Chip Multiprocessors

Todd Davies

March 24, 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`

Commit: `727aad6d6b11f57add6fbd451b55c60cd80abb8d`
Build: `35`

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[1]. 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 see three problems which stopped transistors from becoming smaller and faster:

Interconnect capacitance: the capacitance between neighbouring wires).

Power density: As the power density increased, cooling became a serious problem; each transistor produces heat as it switches and the number of transistors per unit area dictates how much heat is produced.

Impurities: 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-rate percentages 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 both cases, the end result of the execution is the same as if all the instructions were executed in order.

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). If it often desirable for the output of threaded programs to be deterministic, i.e. for all possible sequences of execution, the output must be the same.

[1] An observation that the number of transistors in processor chips doubles approximately every two years.

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). While ILP is applied automatically to the instruction stream issued to the CPU[2], whereas TLP relies on the programmer finding a way to express an algorithm in a parallel manner; something that is not always achievable for all programs.

1.2 Data and instruction parallelism

Flynn [**?**] classified parallelism as **instruction stream** and **data stream** parallelism, both of which can come into effect at the same time.

Data parallelism is when the same computation can be carried out on multiple elements of some dataset, usually an array. Vector parallelism is an example of this. Only certain problems are amenable to data parallelism and can be sped up in this manner, but doing so is often very easy[3].

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

Flynn gave 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 what happens on most modern computer chips, where the operating system will schedule processes on different cores at the same time.

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[4] 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.

For any given problem, the complexity of parallelising it is proportional to how irregular the parallelism is, and how much data sharing there is between threads of execution (particularly if the threads are writing to the shared data).

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.

[2]Either the compiler can make use of ILP (e.g. instruction reordering at compile time), or the CPU can do it automatically using a technique such as scoreboarding.

[3]E.g. vector addition can be easily parallelised by splitting the vector up into n chunks and assigning each one to a thread, where you have n threads.

[4]GPU's often have upwards of hundreds or even thousands of processing cores, and in order to manage the complexity of assigning work to these cores, will use a combination of SIMD and SPMD techniques. The same program usually runs on each core, but groups of cores execute in lock-step on the same data.

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. All memory accesses have equal access time so long as there isn't too much contrention.

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 busses 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.

More explicitly, **data sharing** is when a program with shared memory space facilitates inter-thread communication by having threads read and write to the shared memory. **Message passing** is when seperate parts of the program communicate by sharing messages (e.g. using a socket).

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; unfortunately, most 'normal' computing problems are irregular and dynamic.

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).

3.1 Other cache coherence protocols

In the previous cache coherence protocols we have discussed, we came across situations where there was unnecessary bus usage (e.g. when one core writes to its cache and makes a cache line dirty, other cores would write their copies back to memory, even though that value would never be used since there was a newer dirty version).

However, the bus is a *critical shared resource*, and we certainly don't want to waste bandwidth on messages that have no effect. We can distinguish between two cases of writing to a cache:

- When the cache holds the only copy of a value, and it's not dirty (if it was dirty, we'd just update it).
- When the cache holds a copy of the value, but there are other copies in other caches.

It is only the first case where we don't need to send an invalidate message, but this is nevertheless a common case. In most multithreaded programs, only a minority of memory locations are shared between threads (and a smaller minority are both read and written to by multiple threads), so the majority of memory locations are unshared. If we split the 'V' (valid) state into two more states, we can account for this case:

- **E** - Exclusive to one cache
- **S** - Shared between multiple caches

This gives us the `MESI` protocol (as opposed to the `MSI` protocol). This is easy to implement on top of the `MSI` protocol; simply set the state to E if the value was read from memory, and to S if it was read from another cache.

Though minor changes in the protocol are required, the only inconsistency is that if all E/S lines are evicted from other caches except one S line, then the S line is now exclusive (even though it is marked as S). This isn't a problem in practice, since it doesn't happen often, and since it's hard to detect, it's just ignored.

The `MOSEI` protocol is a further optimisation, where the M state is split into:

- **M** - Modified; the cache contains a copy which differs from memory, and no other caches contain a copy.
- **O** - Owned; the cache contains a copy which differs from the one in memory, and some other caches also contains a copy, but those are in state S and have the same value as the owner.

With the additional O state, we can share the latest value and don't have to write it back to memory straight away. Only when a cache line in the O or M states is evicted, will any writing to memory be done.

3.2 Directory based coherence

So far, we've looked at methods for letting multiple cores communicate over a single bus, and have assumed that bus communication happens instantaneously. However, as the number of cores increases, the bus capacitance increases and you have to slow it down. This limit is at around 32 cores.

A directory based coherence method is an attempt to make everything less directly connected to get around this limitation. A centralised directory is created that holds information about each cache line (which contains multiple words) in memory.

For each cache line, the directory contains a bit map for which core has a copy, and whether that copy is dirty, which is another bit map (though only one bit is true). Each cache has its own valid and dirty bits, which are used to dictate whether to write back to memory or not. If a core whats to make a memory access, we have the following cases:

Read hit in local cache:
Just read the local value!

Read miss in the local cache:
Ask the directory:

Directory dirty bit is false:
Read the data from main memory into the cache, set the directory presence and local valid bit for that core.

Directory dirty bit is true:
Cause the owner to write back the value to memory, which is also sent to the cache that was asking. The directory dirty bit is cleared, but the directory presence bits are set, as is the local valid bit.

Write miss in the local cache:

- Set the local cache to dirty.
- Depending on the directory dirty bit:

It's false:
Invalidate any cores that are valid for this cache line, set the present bit for that core and set the dirty bit for the directory.

It's true:
Send a message to the core marked as O (owner) to write back (note, this can be optimised out). Clear the owner's present bit, and set if for the writing core. Leave the dirty bit set.

Write hit in local cache:

- If the local dirty bit is set, just update the value,
- Otherwise, update the local cache and dirty bit then:

If the directory bit is not dirty:
Invalidate any cores with the cache line present, then clear the present bits.. After that, set the present bit for the writing core and set the directory dirty bit to true.

If the directory dirty bit is set: Send a message to the owner to update core memory (again, this can be optimised out), clear the owner's present bit and set the writing cache's present bit. Keep the directory bit set.

In this architecture, the directory becomes the bottleneck. To avoid this, we can distribute the directory between different caches, and make each part responsible for part of the address space.

Sometimes in multi-processor systems, the memory is not all in one place and is spread over multiple chips.

Since chips with a directory based protocol are often heavily networked (inside the chip) in order to connect all the sub-components, messages can take variable amounts of time to send, requiring handshake protocols. This means that the CPU can sometimes waste a significant number of cycles waiting for responses from messages.

4 Barriers and locks

4.1 Barrier

The idea of a barrier is that a number of threads should all meet up at the same point. When all threads reach the point, they can continue, but until then, they all wait. It's natural when threads are used to implement data parallelism.

Barriers are often used when there are data dependence (we need to wait for the whole answer before continuing). Take a look at `java.util.concurrent.CyclicBarrier`.

4.2 Locks

Any object can be locked in Java by using as the target of a `synchronized` block, or calling an instance method that is declared as being `synchronized`. Only one thread can lock an object at any given time, and other threads must wait to acquire the lock, or the lock holder executes `wait`.

We want to lock objects so that we can achieve 'correctness'. If two code blocks take a lock on the same object, one should complete before the other starts, which means two threads won't be mutating the same object at the same time, and the normal meaning of the code blocks is preserved.

Sequential Consistency is when:

This is a common interpretation of what it means to be 'correct', but not the only one.

- Method calls should appear to happen one at a time in sequential order.
- Method calls should appear to take effect in the order a thread performs them.

Deadlocks can occur in cases where code tries to acquire more than one lock. 'Lost WakeUp' can happen if one thread sends a `notify` message that is not properly received by other threads, one way to avoid this is by using `notifyAll` instead of `notify` in Java.

The **granularity** of the lock depends on how big the chunk of code that depends on a lock is. If the chunk is large, then it's coarse grained and the parallelism is more limited, and if its too small, then its fine grained, and you may spend too much time obtaining and releasing locks.

Java also has `java.util.concurrency.locks.Lock` for explicit locks, since there are situations where the implicit object locking is not adequate. An example of this is when doing 'hand over hand' locking (e.g. lock one item, obtain the next, lock the newly obtained one, unlock the first etc). This is impossible using the implicit locks, since there is an implicit nesting structure (e.g. a synchronized call is either inside a synchronized block or in a synchronized method call, and you can only get out by exiting the whole block).

5 Hardware support for synchronization

Most shared memory parallel Programming models require that the programmer implements some sort of synchronization logic in order to control how threads access the shared memory.

There are several different ways of doing this, all of which are closely related. Hardware support is usually required for shared memory multiprocessor systems.

5.1 Binary semaphores

A *binary* semaphore is essentially a boolean indicating whether some resource is free; (`1` $\rightarrow$ free, *0* $\rightarrow$ in use).

There are two atomic operations that can be done on a semaphore; `wait(s)` and `signal(s)`. `wait` will block until `s` is equal to 1 (i.e. wait until it's free), then set `s` to 0. `signal` sets `s` to be equal to 1. For example:

A **broken** implementation of `wait` could be:

```
# Move the address of the semaphore into R2
ADR   R2, semaphore

...

wait:
      PUSH LR
      PUSH R1
loop: LDR R1, R2
      CMP R1, #0
      BEQ loop
      STR #0, R2
      POP R1
      POP PC
```

The reason why this wouldn't work, is that if another thread was to change the value of the memory address pointed to by `R2` while were were inside `loop`, then we could get unpredictable results.

In order to prevent this from happening, we need to make sure that `wait` is indivisible, which requires special hardware instructions. Even then, if `semaphore` was cached, then even indivisible waiting methods might get confused, so we need coherence operations in the cache.

A simple solution (often implemented in older processors) is to provide an instruction called `test and set`, which is atomic. This takes a pointer as an argument, and tests if the memory location is `0`. If it is, then the memory location is set to `1`, and if its not, then the processors zero flag is cleared. Now our wait function becomes:

```
# Move the address of the semaphore into R2
ADR   R2, semaphore

...

wait:
      PUSH LR
      PUSH R1
loop: TAS R2
      BNZ loop # loop while *R2 != 0
      POP R1
      POP PC
```

Note that this is the logical opposite of how we defined the semaphore before, but it doesn't matter. Furthermore, though it looks like we've reduced the number of instructions, the `TAS` operation will be slow since it must read and possibly write a value in memory, which requires that memory location to be locked.

If `semaphore` is cached, (which is likely since its going to be a shared memory location), then we're going to be using a lot of bus bandwidth because the processor must lock the snoopy bus for every `TAS` operation since it cannot let other cores write to that memory location. If the value of the semaphore was 1, then the lock was wasted.

A simple solution is to sit in a loop reading the value of the semaphore until it seems to be free, then use a `TAS` operation to obtain the lock. If the `TAS` operation was successful, then return to the calling code, and if it wasn't, then another thread must have got there first and the current thread should continue looping:

```
# Move the address of the semaphore into R2
ADR   R2, semaphore

...

wait:
      PUSH LR
      PUSH R1
loop: LDR R1, R2
      CMP R1, #1
      BEQ loop
      TAS R2
      BNZ loop
      POP R1
      POP PC
```

5.2 Other synchronization primitives

There are other machine level primitives including `fetch and add` (get a memory location, add one and write back), and `compare and swap` (compare a memory location to a value and exchange it for another value if it was as expected). These are read-modify-write instructions and require the snoopy bus to be locked while they execute.

The trouble with these operations, is that they are quite complicated (i.e. like CISC instructions, not RISC instructions), and as a consequence, they don't fit well with modern pipelined processors.

References

[1] Michael J Flynn. Some computer organizations and their effectiveness. *Computers, IEEE Transactions on*, 100(9):948–960, 1972.