MANCHESTER 1824

The University of Manchester

Chip Multiprocessors COMP35112

Lecture 2 - The World of Parallelism

Graham Riley

Directions in Parallel Computing

- The move to parallelism required by chip multiprocessors is relatively recent
- Most current general purpose commercial chips have, around, 16-24 cores (Xeon Phi has 60+)
- Some, aimed at server applications, have more
- Special purpose chips (e.g. graphics GPUs) have from several tens to hundreds of cores
- But general purpose parallel computing is not yet mainstream
- Yet industry is talking of the number of cores doubling every 2 years (new Moore's law?)

The Multicore 'Roadmap'

year, cores per chip, feature size

- 2006, ~2 cores, 65nm
- 2008, ~4 cores, 45nm
- 2010, ~8 cores, 33nm
- 2012, ~16 cores, 23nm
- **2014**, ~32 cores, 16nm
  sharing discontinuity
- 2016, ~64 cores, 12nm
- **2018**, ~128 cores, 8nm
- 2020, ~256 cores, 6nm
- 2022, ~512 cores, 4nm
- **2024**, ~1024 cores, 3nm
- 2026, ~2048 cores, 2nm
  scale discontinuity?
- 2028, ~4096 cores
- 2030, ~8192 cores
- **2032**, ~16384 cores

Existing Parallel Computing

- But parallel computing has been around in some form for a long time
- Many forms have been studied in the research world
- Until relatively recently, most in the commercial world were ‘scientific supercomputers’ used for large data parallel applications
- More recently, highly parallel machines have been developed for commercial ‘server’ (web & database) Data Centre applications, such as search engines and financial processing

Where Do Chip-multiprocessors Fit?

- Most previous parallel systems have not been housed within a single chip
- Many are just collections of (high bandwidth) networked conventional processors
- However, most of the previous work on parallelism is relevant to developing new ‘on-chip’ systems
- Therefore, there are several aspects of prior parallel hardware and software systems which need to be studied

Flynn's Taxonomy

- An early attempt (published in 1972) to classify different sorts of parallel machine
- Expressed in terms of whether the *Instruction Stream* and/or the *Data Stream* are parallel
- A serial processor is **SISD** which is short for **S**ingle **I**nstruction (stream) **S**ingle **D**ata (stream)
  - i.e. a single serial sequence (stream) of instructions is operating on a stream of data, one element (word) at a time

Flynn's Taxonomy (cont.)

- **MIMD** (**M**ultiple **I**nstruction **M**ultiple **D**ata) is the category into which modern processors (chips) fall
  - A collection of cores (CPUs) each executing their own serial sequence of instructions and each operating on a separate stream of data

- A third important category is **SIMD** (**S**ingle **I**nstruction **M**ultiple **D**ata)
  - A collection of cores (CPUs) all performing the same sequence of instructions on multiple streams of data in 'lockstep'

SIMD Uses

- Special SIMD machines were built in the 1970s to solve simple data parallel problems
  - ICL DAP
  - ILLIAC IV
  - Thinking Machines' Connection Machine (CM1, CM2)
- Widest current use of SIMD is in graphics applications
  - GPUs (graphics cards) make extensive use of SIMD
  - Single general purpose CPUs have SIMD instructions (MMX, SSE, AVX, 3DNow) to work on packed data
- Will return to GPUs later in this course

SPMD

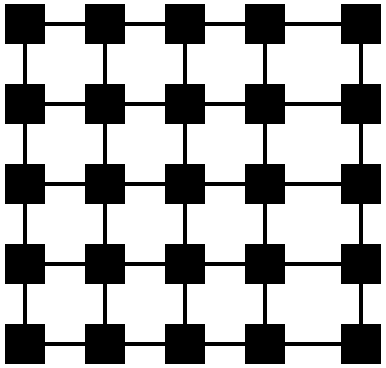
- **S**ingle **P**rogram **M**ultiple **D**ata
- A slightly more general version of SIMD
- Processors all execute the same code, but each on different data
- However, processors do not need to be exactly in ‘lockstep’, i.e. all executing exactly the same instruction at the same time
- Used in data parallel machines built from independent processors

Interconnection

- Processor cores need to be interconnected in order to form a parallel machine
- They must also be connected to memory
- The way they are connected often influences the form of computation that they will support (SIMD, MIMD etc.)
- Although there are a wide variety of ways that have been tried, the fact that chip multiprocessors are on a single integrated circuit may restrict possibilities

A Grid

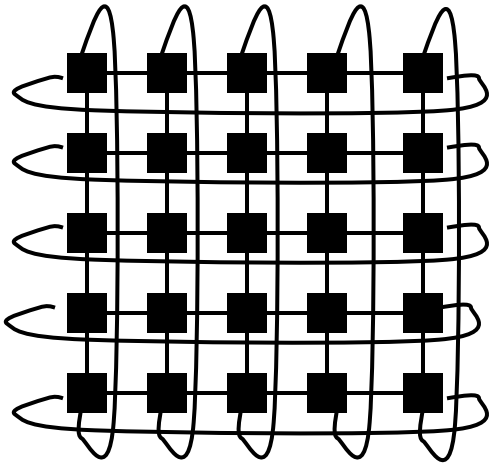
- A fairly obvious 2-dimensional way



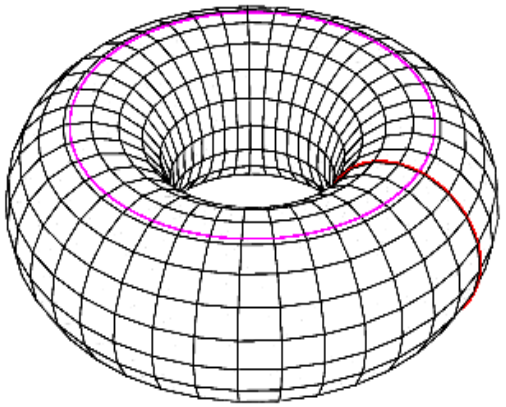
- Processors can communicate with neighbours
- Memory is usually private to each core
- Staged communication required beyond immediate neighbours

Torus: A Grid with 'Wraparound'

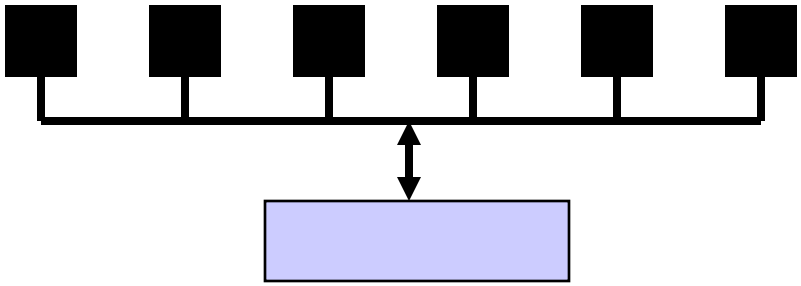
- More symmetrical structure, enhances communications



- Can be viewed as a toroid



Bus Interconnection



- Simple to build
- But single interconnection has to be shared using ‘time slicing’
- Memory can be local to each CPU/core
- But also can add shared memory to bus
- All communications have equal access time
  - as long as there is no contention

Other Interconnects

- Lots of other physical possibilities – crossbars, hypercubes, trees, rings, hierarchical busses, MINs etc.
- Other major variation is
  - Circuit switched: a physical connection is established between source & destinaton
  - Packet switched: data is grouped into one or more blocks which may be buffered, take different routes, maybe arrive out of order
- Basic reason for “sharing discontinuity” (see slide 3)
- Will not go into all the detail

Shared vs. Distributed Memory

- Probably one of the most important distinctions in parallel computing
- **Shared**: memory is accessible from every part of the computation
- **Distributed**: memory is distributed among parts of the computation and (usually) only accessible from one part of the computation
- Obviously it is possible to have a system with some shared and some distributed, but we will consider only one or the other

Hardware View

- What do we mean by "part of the computation"?
- We can take a hardware view:
  - **Shared Memory**: the memory is physically organised so that it can be accessed from any processor/core
  - **Distributed Memory**: the memory is physically connected to only one CPU/core and only that core can access it (directly)
- We will, for the moment, ignore the issue of access time, i.e. shared memory accesses may not all take the same time

Software View

- Or we can take a software view:
  - **Data Sharing**: a program has global memory which is accessible from any part (parallel thread) of the program. Communication between parallel parts can take place by writes to and reads from that global memory
  - **Message Passing**: separate parts of the program (parallel processes) have local memory on which they can operate, but communication between parts can only take place by sending messages between the parts

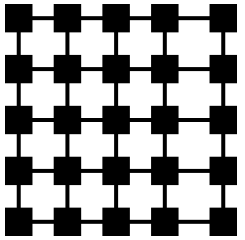
Programming Models

- The software view available to the programmer is usually referred to as the **programming model**
- You are already familiar with the traditional serial programming model – one memory which can be accessed globally by a single thread of computation
- In this serial programming model, the scope of variables can be restricted, but this is invariably a software implemented restriction on top of a globally addressable hardware memory

Parallel Programming Models

- Can usually be classified into Shared Memory (sometimes called **Data Sharing**) or Distributed Memory (often called **Message Passing**)
- Within these categories there are many variations – will explore later
- It is important to remember that this is a software viewpoint, distinct from the hardware viewpoint, so it is better to use distinct names
- In practice, either model can be implemented on either type of hardware; which combinations make sense?

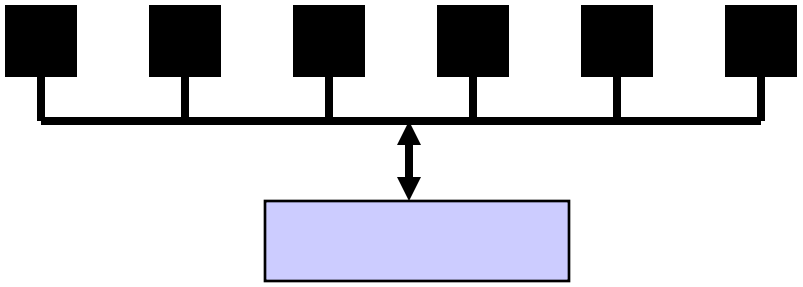
Data Sharing on Distributed Memory



- Consider a grid layout with private memories per core – obviously message passing?
- Code running on a core can only communicate with another by sending a message
- But assume a run-time system on each core
  - ensures messages can be relayed to allow any core to any core communication
  - is able to access its own memory in response to an incoming message and send a reply message
- Allows the ‘simulation’ of shared memory

Message Passing on Shared Memory

- Any core can write to a shared memory location which can be read by another core
- So an area of memory can be used as a communication buffer between separate threads of computation
- Clearly a straightforward implementation of a message passing parallel programming model is possible



But Efficiency?

- Data Sharing simulation on Distributed Memory hardware will clearly be slow if we have lots of remote memory accesses
- Message Passing on Shared Memory hardware should not be too inefficient
  - But, if we only had local communication, total bandwidth available on Distributed Memory hardware could be much higher
- It depends on what we want, but generally it makes sense to match the programming model to the hardware

Which is Better?

- Distributed Memory hardware is often easier to build (later) and can have higher total communication bandwidth
- If our problem is suited to Message Passing – often true of simple data parallel computations – then we will be better off with Distributed Memory hardware
- E.g. Currently, NONE of the world's fastest 500 computers (www.top500.org) are Shared Memory – but all are solving data parallel problems
  - Top500 dominated by multicore + Xeon Phi or GPGPU-like systems (Nov. 2017)

Which is Better? (cont.)

- But there is a wide consensus that (except for very simple cases) Data Sharing parallel programming is easier than Message Passing parallel programming
- The less regular the problem and the more dynamic its structure, the harder Message Passing programming becomes
- Unfortunately, ‘general purpose’ parallel computing is almost certainly going to require irregular and dynamic structured programs

Summary

- Message Passing, Distributed Memory systems are simple and fast for the right kinds of problem
- Shared Memory hardware is more difficult to build, but better supports Data Sharing programming models which are easier to use for general purpose computing
- So most general purpose hardware, in particular chip multiprocessors, is Shared Memory
- But how easy are these to program?