ways to do this, and the best way depends on the use case.

The processors can be laid out in a grid. In this case, each processor can communicate with its neighbours, and memory is usually private to each core. In order for cores to access memory in other cores, they must send messages through the grid.

A Torus is a variation of the grid, where each edge is linked to the opposite edge. This makes a doughnut shape (in a logical, not physical sense) and means that fewer steps are needed to communicate between cores.

A bus can be used to connect multiple cores. The memory is usually situated on the bus, and all cores have access to it (though they may also have their own memory instead). Time slicing is used to give equal access to the bus, but can make the bus become a processing bottleneck.

There are, of course, more types of interconnects. Crossbars are where each node is connected to each other node (which has a complexity of $n(n-1)$), but the best ones are usually tree structures or hierarchical buses where the complexity is logarithmic ($nlog(n)$).

1.4 Shared and distributed memory

Shared memory is accessible from all of the cores and every part of the computation, while distributed memory is spread out in different components, and is usually only accessible by the component that owns it. We are considering systems that are either one of these, or the other. Either one of these can emulate the other from a software point of view; it is fairly easy to provide abstraction layers that make a distributed memory behave like a shared one, or impose restrictions on shared memory so that parts are unavailable to certain components. Imposing a foreign memory layout onto the hardware comes at a performance penalty.

Most supercomputers use distributed memory, since its easier to build, provides a higher total communication bandwidth and is more suited to many data-parallel problems. However, programs using data sharing (shared memory) are widely seen to be easier to code than programs using message passing (distributed memory), so there is an overhead involved with distributed memory.

2 Using threads

A thread is a flow of control executing a program, and a process can consist of one or more threads. Each thread inside a process has access to the same address space, and most programming languages provide some method of using threads.

Now, go and look up Java threads (`https://docs.oracle.com/javase/tutorial/essential/concurrency/`) and C's pThreads (`https://computing.llnl.gov/tutorials/pthreads/`).

The easiest form of parallelism to find and exploit using threads is data parallelism. This is where computation is divided into roughly equal chunks, where hopefully, each chunk is independent of the next.

An example of this is multiplying two $n \times n$ matrices. We could use n^2 threads, each computing one element (remember the area of the output matrix is going to be n^2), or we could use n threads and have each thread compute a whole row or column. If we didn't have that many threads (or perhaps making more threads is inefficient), then we could make p threads and have each one compute q columns or rows, where $p \times q = n$.

In Fortran, `DOALL` statements let us execute the body of a loop in parallel.

There are two types of parallelism:

Implicit: This is when the system works out how the parallelise something for itself. This is particularly relevant to functional languages, since many operations (map, reduce etc) are inherently parallelisable.

Explicit: Here, the programmer must have a mental model of the parallelism in his or her head, and specifies exactly what should be done.

A lot of the time, parallel programs are made in a way that blends the two (so you might not have to specify everything explicitly, but you have to give hints to the system as to what should be parallelised and how). An example of this is the `DOALL` statement mentioned above.

In an ideal world, we would have computers that would automatically parallelise programs. However, since dependency analysis is hard to do, this approach is limited. Computers do automatic parallelisation to an extent (e.g. instruction reordering), but must be 100% sure that an operation is safe to parallelise, and the results will be correct.

3 Caches in shared memory multiprocessors

Obviously caches are vital to the efficient functioning of processors. Without them, every memory access would cause the CPU to wait around 200 cycles, and so having a cache is vital to having the CPU run at close to full speed.

However, caches don't just fix the problem; we need to work out how to populate them, and keep the data in them correct. Data that we write to a cache must eventually be written back to memory, and new memory locations need to be loaded into the cache when they're required by the CPU.

We solved these problems in `COMP25111`, however, more problems arise when you consider a multi-core processor. In most multi-core processors, each core has its own cache, and since each cache can potentially hold its own copy of the same memory location, we need to make sure that

they agree with each other about the values of these locations. This is the **cache coherence problem.**

An easy solution to this would be to require that every write would go through to memory straight away, and the other cache(s) in other cores would load the value. This is obviously slow though and there may be bus bandwidth problems. Furthermore, we'd only be getting a benefit from cache-reads, which defeats (half of) the object of the cache!

We can overcome this by making the caches talk to each other. When a new value is written to one cache, the others should invalidate their own cache lines containing this value. This means a write to a cache doesn't need to go straight through to memory, but just flips a bit inside the other caches.

However, when we introduce more state to our caches (such as invalidated cache lines) we also increase the complexity, and need a model to make sure things don't go wrong. Each cache line can be in three states:

Invalid; There might be an address math on this line, but the data is not valid any more. It needs to be fetched from memory again.

Dirty; The cache line is valid, and has been update in the cache since it was loaded from memory. It must be written to memory at some point in the future.

Valid; The cache line matches what's currently in memory.

In order to let caches know what other core's caches are doing, we have them do **bus snooping**. This involves having hardware watch each core's cache and modify the cache independently of the core so that the flags on the cache lines are correct.

Given two cores, the following states are valid and invalid:

	V	D	I
V	T	F	T
D	F	F	T
I	T	T	T

Notice how the table is a mirror image of itself. We can't have two dirty states, since then we won't know which we should write to memory, and we can't have a dirty and valid state (since then, by definition, the valid state is not valid).

There are two types of messages between cores; read requests for a cache line (one core hopes that another core's cache has the cache line so that it doesn't have to go to memory), and invalidate messages.

We can easily extend the protocol we've described beyond two cores; any core with a valid value can respond to read requests (the bus will decide who 'wins'), and invalidate requests work as normal, invalidating the cache line on all cores.

The only extra requirement is that invalidation must happen in one cycle, since we want all cores to have the same view of memory, and if one core receives the invalidation message after another, there will be a period of time where their views of memory will be inconsistent. This gets harder as we increase the number of cores; the bus gets longer and so slows down (signals take longer to propagate, and the clock will have to be reduced).

The impact from this is that the consistency protocol is the biggest limitation when trying to add more cores to a processor. The protocol we have described is called the `MSI protocol` (modified, shared, invalid).