THE UNIVERSITY of York

High Performance Computing - Parallel Computers and Networks

Prof Matt Probert

http://www-users.york.ac.uk/~mijp1

Overview

- Parallel on a chip?
- Shared vs. distributed memory
- Latency & bandwidth
- Topology
- Beowulf

Parallel Execution

- Many modern CPUs have some intrinsically parallel features on a single chip.
- Superscalar
  - Multiple functional units within a single CPU
  - Most modern CPUs have at least two floating-point functional units.
  - Hence *theoretical peak* speed of Viking (on 1 core of Intel Xeon Gold 6138 CPU)
    3.7 GHz x 2 FP units x 2 threads = 14.8 GFLOPS

SMT Parallel Execution

- Many chips support simultaneous multi-threading (SMT) – called “hyperthreading” by Intel
  - Each thread shows up as a separate CPU to the operating system.
  - Each thread has its own registers, but threads share functional units – see hardware lecture
  - Benefit (or lack thereof) is dependent on the code – see lectures on parallel compilers and OpenMP.

Multi-Core Parallel Execution

- Many chips are now “multi-core”
  - 2/4/6/... distinct CPUs on a chip, each with its own registers and functional units.
  - Standard in modern AMD and Intel chips.
  - Exynos 8890 is 8-core (as in my smartphone) [Apple iPhone 11 has A13 Bionic with 2 high-performance cores + 4 energy-efficient cores]
  - True parallel processing on a single chip.

SIMD

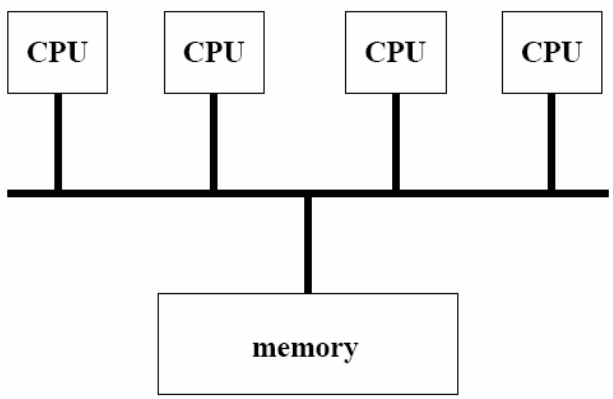
- A useful terminology is SIMD (Single Instruction Multiple Data).
  - Now present in commodity CPUs
  - Driven by multimedia and games – not HPC
  - AMD first with “3DNow!” in K6-2
  - Intel followed with MMX (integer only - Pentium)
  - then came SSE (Streaming SIMD Extensions) (single precision floating point – Pentium 3)
  - and SSE2 (double precision floating point – Pentium 4)
  - and SSE3 (multiple ops per register – later P4s)
  - and SSE4 (string processing - Core2)
  - …
  - And AVX (‘Vector’ registers increased from 128 to 256 bits) and then AVX512 …

Multi-Tasking is Not Parallel

- A single CPU (single core) can only perform one task at a time but can give the illusion of concurrency
  - With *multiprogramming*, the current task keeps running until it performs an operation that requires waiting for an external event (e.g. reading from a disk). Uncommon these days
  - With *real-time* systems, some waiting time-critical tasks are guaranteed to be given the CPU on an external interrupt.
  - With *co-operative time-sharing* (e.g. early versions of MacOS and MS Windows), the running task voluntarily relinquishes CPU – hence vulnerable to bugs and badly-written software.
  - With *pre-emptive time-sharing* (e.g. UNIX, Win9x, etc), the running task knows nothing about multi-tasking, and it is the O/S which suspends the task after given timeslice (typically 1-100 ms). Needs hardware support from the CPU.
- Nowadays, *multitasking* usually means *time-sharing*
- *CPU is not just running your code: OS + other users*

Simple Multi-Processor Machine

- What if put 2 or more distinct CPUs on a single motherboard bus and share other resources? Would this be a useful design?



- Known as SMP (symmetric multi-processor, or shared-memory processor)
- All CPUs equivalent (hence symmetric architecture) and memory is shared by all CPUs using a single system bus

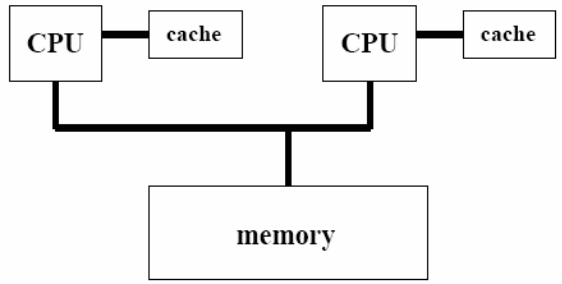
SMP

- Common approach to make a dual-processor machines.
- Cheap and simple to build
- Crazy?
  - Memory is the bottleneck, not CPU speed
  - Typical modern single-CPU computer spends 75% or more of its time waiting for memory, so sharing one memory bank with multiple CPUs is daft?

SMP Advantages

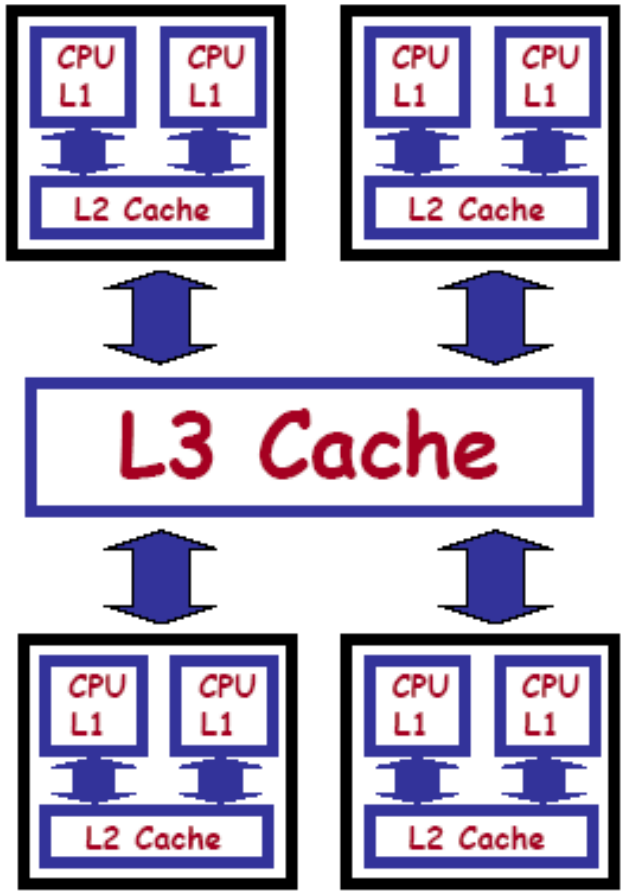
- Easy to program as all processors see same memory, hence easy to exchange data!
  - If one CPU writes result to main memory then all other CPUs can see the result easily
- So only have to divide up task so that each CPU can operate simultaneously – which can be done by an auto-parallelising compiler!
- Or can get better performance by human intervention, telling compiler what to do where, using OpenMP directives. See later lectures.
- But …

SMP Disadvantages



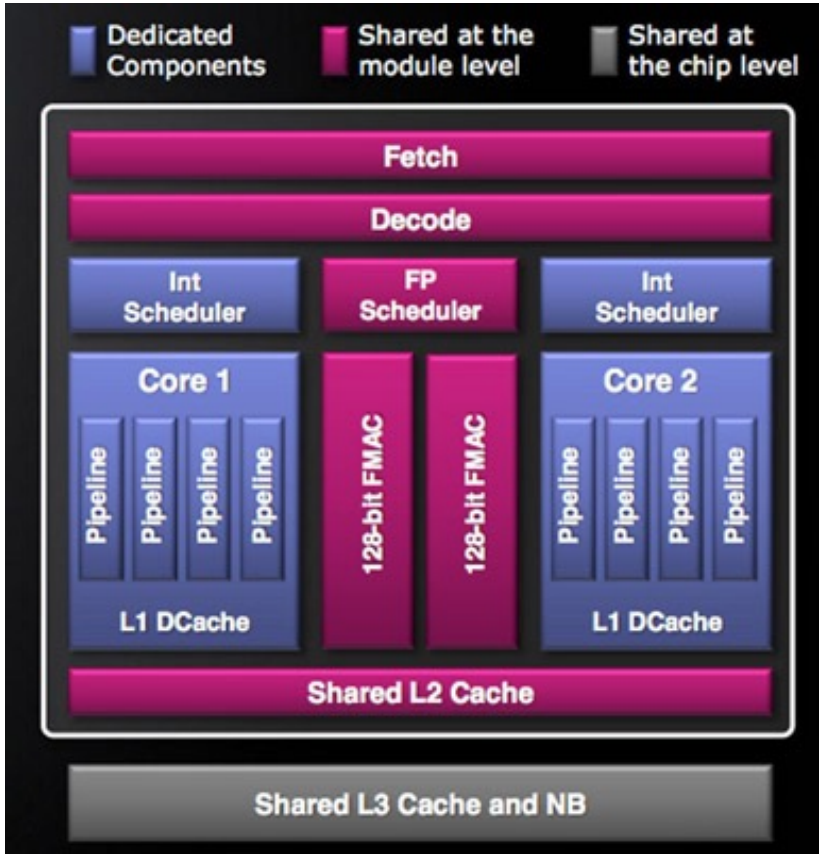
- Most CPUs don't talk directly to main memory but use caches instead
- Consider: CPU A reads a value from memory, operates on it, writes it back into cache. Then CPU B needs same memory location – does it get it from memory or CPU A's cache? Need to maintain *cache coherency*!
  - Can either abandon all caches or have much more complex cache protocols e.g. *snoopy cache*
- Net result is less than 2x speedup for adding second processor, and a scenario that does not scale well – need to broadcast coherency traffic to cache controllers

IBM Power4 Example (2001)



- L1 64 KB Instruction + 32 KB Data
  - 4-5 cycles latency
- L2 1440KB Unified – Shared by 2 cores
  - 14-20 cycles latency
- L3 128MB per 8 core module (‘MCM’)
  - 100 cycles latency
- Main Memory – 8 GB per MCM
  - 350-400 cycles latency
- 4 MCMs per compute node (32 cores)

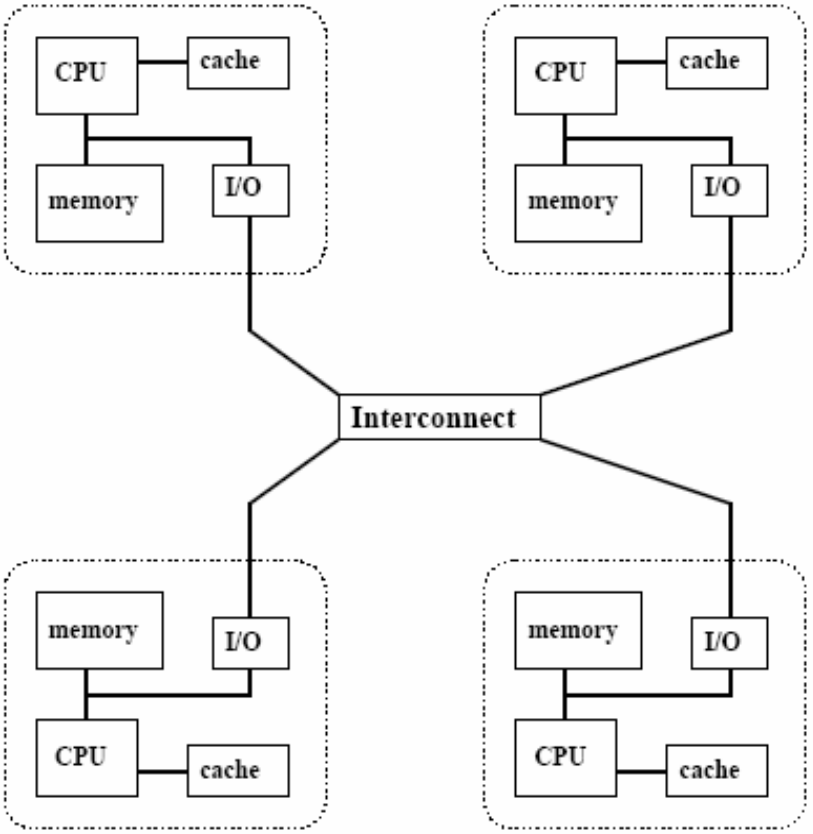
AMD Bulldozer (2011)



- Designed for HPC and servers
- Each module contains 2 cores
  - up to 8 modules into 1 chip
- L1 4-way 16 KB Instruction per core + 2-way 64 KB Data per module
  - 3 cycles latency
- L2 2MB – shared within a module
  - 17 cycles latency
- L3 16MB shared between all modules
  - 24 cycles latency
- Multi-threading + hyper-transport
- 4 arithmetic ops per clock per core

Distributed Memory

- Why not give each CPU its own memory and cache?
  - Adding more processors then adds more local (not global) memory and hence total memory bandwidth scales with number of processors!
  - Known as distributed memory computer, or MPP (massively parallel computer) as it can scale to 1000s of CPUs with no cache coherency problems!
  - Can use multiple topologies



Making a Parallel Computer

- How do we make a parallel computer - do we just need network cards + cables?
- Yes and No – you can build a simple PC cluster using this approach but it will not be any faster than a single PC
- Need to add some software magic – either to distribute different tasks to different computers (e.g. SETI@home) or to divide up a given task so that different computers work together

Latency and Bandwidth

- Which is more important? Latency or Bandwidth?
  - Most computer users focus on bandwidth – simple folk just quote bandwidth, e.g. 8 Mb broadband or Gigabit Ethernet – but latency is also important
  - *Bandwidth* measures how much information can be transferred over a connection in a given period of time. *Latency* is how long it takes for a response to return from a request, i.e. to send a zero-size message
- If you have a low-latency network then it is simple to increase bandwidth by putting links in parallel
- But if you have high-latency then you are stuck with it! No amount of data compression or doubling-up of connections is going to help.

Latency and Bandwidth Example

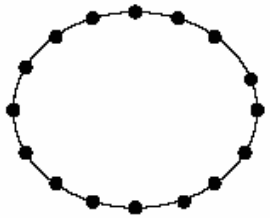
- Typical 10 Gbit Ethernet latency ~ 0.3ms
- Typical ADSL latency ~ 10ms
- Typical dial-up modem latency ~ 100ms
- How long would it take to send 10 characters over 56 kbs modem (10 chars = 80 bits)?
  - Naïve is 80 bits/57344 bits/sec = 1.4 ms
  - Actual is 101.4 ms due to 100 ms latency
- But sending 100 kB would take 8*100/56 + 0.1 = 14.4 sec so the 0.1 sec latency is negligible
- May be slower due to contention, software protocol overheads, etc

Throughput

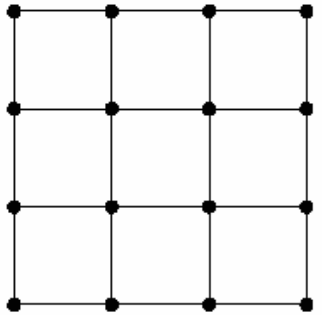
- So depending on your pattern of usage, latency *may* be much more important than bandwidth:
  - What matters most is the total time to get the job done, i.e. the *throughput*. This depends on the size of the message …
  - If **size < (latency * bandwidth)** then latency will dominate
  - Need large message size to get best performance
  - Satellite comms has high bandwidth but poor latency due to length of trip (typically 1-2 sec) hence OK for broadcast TV but not for speech or Internet gaming …

Topology

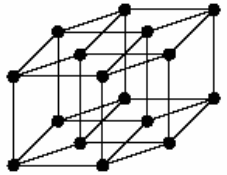
- Different ways of connecting nodes in a network (topologies) have different characteristics and costs
- Ideally, network topology should not be apparent to any users …
- Earlier Ethernets used a ring – very vulnerable if a user unplugged their connection!



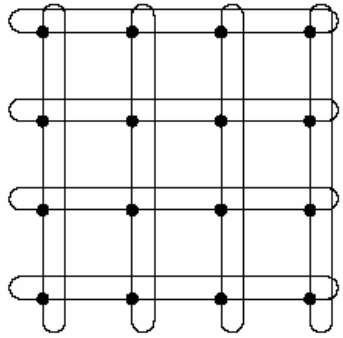
Ring (1D torus)



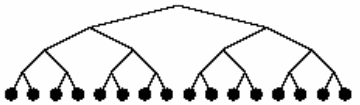
2D mesh



Hypercube



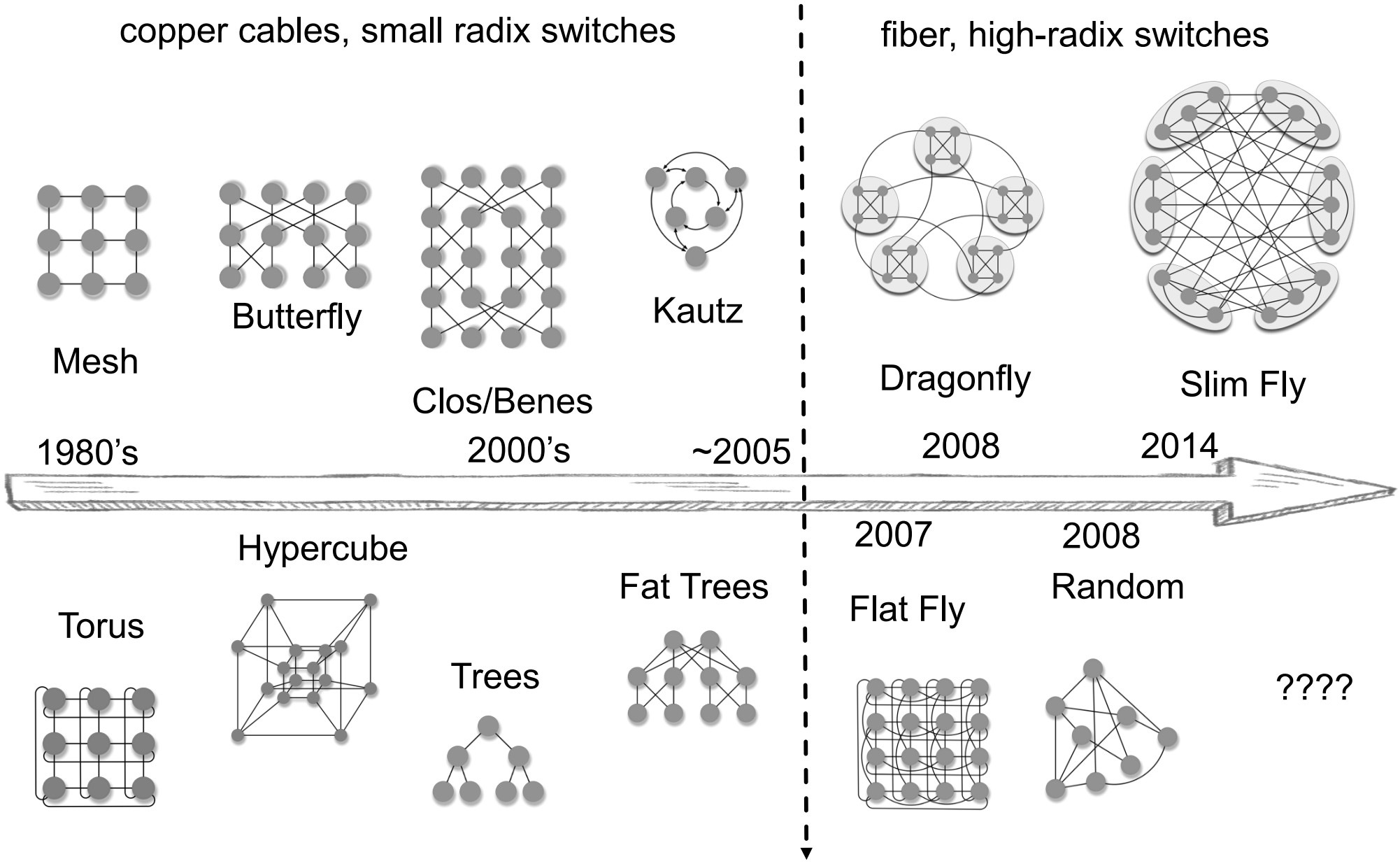
2D torus



Tree (log 2)

Fat Tree (log 4)

Evolution of topology



Effect of Topology

- Bisectional bandwidth is the bandwidth between two halves of network – want it big
- Diameter is the max. number of hops from one node to another – want it small
- Slim Fly uses less cables & routers than Dragonfly so cheaper & less power

	Bi-BW	*Diameter*
Ring	2	N/2
2D Grid	$\sqrt{N}$	$2\sqrt{N}$
2D Torus	$2\sqrt{N}$	$\sqrt{N}$
Hypercube	N/2	$\log_2 N$
Tree	2	$2\log_2 N$
Fat Tree	N/2	$2\log_2 N$
Dragonfly	N/4	3 - 5
Slim Fly	N/4	2 - 4

(*fly only for large N as in ARCHER-2)

MPP Programming

- Programming an MPP machine is more challenging
  - Programmer must “think parallel”
  - Each processor is working on same code but with its own independent memory and own local values of all variables
  - Any inter-processor communication must be coded explicitly
  - Early MPP days – each vendor had its own way of doing things with consequent loss of code portability
  - First portable approach was PVM (Parallel Virtual Machine, 1991), now superseded by MPI (Message-Passing Interface, 1994) – standard library (C and FORTRAN bindings)
  - We will do MPI programming later in the course.

Coding Example

- Imagine doing a dot-product between two vectors, A and B, already set up:

SMP version

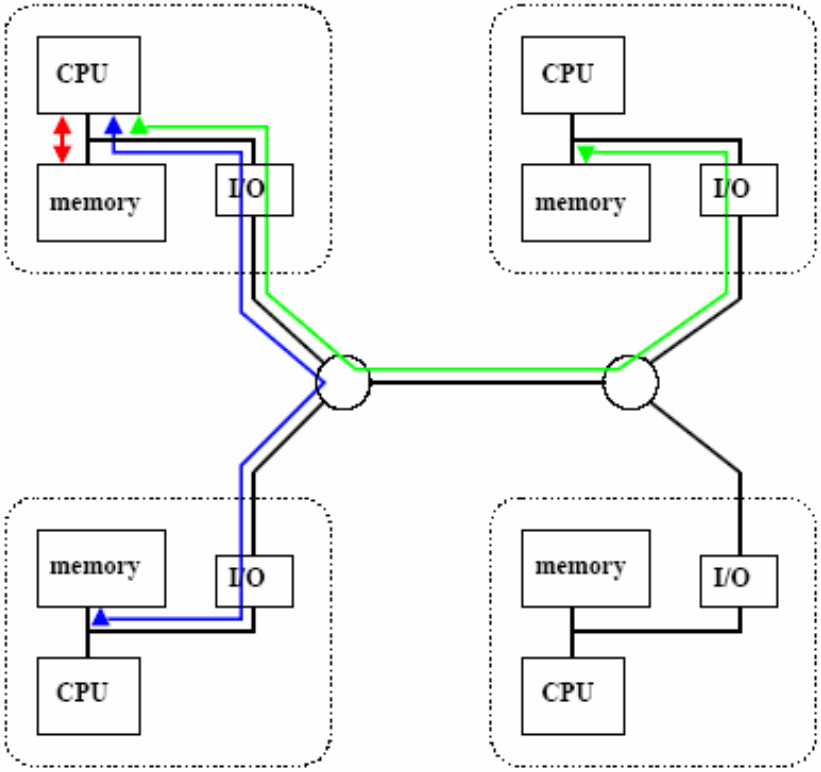
```
Nmax=10000
!Hope compiler can optimise
!this loop OK
t=0.0
do i=1,Nmax
   t=t+A(i)*B(i)
end do
```

MPP version

```
Nmax=10000/ncpu
!calculate partial sum
my_t=0.0
StartNum = myRank*Nmax+1
EndNum   = StartNum+Nmax-1
do i=StartNum,EndNum
   my_t=my_t+A(i)*B(i)
end do
!condense all results
Call MPI_AllReduce(my_t,t,…)
```

Advanced SMP Designs

- What if could make a machine *look* like it was shared memory but was actually distributed?
  - Easier to program?
  - Effect on performance?
- NUMA – non-uniform memory access
- cc-NUMA – cache-coherent NUMA – developed by SGI
  - Abandoned early 2000s but revived in 2009 with Altix UV up to 2048 cores &18 TFLOP/s
  - Bought by HPE in 2016 and rebranded as FLEX



4 nodes gives 3 different memory access times – on-node, nearest neighbour and next-nearest neighbour

Modern Supercomputer Designs

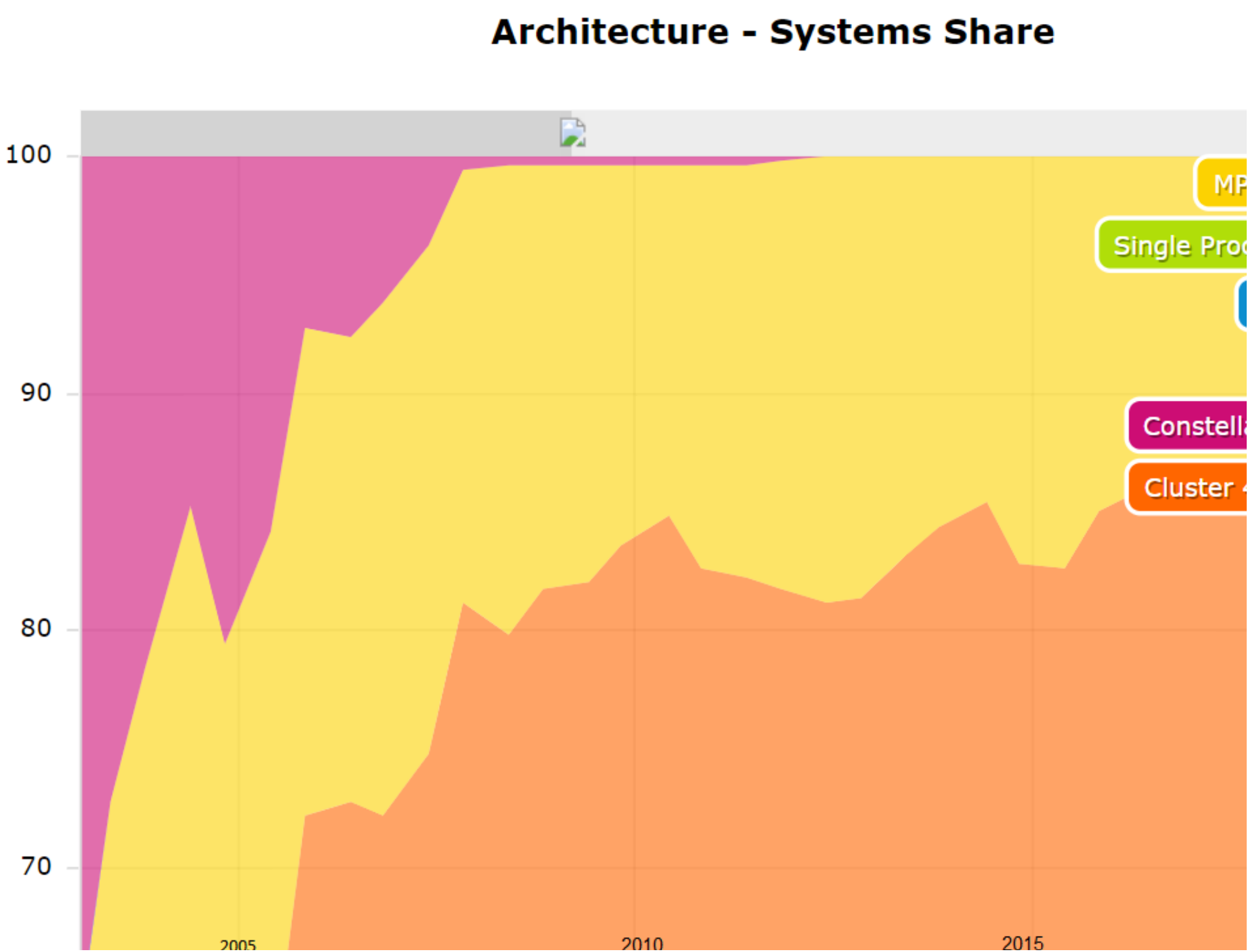
- Hybrid MPP/SMP – “fat nodes”
- Multi-core processors
- Multiple processors/board = SMP = fat node
- Multiple boards/rack = MPP
- Multiple racks = clustered (constellation) MPP
  - Another hierarchy of memory/communications bottlenecks!
- Good networking essential!
  - HPCx phase I crippled by poor switch between racks – had latency ~ 400 μs!
  - Current ARCHER2 is bandwidth limited between nodes with only 2 bi-directional NICs for 128 cores!
- Challenge to write code for 1000s of CPUs

Beowulf

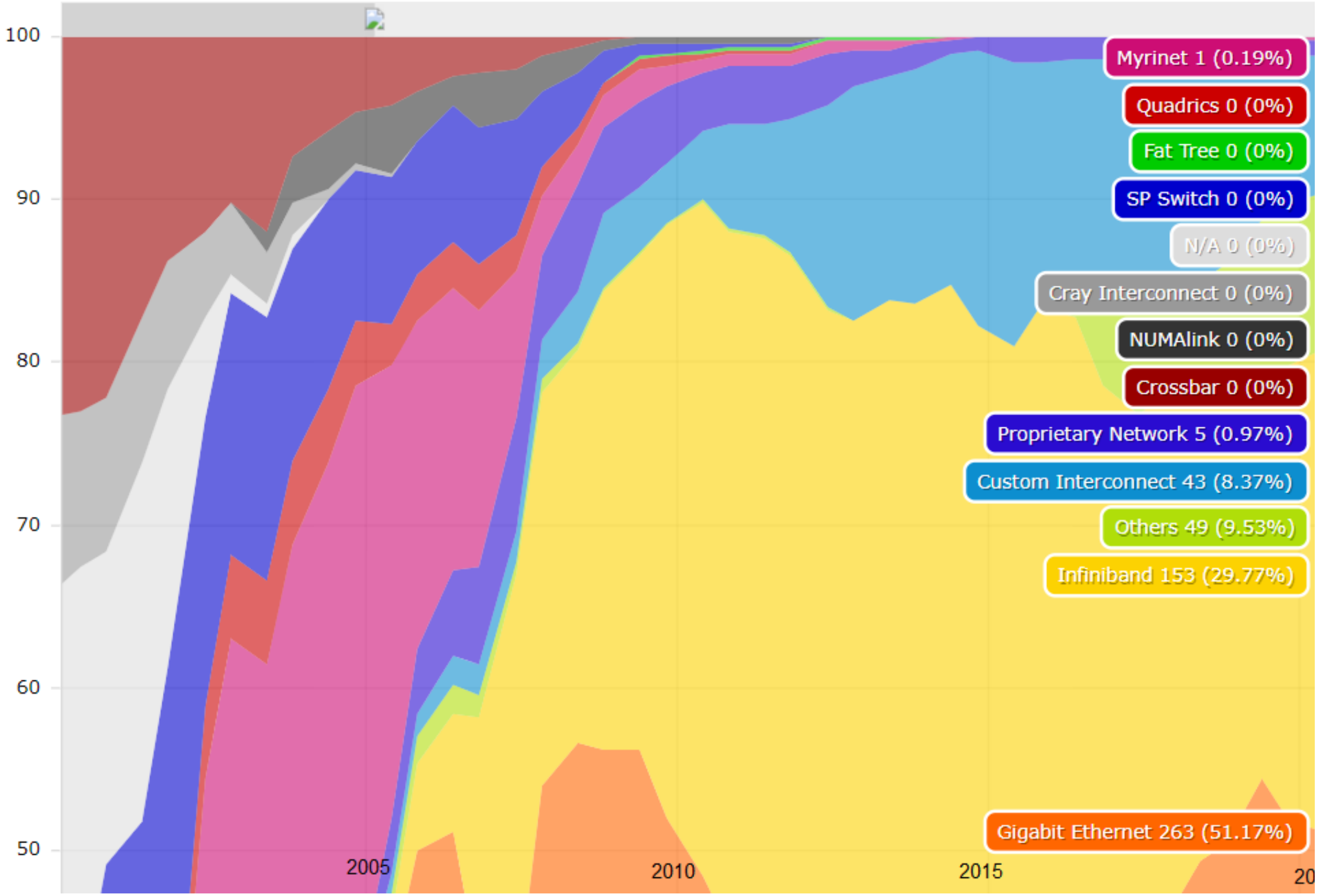
- How about building a cheap supercomputer from commodity components?
- “Beowulf” design
  - Developed by Donald Becker in 1993 at NASA
  - Join together lots of PCs with network and MPI to make a distributed memory machine
  - Developments in high-speed, low-latency networking hardware (e.g. Mellanox or InfiniPath) plus open-source software makes this an attractive proposition

UoY Physics

- In Dec 2001 the department bought Erik for £120k
  - 64 Intel P4 Xeons, peak ~ 0.1 TFLOP/s
  - 32 nodes with 2 GB RDRAM and 80 GB disk per node
  - Use Myrinet-2000 interconnect
    - fibre optic comms with latency ~ 6 μs and BW= 480 MB/s
- In June 2008 we bought Edred for £100k
  - 64 AMD Barcelona (quad-core), peak ~ 1.0 TFLOP/s
  - 32 nodes with 16 GB DDR2 and 500 GB disk per node
  - Use InfiniPath interconnect
    - fibre optic comms with latency ~ 1 μs and BW= 1800 MB/s
- In June 2010 we bought Jorvik for £5k *for teaching*
  - 5 nodes AMD Phenom II hex-core, peak ~ 0.17 TFLOP/s
  - Gigabit Ethernet, 4 GB DDR3 and 200 GB disk per node
- In Aug 2015 we bought Ebor for £40k *for teaching*
  - 5 nodes Intel E5-2630, 8 core, peak ~ 12 TFLOP/s
  - Intel Infiniband, 32 GB DDR4 and 80 GB SSD per node


Architecture - Systems Share
100
90
80
70
2005
2010
2015
MP
Single Proc
Constell
Cluster

Interconnect Family - Systems Share



Further Reading

- Chapter 4 of “Introduction to High Performance Computing for Scientists and Engineers”, Georg Hager and Gerhard Wellein, CRC (2011).
- “Computer Architecture – A Quantitative Approach (6^{th} edition)", John L Hennessy and David A Patterson, Morgan Kaufmann Pub. Inc. (2017).
- http://www.ibm.com
- http://www.hpe.com
- Top500 supercomputer data http://www.top500.org