Multi-core systems

COMP25212  System Architecture

Dr. Javier Navaridas
Multi-Cores are Everywhere

• Many processors in desktops/laptops/smartphones/tablets are advertised as ‘dual core’ or ‘quad core’

  – What does this mean?

  – Why is it happening?

  – How are they different?

  – Where are they going?

  – Do they change anything?
Moore’s Law

• Intel’s cofounder Gordon E. Moore conjectured in 1965 that the number of transistors per processor would double roughly every two years
  – This would be translated to doubling processor performance every roughly 18 months

• Driven mainly by shrinkage of circuits
  – more transistors per chip
  – shorter connections
  – lower capacitance

• This “law” has been true for about 40 years but is arriving to an end
  – Number of transistors doubling rate still holds – until when?
  – Performance cannot keep the pace
Processor Evolution and Trends


C. Moore, Data Processing in ExaScale Class Computer Systems, Salishan, April 2011
Performance Evolution

Single-Threaded Floating-Point Performance
Based on adjusted SPECfp® results

~60% per year

~20% per year

From Imgur, LLC
Improving Single-core Performance at the Architecture Level

- **Caches** minimise memory access impact
- **Pipelines** and **superscalar** increase instruction level parallelism
- **Out-of-order** and **Multithreading**: more independent instructions
- Other mechanisms (**branch prediction**, **forwarding**, **register renaming**) minimise the impact of hazards

- But we have seen that all of them have limited scalability
  - Scaling up beyond a certain point is not practical
  - Hardware, power and cost increases more than linearly for most of them
  - ... but performance increases sublinearly and is limited by applications
The End of “Good Times”

• We cannot extract more performance from a single-processor architecture

• Slowdown for several reasons
  – Power density increasing (more watts per unit area) - cooling is a serious problem (Power Wall)
  – Limited parallelism within applications (ILP wall)
  – Memory does not get faster at the same rate as processors (the Memory wall)
  – Architectural innovation hitting design complexity problems
  – Smaller transistors have less predictable characteristics (unreliability)
The Power Wall

• Power dissipation depends on clock rate, capacitive load and voltage

\[
\text{Power} = \text{Capacitive Load} \times \text{Voltage}^2 \times \text{Frequency}
\]

– Increasing clock frequency means more power dissipated
– Decreases in voltage reduce dynamic power consumption but increase the static power leakage from transistors
  • Leakage is becoming the main contributor to consumption due to transistor shrinking

• We have reached the practical power limit for cooling commodity microprocessors
  – We can not increase clock frequency without expensive cooling
  – Inappropriate cooling may lead to burn the device
  – Even more constrained for portable devices (smart phones, tablets)
Effects of Transistor Shrinking on Leakage
The ILP Wall

• The implicit parallelism between the instructions in 1 thread (i.e. number of independent instructions) is limited in many applications

• **Hardware** also imposes some limitations
  – Number of instructions that can be fetched per cycle
  – Instruction window size

• This reduces the effectiveness of all the mechanisms we have studied
  – Increasing the number of instructions we can run per unit of time is not useful if we can not find enough instructions to be executed
The Memory Wall

• From mid 1980’s to mid 2000’s
  – CPU speed increased over 50% per year
  – Memory Speed increased by 10% per year

• Fetching data from memory limits processor utilization
  – CPU use in memory intensive applications can be as small as 10%
A solution is replication

- Put **multiple cores** (CPUs) on a single chip
- Use them **in parallel** to achieve higher performance
- Simpler to design than a more complex single processor
- Need more computing power – just add more cores
  - Is it that **easy**?
How to Put Them Together?

• Could have independent processor/memory pairs – distributed memory
  – Each core has its own, independent memory
  – Communication/synchronization needs to be coded explicitly

• At the software level the majority of opinion is that shared memory is the right answer for a general purpose processor
  – All processors have the same view of memory
  – Communication/synchronization is implicit – programming is easier
  – E.g. 61-core Xeon Phi (2012)

• But, when we consider more than a few cores, shared memory becomes more difficult to attain
How Can We Use Multiple Cores?

- Small numbers of cores can be used for separate tasks
  - e.g. run the OS on one core, a virus checker on another and a browser in yet another one

- If we want increased performance on a single application we need to move to parallel programming
  - Shared memory - OpenMP
  - Message passing - MPI
  - Independent threads - pthreads

- General purpose parallel programming is hard
  - Consensus is that new approaches are needed
There Are Problems

- We do not know how to engineer extensible memory systems
- We do not know how to write general purpose parallel programs
- Parallel programs are difficult to debug
- Parallel programming methods do not fit with existing serial processor designs
- Power, ILP and Memory walls still apply
Summary

• Multi-core systems are here to stay
  – Physical limitations
  – Design costs

• The industry did not want to come but there is no current alternative

• One of the biggest changes for our field
  – General Purpose Parallel Programming must be made tractable
Architecture of Multi-Core Systems
Traditional Structure – "Historical View”

(Processor, Front Side Bus, North Bridge, South Bridge)
Typical Multi-core Structure

- **Main Memory (DRAM)**
- **Memory Controller**
- **L3 Shared Cache**
- **L2 Cache**
- **L1 Data**
- **L1 Inst**
- **On Chip**
- **Input/Output Hub**
- **Graphics Card**
- **Input/Output Controller**
- **QPI or HT**
- **PCIe**
- **Motherboard**
- **I/O Buses (PCIe, USB, Ethernet, SATA HD)**
Intel Core i7 (Sandy Bridge Extreme)
AMD Kaveri APU (Steamroller)
IBM Power8
Apple A9 (iPhone 6s)
Simplified Multi-Core Structure

- **core**
  - L1 Inst
  - L1 Data

- **Interconnect**

- **Main Memory**

On Chip
Data Coherency and Consistency
Applications on Multi-cores

• Processes – operating system level processes e.g. separate applications
  – Normally do not share any data – separate virtual memory spaces
  – If they do, communication is explicit through I/O (storage, network)

• Threads – parallel parts of the same application sharing the same memory – this is where the problems lie
  – Memory coherency: ensure changes are seen everywhere
  – Memory consistency: ensure correct ordering of memory accesses
Memory Coherence

• What is the coherence problem?
  – Core writes to a location in its L1 cache
  – Other L1 caches may hold shared copies - these will be immediately out of date

• The core may either
  – Write through to main memory
  – Copy back only when cache line is rejected

• In either case, because each core may have its own copy, it is not sufficient just to update memory
  – We need to ensure that any core having a copy will get the updated value
  – This can become very expensive

• Memory coherence is not enough to address all kinds the problems
Memory Coherence

- On Chip
- Interconnect
- Main Memory

str r0, X
Memory Consistency

- Consistency: The model presented to the programmer of when changes are seen

- Multiple copies are consistent if:
  - A read operation returns the same value from all copies
  - A write operation updates all copies before any other operation takes place

- Some times we need to read and then update a variable as a single **atomic** operation
  - No other thread can read or write that variable while this operation happens
Example of Consistency Problem

- Imagine we want to count the number of threads are alive within a given process

  Each thread executes: \( \text{count} = \text{count} + 1 \)

**What are we expecting?**  **What could happen w/o consistency?**

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reads ( \text{count}=0 )</td>
<td></td>
<td>Reads ( \text{count}=0 )</td>
<td></td>
</tr>
<tr>
<td>Writes ( \text{count}=1 )</td>
<td></td>
<td>Writes ( \text{count}=1 )</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sequential Consistency

- L. Lamport
  - “the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program"

- Informally
  - memory operations appear to execute one at a time
  - operations of a single core appear to execute in the order described by the program
Sequential Consistency

- Sequential Consistency is not the most stringent memory model
- It provides the behaviour that most software developers expect
- Computer architectures and Java use relaxed consistency models
- The compiler has to insert special instructions in order to maintain the program semantics
  - **Fence**: All memory accesses before the fence need to complete before start executing the ones after (out-of-order)
  - **Barrier**: All threads need to reach the barrier before any of them can continue execution
  - **Lock**: Only one thread can proceed to the atomic section
Synchronization Examples

• **Fence:**

  Thread A
  \[ x = (a+b+c)^2 \]
  fence needed!
  done = 1

  Thread B
  if (done == 1)
  print(x)

  Without the fence, an out-of-order processor could execute ‘done=1’ before storing the updated value of ‘x’.
  Thread B would print an **incorrect** value!

• **Barrier:**

  Thread A
  a = f(x,y)
  Barrier needed!

  Thread B
  b = g(x,y)
  Barrier needed!

  Thread C
  c = h(x,y)
  Barrier needed!

  Thread D
  d = i(x,y)
  Barrier needed!
  total = a + b + c + d

  Without the Barrier, we can not ensure that Thread D will read all the updated values of a, b and c, so ‘total’ will possibly end up with an **incorrect** value.
Synchronization

• Locks are very important synchronization mechanisms as others can be implemented based on them

• In the example above
  – Using Locks to access count will ensure the expected behaviour
Lock

• How do we implement a lock?
  – Regular read and write operations

```c
read flag
if (flag==0) {// lock is free
  write flag 1
} else {// wait until lock is free
  goto Label // We need hardware support to do this atomically
}
```

• Does it work? Do we have everything that we need?

Both proceed to the atomic section !!!
ISA support for Synchronization

- Atomic compare and swap instruction
  - Parameters $x$, old_value, new_value
  - If $[x] == \text{old\_value}$ then $[x] = \text{new\_value}$
  - Return $[x]$

- Load-linked and store-conditional instructions
  - LL $x$ - hardware locks the cache line corresponding to $x$ and returns its contents
  - SC $x$, new_value – hardware checks whether any instruction has modified $x$ since LL, if intact the store succeeds. Otherwise, it leaves unmodified the contents of $x$

- Transactional Memory
  - Execute the atomic section assuming there will be no conflicts
  - After completing execution check for conflicts
    - If there were no conflicts commit the changes
    - If there were any conflict roll back and start over
Memory Coherence Systems
Overview of Coherence Systems

• Hardware mechanisms that ensure our architecture provides a coherent view of data stored in memory

• Require extra information about cache lines and many decision to be taken
  – Are there other copies? Where?
  – What to do when the value changes?
  – How to deal with shared data?

• How we share information about the caches lines
  – Snooping protocols: all caches are aware of everything that happens in the memory system
  – Directory protocols: cache line information is stored somewhere and changes are updated there
Snooping protocol
Snooping Protocols

- Each cache ‘snoops’ (i.e. listens continually) for activity concerned with its cached addresses.
- Normally implemented using a centralized structure, typically a bus, so all communication are seen by all.
- Snooping Protocols can be implemented without a bus, but for simplicity we will consider a shared bus.
Simplest Snooping Protocols

Write Update

1. A core wanting to write grabs bus cycle and broadcasts address & new data as it updates its own copy
2. All snooping caches update their copy
Simplest Snooping Protocols

**Write Invalidate**

1. A core wanting to write to an address, grabs a bus cycle and sends a ‘write invalidate’ message which contains the address.
2. All snooping caches invalidate their copy of that cache line.
3. The core writes to its cached copy.
4. Any shared read in other cores will now miss and get the value from the updated cache.

![Diagram showing write invalidate protocol]
Update or Invalidate?

- In both schemes, the problem of simultaneous writes is taken care of by bus arbitration
  - Only one core can use the bus at any one time
- Update looks the simplest, most obvious and fastest, but:
  - Multiple writes to the same word (no intervening read) need only one invalidate message but would require an update for each
  - Writes to same block in (usual) multi-word cache block require only one invalidate but would require multiple updates
- Remember Write Through vs Copy Back?
Update or Invalidate?

- Due to both spatial and temporal locality, the previous cases occur often.
- Bus bandwidth is a precious commodity in shared memory multi-core chips.
- Experience has shown that invalidate protocols use significantly less bandwidth.
- We will only see in detail the implementation of invalidate protocols.
Implementation Issues

- In both schemes, knowing if a cached value is not shared (no copies in another cache) can avoid sending any messages.

- Invalidate description assumed that a cache value update was ‘written through’ to memory. If we used a ‘copy back’ scheme (usual for high performance) other cores could re-fetch incorrect old value on a cache miss.

- We need a protocol to handle all this.
MESI Protocol (1)

- A practical multi-core invalidate protocol which attempts to minimize bus usage
- Allows usage of a ‘copy back’ scheme - i.e. L2/main memory is not updated until a ‘dirty’ cache line is displaced
- Extension of the usual cache tags, i.e. invalid tag and ‘dirty’ tag in normal copy back cache
MESI Protocol (2)

Any cache line can be in one of 4 states (2 bits)

- **Modified** – The cache line has been modified and is different from main memory – This is the only cached copy. (cf. ‘dirty’)

- **Exclusive** – The cache line is the same as main memory and is the only cached copy

- **Shared** – Same value as main memory but copies may exist in other caches.

- **Invalid** – Line data is not valid (as in simple cache)
MESI Protocol (3)

- Cache line state changes are a function of memory access events.

- Events may be either
  - Due to local core activity (i.e. cache access)
  - Due to bus activity – as a result of snooping

- Each cache line has its own state affected only if the address matches
MESI Protocol (4)

• Operation can be described informally by looking at actions in a local core
  – Read Hit
  – Read Miss
  – Write Hit
  – Write Miss

• More formally by a state transition diagram (later)
MESI Local Read Hit

• The line must be in one of MES

• This must be the correct local value (if M it must have been modified locally)

• Simply return value

• No state change, no message needed
MESI Local Read Miss (1)

• No other copy in caches
  – The core waits for a memory response
  – The value is stored in the cache and marked E

• One cache has an E copy
  – The snooping cache puts a copy of the value on the bus
  – The memory access is cancelled
  – The local core caches the value
  – Both lines are set to S
MESI Local Read Miss (2)

- Several caches have a copy (S)
  - One cache puts copy value on the bus (arbitrated)
  - The memory access is cancelled
  - The local core caches the value and sets the tag to S
  - Other copies remain S

- One cache has M copy
  - The snooping cache puts its copy of the value on the bus
  - The memory access is cancelled
  - The local core caches the value and sets the tag to S
  - **The source (M) value is copied back to memory**
  - The source value changes its tag from M to S
MESI Local Write Hit

- **M** – Line is exclusive and already ‘dirty’
  - Update local cache value
  - No state change, No message required

- **E** – Line is exclusive but not dirty
  - Update local cache value
  - Change E to M, No message required

- **S** – Other caches have not dirty copies
  - Core broadcasts an invalidate on bus
  - Snooping cores with an S copy change S to I
  - The local cache value is updated
  - The local state changes from S to M
MESI Local Write Miss (1)

• No other copies
  – Value read from memory to local cache - bus transaction marked RWITM (read with intent to modify)
  – Local copy state set to M

• Other copies, either one in state E or more in state S
  – Core issues bus transaction marked RWITM
  – The snooping cores see this and set their tags to I
  – The local copy is updated and sets the tag to M
MESI Local Write Miss (2)

- Another copy in state M
  - Core issues bus transaction marked RWITM
  - The snooping core sees this
    - Blocks the RWITM request
    - Takes control of the bus
    - Writes back its copy to memory
    - Sets its copy state to I
  - The original local core re-issues RWITM request
  - This is now simply a no-copy case
    - Value read from memory to local cache
    - Local copy value updated
    - Local copy state set to M
M consistency:
- **Invalid**
- **Shared**
- **Modified**
- **Exclusive**

**Transitions:**
- **Read Hit**
- **Write Hit**
- **Mem Read**
- **Write Miss**
- **Read Miss(sh)**
- **Read Miss(ex)**
- **RWITM**
- **Invalidate**

**Bus Transaction:**
- Blue box: "= bus transaction"
MESI - snooping cache view

- Invalid
- Modified
- Exclusive
- Shared

- Mem Read
- Invalidate
- RWITM

● = copy back
Example of MESI

Sequence of Memory Accesses from cores

Cache status

Core 0: LDR r7, A
Core 2: LDR r3, A
Core 1: STR r0, A
Core 2: LDR r5, A
Core 1: STR r0, A
Core 3: STR r2, A

Main Memory
Example of MESI

Core 0: LDR r7, A

Core 0

I
Read
???

MEM
READ

Core 1

I
???

Core 2

I
???

Core 3

I
???

A

‘A’ cache line

Main Memory
Example of MESI

Core 0: LDR r7, A
Core 2: LDR r3, A
Example of MESI

Core 0: LDR r7, A  
Core 2: LDR r3, A  
Core 1: STR r0, A

Core 0

Core 1

Write Miss

Core 2

Core 3

‘A’ cache line

Main Memory

S  A

S  A

I  ???

A
Example of MESI

Core 0: LDR r7, A
Core 2: LDR r3, A
Core 1: STR r0, A
Core 2: LDR r5, A

Copy back

Main Memory
Example of MESI

Core 0: LDR r7, A
Core 2: LDR r3, A
Core 1: STR r0, A
Core 2: LDR r5, A
Core 1: STR r0, A
Example of MESI

Core 0: LDR r7, A  
Core 2: LDR r3, A  
Core 1: STR r0, A  
Core 2: LDR r5, A  
Core 1: STR r0, A  
Core 3: STR r2, A

Core 0: LDR r7, A  
Core 1: STR r0, A  
Core 2: LDR r5, A  
Core 3: STR r2, A

Copy back  

Main Memory
Example of MESI

Core 0: LDR r7, A
Core 2: LDR r3, A
Core 1: STR r0, A
Core 2: LDR r5, A
Core 1: STR r0, A
Core 3: STR r2, A

‘A’ cache line
Main Memory
Example of MESI

Core 0: LDR r7, A
Core 2: LDR r3, A
Core 1: STR r0, A
Core 2: LDR r5, A
Core 1: STR r0, A
Core 3: STR r2, A

Main Memory
MOESI Protocol

A more complex protocol with an extra state (O) to avoid the need of copying back to main memory (write update)

- **Modified**
  - cache line has been modified and is different from main memory - is the only cached copy. (cf. ‘dirty’)
- **Owned**
  - cache line has been modified and is different from main memory – there are cached copies in shared state
- **Exclusive**
  - cache line is the same as main memory and is the only cached copy
- **Shared**
  - either same as main memory but copies may exist in other caches, or
  - Different as main memory and there is one cache copy in Owned state
- **Invalid**
  - Line data is not valid (as in simple cache)
Summary of Snooping Protocols

- Seen in detail MESI
- Overview of MOESI
- There is a plethora of coherence protocols
  - MSI, MOSI, MERSI, MESIF, Synapse, Berkeley, Firefly...
- Rely on global view of all memory activity, usually imply a global bus – a limited shared resource
- As number of cores increases
  - Demands on bus bandwidth increase – more total memory activity
  - The bus gets slower due to increased capacitive load
- General consensus is that bus-based systems do not scale well beyond a small number cores (8-16?)
  - Hierarchy of buses is possible, but complicates things and cannot really be extended much further than that
Directory-based protocols
Directory-based protocols

• Overcome the lack of scalability of snooping protocols by using distributed hardware
  – Sounds familiar?
• The memory system is equipped with a directory which stores information of what is being shared
  – Stored info is more complex than in snooping protocols
    • Local state, Home cache, Remote sharing caches
• Coherence messages are point-to-point
  – Bus is replaced for a network-on-chip
  – More messages are needed per each operation
• Gaining importance as number of cores increases
Simplest directory Protocols

• Cache lines can be in 3 states (similar to MESI/MOESI)
  – I – Invalid
  – S – Shared: coherent with main mem and may be copies elsewhere
  – M – Modified: Value changed and no other copies around

• Directory can be in 3 states
  – NC – Not cached: this cache line is not present in any core
  – S – Shared: This cache line is not modified and is in at least 1 core
  – M – Modified: This cache line is modified and is in exactly 1 core

• Directory stores a ‘sharing vector’ to know which cores have copies of each cache line
  – Each bit represents a core
Example of Directory Coherence

- Cache only has information about the address and state
- Some transitions need to check the global status in the Directory, e.g.
  - Read Miss needs to check the directory to know if the read line is modified or not
  - Writes in S or I need to check the directory to see whether there are other caches sharing and, if so, invalidate each of them
- More messages are typically needed but we got rid of the bottleneck (the bus), so the overall performance is improved (for many cores)
Distributed Directory

- The directory is yet another performance-limiting centralized HW
  - Accessing the directory can become the bottleneck
- That is why many-core implementations (e.g. TILERA, Xeon Phi) use distributed directories
  - Coherence information is now distributed
  - Typically with the caches
- To know where is the information for each line
  - We need to add ‘home’ info into the cache
  - We add an extra hop
Summary of Directory-based Protocols

- More scalable and so, better suited for existing and future many-core systems

- Use point-to-point messages to simplify the interconnect and allow parallel communications

- Core do not have a global view of all the caches so we need to store global state separately (in a directory)

- Control becomes more complex, but overall performance is better for large systems
Summary of Cache Coherence Protocols

- Invalidate protocols tend to minimize messages

- Snooping protocols
  - A family of protocols based around snooping write operations
  - MESI protocol, each cache line can be in a set of states
    - Modified, Exclusive, Shared, Invalid
  - Bus interconnect limits scalability

- Directory protocols
  - The global state is stored into a directory
    - The directory can be distributed as well
  - Communications are point-to-point rather than broadcast
    - The possibility of sending messages in parallel greatly increases scalability
Beware of False sharing

- Two cores accessing different words on the same cache line
- If one of them writes, it invalidates (or updates) the other – large penalty in either case
- Harms performance significantly (Another form of trashing!)
  - If the two cores continuously modify their values, the cache line is going from one to the other all the time. Lots of invalidations (or updates) happen
- This is independent of the architecture or the protocol
  - Parallel compilers need to be aware of and avoid this pathological behavior
On-chip Interconnects
The Need for Networks

• Any multi-core system must clearly contain the means for cores to communicate
  – With memory
  – With each other (coherence/synchronization)

• We have considered mostly buses until now
  – Mentioned others for directory-based coherence

• Point-to-point and NoCs are possible
  – But have different characteristics
  – May provide different functionality
  – Different coherence mechanism
The need for Networks

Shared-memory applications

• The multicore processor needs to ensure consistency and coherence

• Memory **consistency**: ensure correct ordering of memory accesses
  – Synchronization within a core
  – Synchronization across cores – needs to send messages

• Memory **coherence**: ensure changes are seen everywhere
  – **Snooping**: all the cores see what is going on – centralized bus
  – **Directory**: distributed communications; more traffic required, but higher parallelism achieved – interconnection network
The need for Networks
Distributed-memory Applications

- **Independent** processor/store pairs
  - Each core has its own memory, independent from the rest
  - No coherence is granted at the processor level
  - Saves chip area

- Communication/synchronization is introduced explicitly in the code – **message passing**
  - Needs to be handled efficiently to avoid becoming the bottleneck

- The NoC becomes an important part of the design

  - Later replaced by the cache-coherent Xeon Phi (2012)
The need for Networks

• Most of the applications we are expected to run in current multi- and many-core processors require some short of communication
  – What would be the point for having tens of cores if not, we rarely run that many number of applications at the same time

• Multicore systems need to provide a way for them to communicate effectively

• What ‘effectively’ means depends on the context
Evaluating Networks

- **Bandwidth**: Amount of data that can be moved per unit of time
- **Latency**: How long it takes a given piece of the message to traverse the network
- Congestion: The effect on bandwidth and latency of the utilisation of the network by other processors
- Fault tolerance
- Area
- Power dissipation
Bandwidth vs. Latency

• Definitely not the same thing

• A truck carrying one million 256Gbyte flash memory cards to London
  – Latency = 4 hours (14,400 secs)
  – Bandwidth = ~128Tbit/sec (128 * 10^{12} bit/sec)

• A broadband internet connection
  – Latency = 100 microsec (10^{-4} sec)
  – Bandwidth = 10Mbit/sec (10^7 bit/sec)
Important features of a NoC

• Topology
  – How cores and networking elements are organised

• Routing
  – How traffic moves through the topology

• Switching
  – How traffic moves from one component to the next
Topology
Common wire interconnection – broadcast medium
Only single usage at any point in time
Controlled by clock – divided into time slots
Sender must ‘grab’ a slot (via arbitration) to transmit
Often ‘split transaction’
  – E.g send memory address in one slot
  – Data returned by memory in later slot
  – Intervening slots free for use by others
Crossbar

- E.g. to connect N inputs to N outputs

- Can achieve ‘any to any’ (disjoint) in parallel
Tree

- Variable bandwidth
  - (Switched vs Hubs)
  - (Depth of the Tree)

- Variable Latency

- Reliability?
Fat Tree
Ring

- Simple but
  - Low bandwidth
  - Variable latency

- Cell Processor (PS3)
Mesh / Grid

- Reasonable bandwidth
- Variable Latency
- Convenient for very large systems physical layout
Routing
Length of Routes

- **Minimal routing**
  - Selects always the shortest path to a destination
    - Packets always move closer to their destination
  - Packets are more likely to be blocked

- **Non-minimal routing**
  - Packets can be diverted
    - To avoid blocking, keeping the traffic moving
    - To run away from congested areas
  - Risk of livelock
Oblivious routing

• Unaware of network state
  – Deterministic routing
    • Fixed path, e.g. XY routing
  – Non-deterministic routing
    • More complex strategies

• Pros
  – Simpler router
  – Deadlock-free oblivious routing

• Con
  – Prone to contention
Adaptive Routing

- Aware of network state
  - Packets adapt to avoid contention

- Pros
  - Higher performance

- Cons
  - Router instrumentation is required
    - More complex i.e. more area and power
  - Deadlock prone
    - Even more hardware

- Barely used in NoCs
Switching
Packet switching

- Data is split into small packets and these into flits
- Some extra info is added to the packets to identify the data and to perform routing
- Allows time-multiplexing of network resources
  - Typically better performance, specially for short messages
- Several packet switching strategies
  - Store and forward, cut-through, wormhole
Store and Forward Switching

• A packet is not forwarded until all its phits arrive to each intermediate node

• Pros
  – *On-the-fly* failure detection

• Cons
  – Low performance
    • Latency: distance $\times$ #phits
  – Large buffering required

• Long transmission times
  – Internet
Cut-through / Wormhole Switching

• A packet can be forwarded as soon as the head arrives to an intermediate node

• Pros
  – Better performance
    • Latency: distance + #phits

• Cons
  – Fault detection only possible at the destination
    • Less hardware
Miscelanea
Amdahl’s Law

\[
\text{Speed up} = \frac{S + P}{S + (P/N)}
\]

- **S** = Fraction of the code which is serial
- **P** = Fraction of the code which can be parallel
- **S + P = 1**
- **N** = Number of processor
Processors Design Objectives

- Execution speed
  - High performance computing (supercomputers)
- Executing the most tasks per unit of time
  - High throughput computing (clusters and datacentres)
- Reliability
  - e-commerce, e-banking, industrial control systems
- Reduce power
  - Smartphones, tablets, laptops
- Real-time systems
  - Control systems that need to respond to events within a given time constraint
Flynn's taxonomy of processors

- **SISD**
  - Single Instruction Single Data
  - Uniprocessor

- **SIMD**
  - Single Instruction Multiple Data
  - Vector processor & Vector operations (MMX & SSE)
  - Agreeably GPUs

- **MISD**
  - Multiple Instructions Single Data
  - Systolic array (uncommon)

- **MIMD**
  - Multiple Instructions Multiple Data
  - Multi-cores (multiprocessors)

<table>
<thead>
<tr>
<th></th>
<th>Single Instruction</th>
<th>Multiple Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Data</td>
<td>SISD</td>
<td>MISD</td>
</tr>
<tr>
<td>Multiple Data</td>
<td>SIMD</td>
<td>MIMD</td>
</tr>
</tbody>
</table>
Classifying Processors

- **RISC**
  - Reduced instruction set
  - Small number of very fast simple instructions
  - Complex instructions are constructed from many smaller instructions

- **CISC**
  - Complex instruction set
  - Lots of instructions
  - Can be slow, but do a lot of work per instruction