Lecture 3 – Overview of Distributed and Parallel Computing Systems

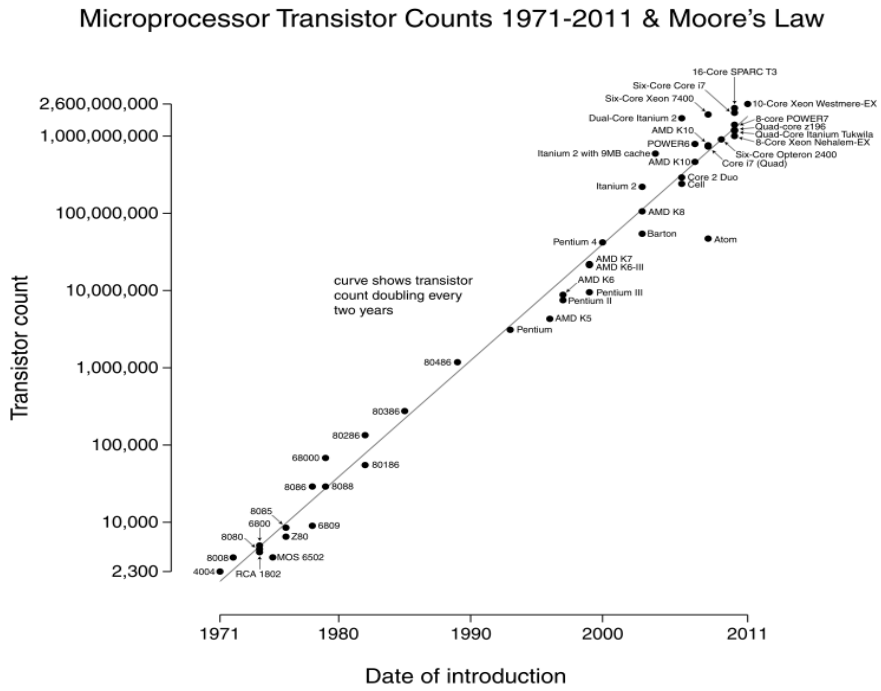
Professor Richard O. Sinnott
Director, eResearch
University of Melbourne
16th March 2016
rsinnott@unimelb.edu.au

Some things…

- Next weeks lectures/workshop
  - (it starts getting practical!!!)
- Assignment 1 – handed out next week
  - Attending next weeks lecture essential to do assignment
- Assignment 2 – teams reminder
  - Organise or be organised
    - Googledocs to be established (details will be on LMS)
- UniMelb data centre visit
  - 20 Minutes per group
  - 20 People per group
    - Planned for end March/early April

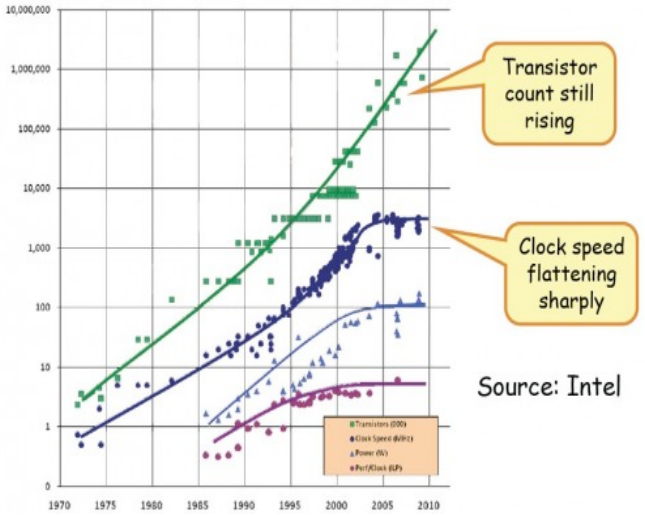
Compute Scaling

- Vertical Computational Scaling
  - Have faster processors
    - Switch your *n* GHz CPU for a *2n* GHz one = 2x faster
      - Easy to do, but costs more
  - Limits of fundamental physics/matter (nanoCMOS)
    - 3.6Ghz (Intel i7-4960X)



Compute Scaling

- Horizontal Computational Scaling
  - Have more processors
    - Easy to add more; cost increase not so great
    - But…
      - harder to design, develop, test, debug, deploy, manage, understand, …



T

    - Note that for many/most folks
      - HTC far more important than HPC
      - (I don't do HPC, but I've done a lot of HTC)

Add More = ???

- Single machine multiple cores
  - Typical PC these days
- Loosely coupled cluster of machines
  - Pooling/sharing of resources
    - Dedicated vs available only when not in use by others
    - Web services, ... Condor, Seti@home, Boinc
- Tightly coupled cluster of machines
  - Typical HPC/HTC set-up (SPARTAN, VLSCI, …)
    - Many servers in same rack/server room (often with fast message passing interconnects)
- Widely distributed clusters of machines
  - UK NGS, EGEE, … distributed systems more generally
- Hybrid combinations of the above
  - Leads to many challenges with distributed systems
    - Shared state (or lack thereof)
    - Message passing paradigms – dangers of delayed/lost messages

Add More…limitations

- Add more processes…?
  - If *n* processors (cores) are thrown at a problem how much faster will it go?
- Some terminology:
  - T(1) = time for serial computation
  - T(N) = time for N parallel computations
  - S(N) = speed up

$$S(N) = \frac{T(1)}{T(N)}$$

  - Proportion of speed up depends on parts of program that can’t be parallelised

Amdahl's Law

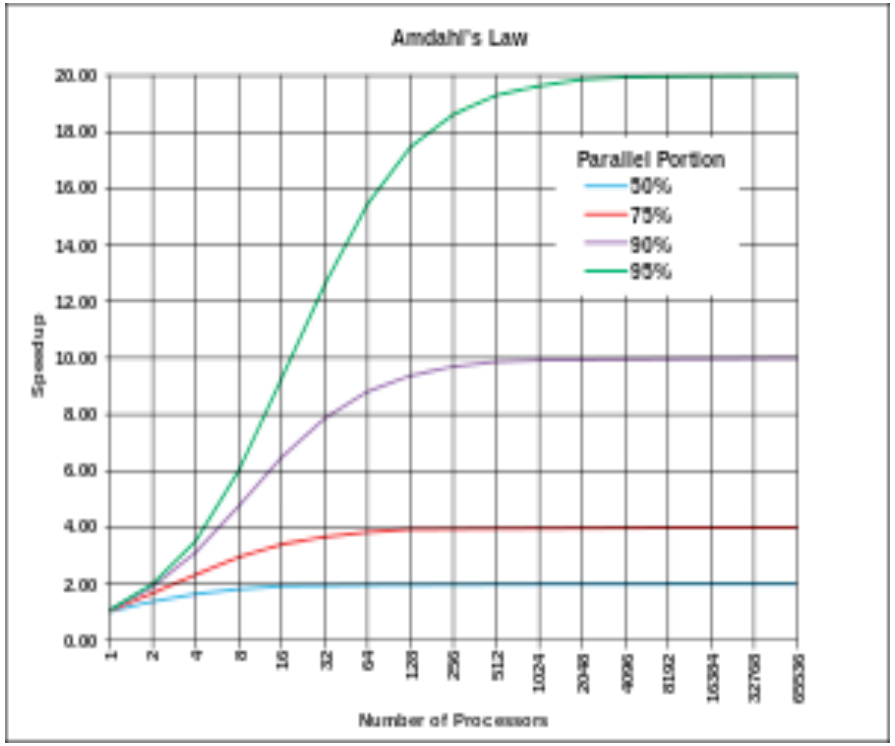
$$T(1) = \sigma + \pi \qquad T(N) = \sigma + \frac{\pi}{N}$$

$$S = \frac{T(1)}{T(N)} = \frac{\sigma + \pi}{\sigma + \pi/N} = \frac{1 + \pi/\sigma}{1 + (\pi/\sigma) \times (1/N)}$$

$$\pi/\sigma = \frac{1-\alpha}{\alpha}$$

α Fraction of running time sequential program spends on parallel parts

$$S = \frac{1 + (1-\alpha)/\alpha}{1 + (1-\alpha)/(N\alpha)} = \frac{1}{\alpha + (1-\alpha)/N} \approx \frac{1}{\alpha}$$



If 95% of the program can be parallelized, the theoretical maximum speedup using parallel computing would be 20×, no matter how many processors are used, i.e. if the non-parallelisable part takes 1 hour, then no matter how many cores you throw at it it won't complete in <1 hour.

Over Simplifications of Amdahl's Law

- Consider a program that executes a single loop, where all iterations can be computed independently, i.e. code can be parallelized. By splitting the loop into several parts, e.g. one loop iteration per processor, each processor now has to deal with loop overheads such as calculation of bounds, test for loop completion etc. This overhead is replicated as many times as there are processors. In effect, loop overhead acts as a further (serial) overhead in running the code. Also getting data to/from many processor overheads?

In a nutshell Amdahl's Law greatly simplifies the real world!

- It also assumes a fixed problem size – sometimes can't predict length of time required for jobs,
  - e.g. state space exploration or differential equations that don't solve…

Gustafson-Barsis's Law

- Gives the “scaled speed-up”

$$T(1) = \sigma + N\pi \quad \text{and} \quad T(N) = \sigma + \pi$$

$$S(N) = \frac{T(1)}{T(N)} = \frac{\sigma + N\pi}{\sigma + \pi} = \frac{\sigma}{\sigma + \pi} + \frac{N\pi}{\sigma + \pi}$$

π Fixed parallel time per process

α Fraction of running time sequential program spends on parallel parts

$$\pi/\sigma = \frac{1-\alpha}{\alpha}$$

$$S(N) = \alpha + N(1-\alpha) = N - \alpha(N-1)$$

Speed up S using N processes is given as a linear formula dependent on the number of processes and the fraction of time to run sequential parts. Gustafson's Law proposes that programmers tend to set the size of problems to use the available equipment to solve problems within a practical fixed time. Faster (more parallel) equipment available, larger problems can be solved in the same time.

Metaphors

The Wikipedia page for Gustafson's Law offers the following metaphor to contrast the two laws.

Amdahl's Law approximately suggests: Suppose a car is traveling between two cities 60 miles apart, and has already spent one hour traveling half the distance at 30 mph. No matter how fast you drive the last half, it is impossible to achieve 90 mph average before reaching the second city. Since it has already taken you 1 hour and you only have a distance of 60 miles total; going infinitely fast you would only achieve 60 mph.

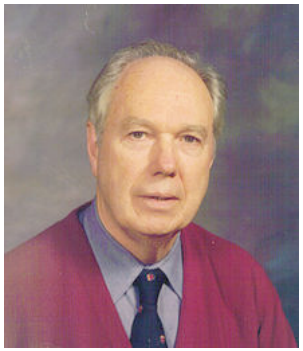
Gustafson-Barsis's Law approximately suggests: Suppose a car has already been traveling for some time at less than 90mph. Given enough time and distance to travel, the car's average speed can always eventually reach 90mph, no matter how long or how slowly it has already traveled. For example, if the car spent one hour at 30 mph, it could achieve this by driving at 120 mph for two additional hours, or at 150 mph for an hour.

Computer Architecture - 101

- At the simplest level a computer comprises
  - CPU for executing programs
    - ALU, FPU, …
    - Load/store unit
    - Registers (fast memory locations)
    - Program counter (address of instruction that is executing)
    - Memory interface
  - Memory that stores/executing programs and related data
  - I/O systems (keyboards, networks, …)
  - Permanent storage for read/writing data into out of memory
  - Of key importance (especially for HPC systems!) is the balance of all of these
    - Superfast CPUs starved of data
    - Bioinformatics jobs killing clusters!!!
  - There are many different ways to design/architect computers
    - different flavours suitable to different problems

Computer Architectures

- Flynn's Taxonomy
  - Single Instruction, Single Data stream (SISD)
  - Single Instruction, Multiple Data streams (SIMD)
  - Multiple Instruction, Single Data stream (MISD)
  - Multiple Instruction, Multiple Data streams (MIMD)

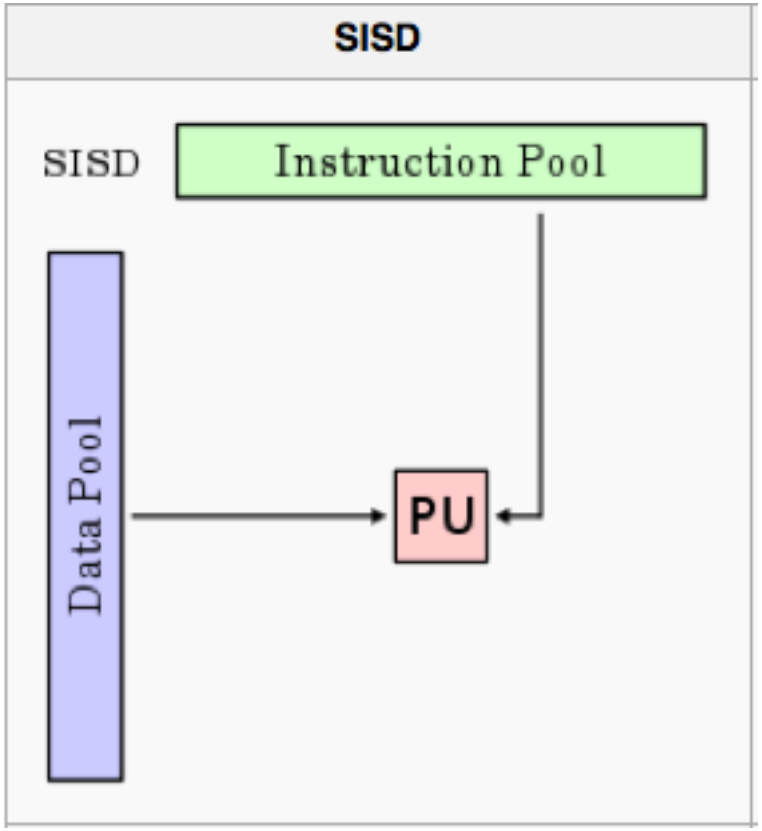


Flynn's taxonomy

	Single instruction	Multiple instruction
Single data	SISD	MISD
Multiple data	SIMD	MIMD

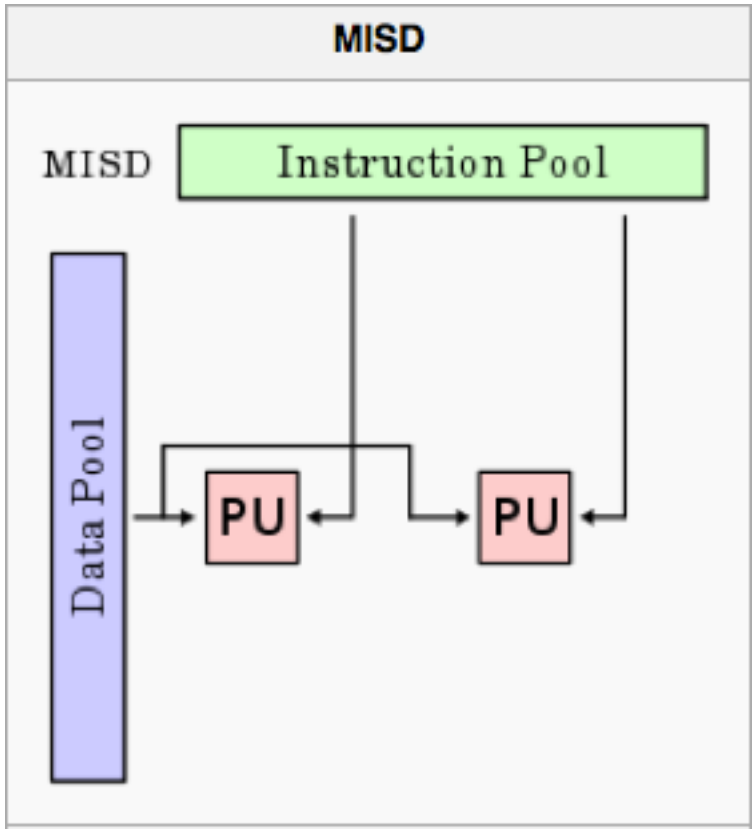
Flynn's::SISD Architectures

- Single Instruction, Single Data stream (SISD)
  - Sequential computer which exploits no parallelism in either the instruction or data streams
  - Single control unit (CU/CPU) fetches single Instruction Stream from memory. The CU/CPU then generates appropriate control signals to direct single processing element to operate on single Data Stream, i.e. one operation at a time.
  - Pretty much obsolete...!
    - Basic idea of von Neumann computer



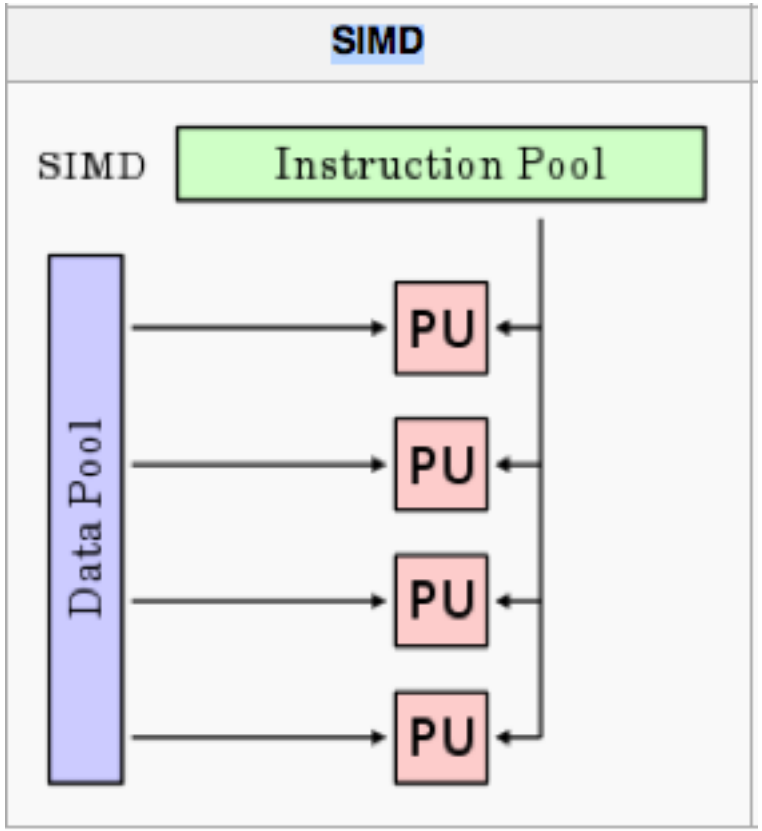
Flynn's::MISD Architectures

- Multiple Instruction, Single Data stream (MISD)
  - Parallel computing architecture where many functional units (PU/CPU) perform different operations on the same data
  - Examples include fault tolerant computer architectures, e.g. running multiple error checking processes on same data stream
    - (Not very common though!)



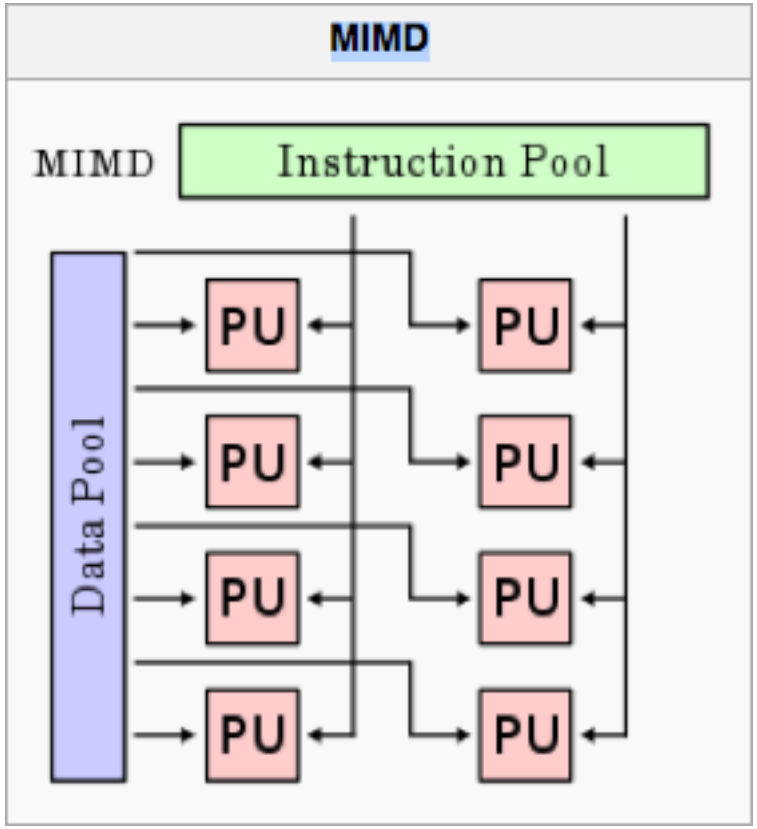
Flynn's::SIMD Architectures

- Single Instruction, Multiple Data stream (SIMD)
  - multiple processing elements that perform the same operation on multiple data points simultaneously
  - focus is on data level parallelism, i.e. many parallel computations, but only a single process (instruction) at a given moment
  - Many modern computers use SIMD instructions, e.g. to improve performance of multimedia use such as for image processing



Flynn's::MIMD Architectures

- Multiple Instruction, Multiple Data stream (SIMD)
  - number of processors that function asynchronously and independently
  - at any time, different processors may be executing different instructions on different pieces of data
  - machines can be shared memory or distributed memory categories
    - depends on how MIMD processors access memory
  - Most systems these days operate on MIMD
    - HPC …



Approaches for Parallelism

- Where and how
  - Explicit vs Implicit parallelism
  - Hardware
  - Operating System
  - Software/Applications
  - Some or all of these

Explicit vs Implicit Parallelisation

- Implicit Parallelism
  - Supported by parallel languages and parallelizing compilers that take care of identifying parallelism, the scheduling of calculations and the placement of data
    - Pretty hard to do (more later!)
- Explicit Parallelism
  - In this approach, the programmer is responsible for most of the parallelization effort such as task decomposition, mapping tasks to processors, inter-process communications
  - This approach assumes user is best judge of how parallelism can be exploited for a particular application
    - Typically non-trivial to achieve!
  - Consider SPARTAN HPC cluster assignment

    Slurm script (#SBATCH --ntasks=2 --cpus-per-task=4 vs

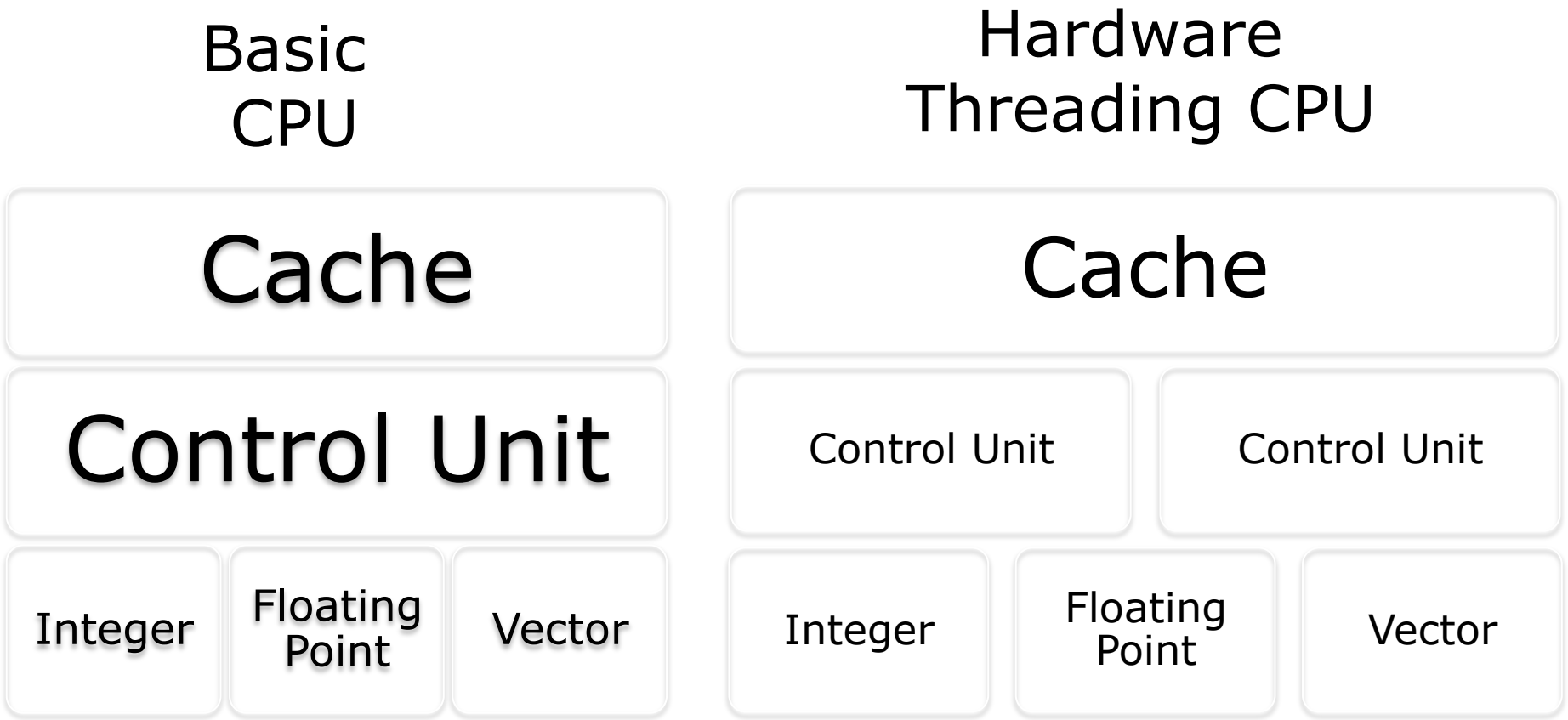
    #SBATCH –ntasks=1 --cpus-per-task=8)

    – (More next week from Lev!)

Approaches for Parallelism

- Where and how
  - Explicit vs Implicit parallelism
  - Hardware
  - Operating System
  - Software/Applications
  - Some or all of these

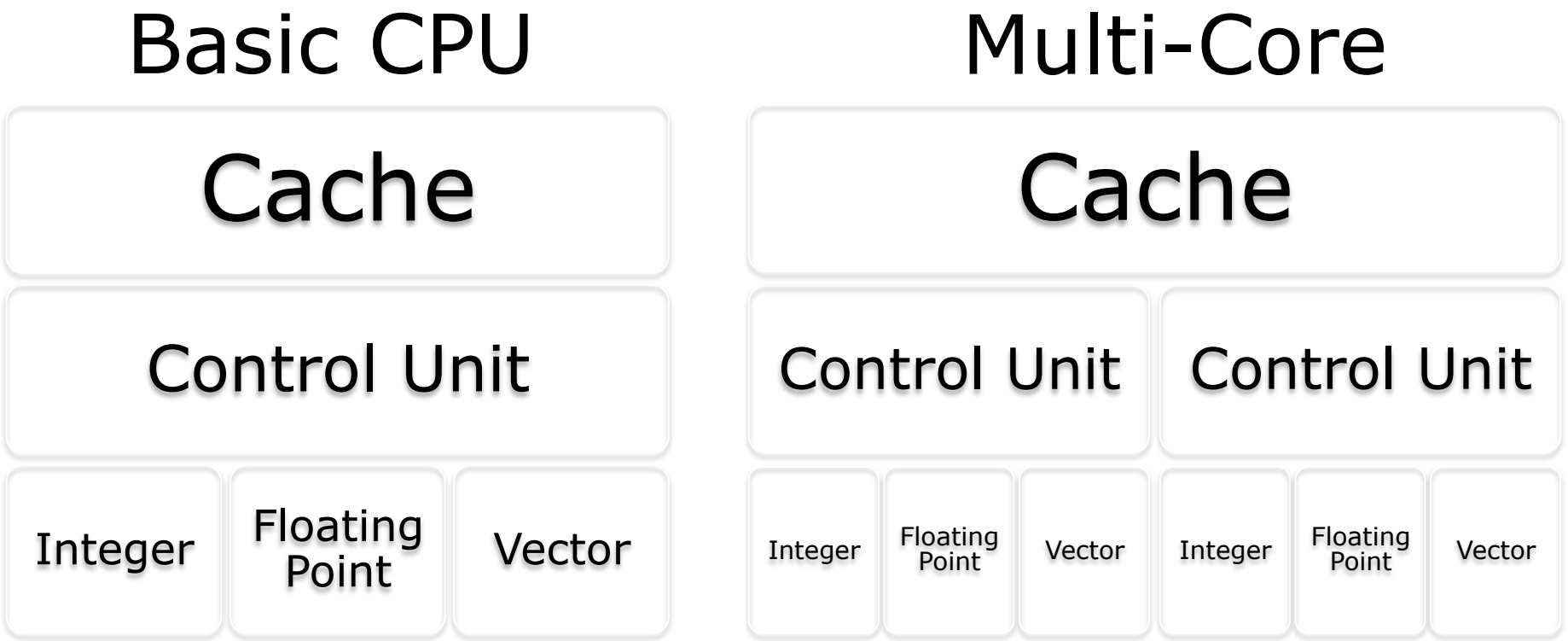
(Heavily Simplified) Hardware Parallelisation



Cache – much faster than reading/writing to main memory; instruction cache, data cache (multi-level) and translation lookaside buffer used for virtual-physical address translation (more later on Cloud).

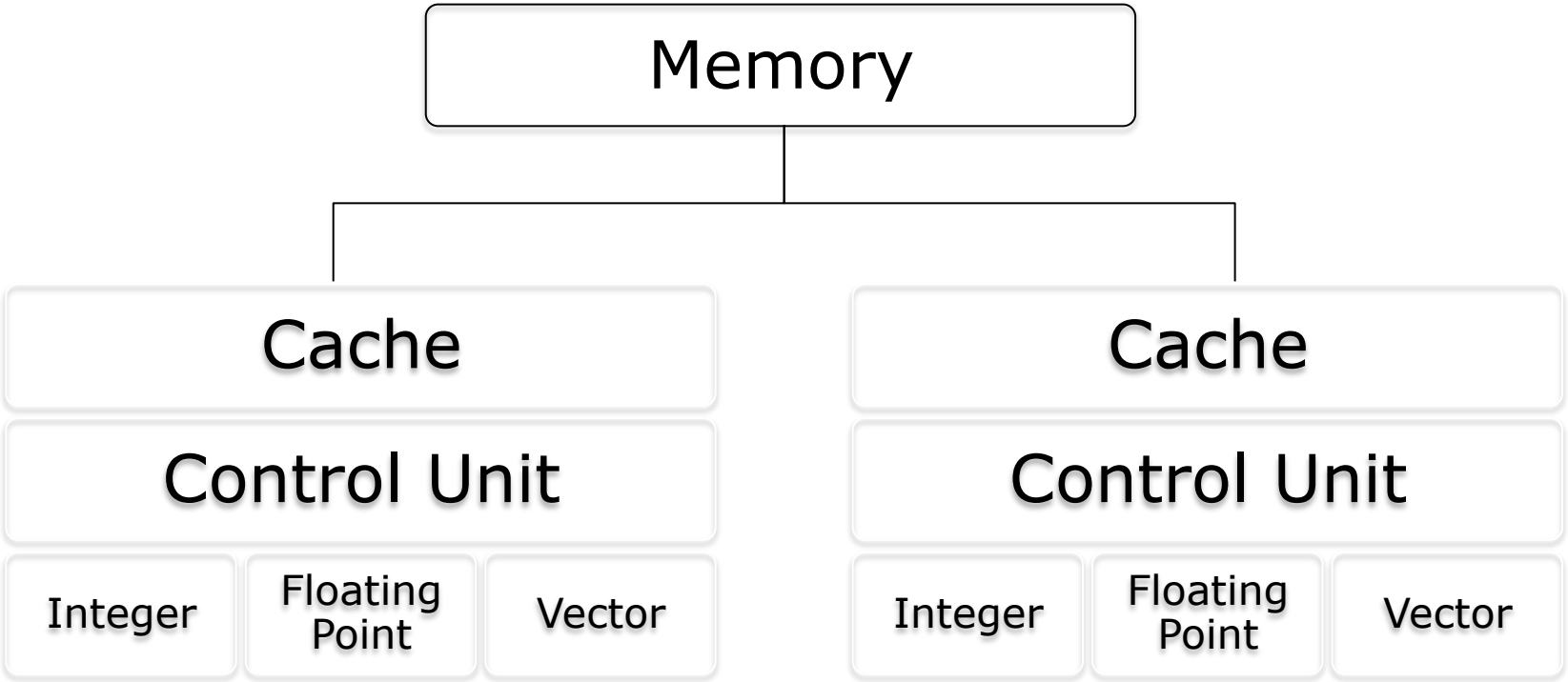
Parallelisation by adding extra CPU to allow more instructions to be processed per cycle. Usually shares arithmetic units. Heavy use of one type of computation can tie up all the available units of the CPU preventing other threads from using them.

(Heavily Simplified) Hardware Parallelisation



Multiple cores that can process data and (*in principle!!!*) perform computational tasks in parallel. Typically share same cache, but issue of cache read/write performance and cache coherence. Possibility of cache stalls (CPU not doing anything whilst waiting for caching); many chips have mixture (L1 cache on single cores; L2 cache on pairs of cores; L3 cache shared by all cores); typical to have different cache speeds and cache sizes (higher hit rates but potentially higher latency).

Heavily Simplified (Symmetric Multiprocessing (SMP))



Two (or more) identical processors connected to a single, shared main memory, with full access to all I/O devices, controlled by a single OS instance that treats all processors equally. Each processor executes different programs and works on different data but with capability of sharing common resources (memory, I/O device, …). Processors can be connected in a variety of ways: buses, crossbar switches, meshes. More complex to program since need to program both for CPU and inter-processor communication (bus).

Non-Uniform Memory Access (NUMA)

Non-uniform memory access (**NUMA**) provides speed-up by allowing a processor to access its own local memory faster than non-local memory. Improved performance as long as data are localized to specific processes/processors. Key is allocating memory/processors in NUMA friendly ways, e.g. to avoid scheduling/locking and (expensive) inter-processor communication. Approaches such as ccNUMA with range of cache coherency protocols/ products.

```
X=X+1;
Y=1;
If Y=0 print X
```

```
LOAD   R03, %A10 – load X into register
ADD    R03, %A10 – add 1 to value of R03 (=X)
STORE  R03, %A10 – store result
```

Approaches for Parallelism

- Where and how
  - Explicit vs Implicit parallelism
  - Hardware
  - Operating System
  - Software/Applications
  - Some or all of these

Operating System Parallelism Approaches

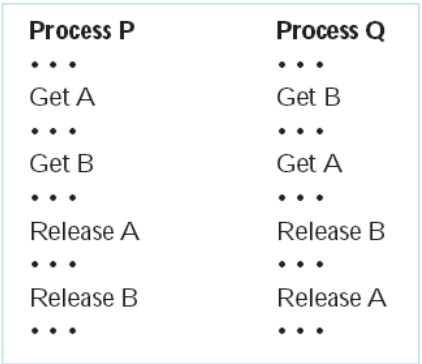
- Most modern multi-core operating systems support different “forms” of parallelisation
  - parallel vs interleaved semantics
    - A || B vs A ||| B
- Compute parallelism
  - Processes
    - Used to realise tasks, structure activities, …
  - Threads
    - Native threads
      - Fork, Spawn, Join
    - Green threads
      - Scheduled by a virtual machine instead of natively by the OS
- Data parallelism
  - Caching (cache coherency)

Approaches for Parallelism

- Where and how
  - Explicit vs Implicit parallelism
  - Hardware
  - Operating System
  - Software/Applications
  - Some or all of these

Software Parallelism Approaches

- Many (most!) languages now support a range of parallelisation/concurrency features
  - Threads, thread pools, locks, semaphores, …
- Many languages developed specifically for parallel/concurrent systems
  - http://en.wikipedia.org/wiki/Concurrent_computing
- Key issues that need to be tackled
  - Deadlock – processes involved constantly waiting for each other
  - Livelock – processes involved in livelock constantly change with regard to one another, but none are progressing



Message Passing Interface

- Widely adopted approach for message passing in parallel systems

- Mappings to major languages Fortran, C, C++, Python, Java
  - Standardised, widely adopted, portable, performant, …
  - Parallelisation = users problem

- Key MPI functions
  - MPI_Init :initiate MPI computation
  - MPI_Finalize :terminate computation
  - MPI_COMM_SIZE :determine number of processors
  - MPI_COMM_RANK :determine my process identifier
  - MPI_SEND :send a message
  - MPI_RECV :receive a message

- Supports point-point, broadcast communications

Examples of programming MPI next week (for assignment)

HelloWorld MPI

```
/* C Example */
#include <stdio.h>
#include <mpi.h>

int main (argc, argv)
int argc;
char *argv[];
{
        int rank, size;
        MPI_Init (&argc, &argv); /* starts MPI */
        MPI_Comm_rank (MPI_COMM_WORLD, &rank); /* get current process id */
        MPI_Comm_size (MPI_COMM_WORLD, &size); /* get number of processes */
        printf( "Hello world from process %d of %d\n", rank, size );
        MPI_Finalize();
        return 0;
}
```

```
$mpicc -o hello_world hello_world.c
$ mpirun -np 4 hello_world
Hello world from process 1 of 4
Hello world from process 2 of 4
Hello world from process 3 of 4
Hello world from process 0 of 4
```

(HT)Condor

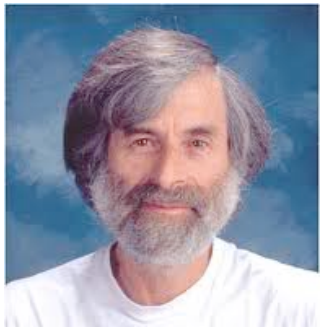
- A specialized workload management system for compute-intensive jobs developed at University of Wisconsin
  - Offers job queueing mechanisms, scheduling policies, priority schemes, resource monitoring/management
  - User submits jobs to Condor and it chooses when and where to run the jobs, monitors their progress, and informs the user upon completion
  - Allows to harvest “free” (?) CPU power from otherwise idle desktop workstations
    - e.g. use desktop machines when keyboard and mouse are idle
      - key press detected checkpoint and migrate a job to a different (idle) machine
  - No need for shared file system across machines
    - Data can be staged to machines as/when needed
  - Can work across organisational boundaries
    - Condor Flocking
  - ClassAds
    - Advertise resources and accept jobs (according to policy)

Break

Challenges with Distribution

- *A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable*

- *Leslie Lamport*



Challenges with Distribution

- *General assumptions that typically don't hold in the real world…*

(Some) Erroneous Assumptions of Distributed Systems

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous
9. Time is ubiquitous

Network reliability

- If I send some data over the network
  - It will arrive
  - It will arrive in the order I sent it
  - It will arrive uncorrupted
- If the network is unreliable it will be so in only one of the above ways
  - Consistently
- The lower layers in the networking stack protect me from these issues
- None of these statements are always true!

Erroneous Assumptions of Distributed Systems

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous
9. Time is ubiquitous

Latency is zero

- If I send some data it will arrive “now”.
  - Or so fast to be effectively “now”
- Really…?
- Consider accessing Twitter (San Francisco) from Melbourne
  - ~12,600km

The Route (traceroute)

```
1.  128.250.7.66                                          0.646ms
2.  172.18.65.89                                          1.135ms
3.  fr151-v2010-vpn-general.unimelb.net.au                1.203ms
4.  fr151-v2011-vpn-fusion-general.unimelb.net.au         0.895ms
5.  br-151-te-2-3.unimelb.net.au                          4.696ms
6.  gigabitethernet1.er1.unimelb.cpe.aarnet.net.au        0.887ms
7.  ge-5-1-0.bb1.a.mel.aarnet.net.au                      1.223ms
8.  so-0-1-0.bb1.a.syd.aarnet.net.au                     15.158ms
9.  so-2-2-0.bb1.a.pao.aarnet.net.au                    171.461ms
10. paix1.cr1.pao1.twttr.com                            172.682ms
11. ae52.smf1-er1.twttr.com                             177.191ms
```

Latency Consequences

- On a 1Gbps link
  - 9MB of data can be in flight
    - Or 145 TCP (maximum) packets
  - You could have sent 18MB of data before you start receiving responses, all of which will be in your TCP stack's buffer
- You *can't* get a message in less than 78ms
- You *can't* get a reply in less than 156ms
- Latency can be much higher
- In a system with many nodes/hops each link can have different latencies

Erroneous Assumptions of Distributed Systems

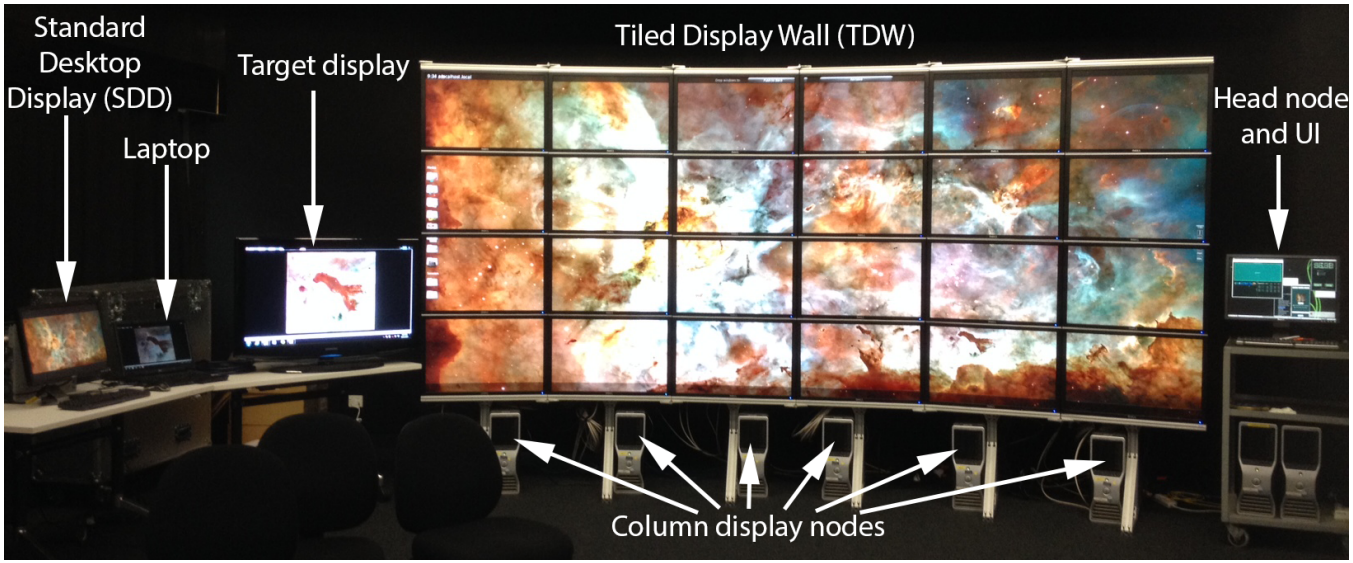
1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous
9. Time is ubiquitous

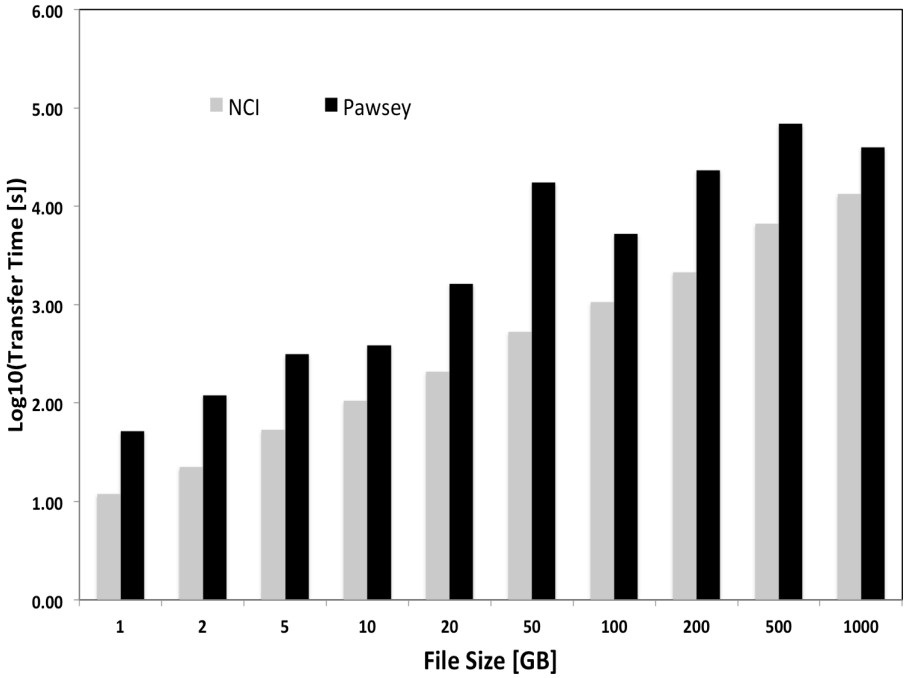
Bandwidth is infinite

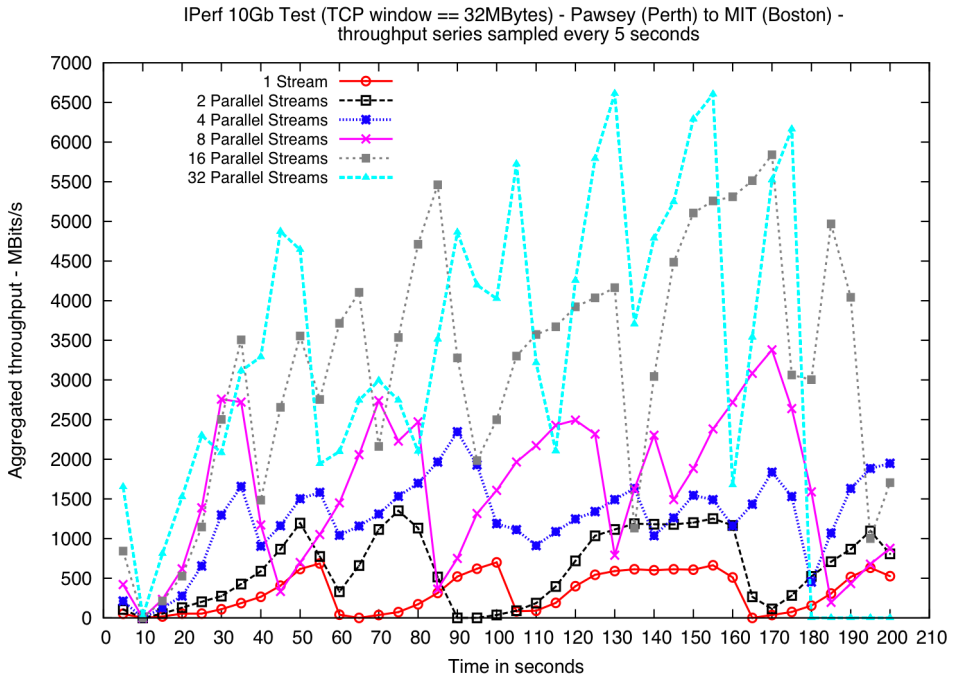
- I can send any amount of data I wish between any nodes

- Really?
  - Think of BIG DATA (Tb->Pb+)
    - SKA, LHC, Genomics, …
      - And consequences for others, e.g. Melbourne hospitals
    - Network bandwidth typically of the order
      - 1GB, 10GB, 100GB, …
      - For hospitals, homes varies wildly…
    - Networking capacity carefully planned
      - JANET (UK) backbone planned for up to 25% actual bandwidth capacity actually used

Bandwidth is infinite







Erroneous Assumptions of Distributed Systems

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous
9. Time is ubiquitous

The network is secure

- Really…?
- So I don’t have to worry about…
  - People sending data to my services
    - Repeated password attempts, SQL injections, …!?
  - People actively attacking me
    - Distributed denial of service attacks
  - People reading the data sent over the network
    - Man in the middle attacks
  - People masquerading as one of my nodes
    - Spoofing
  - People breaking into one of my nodes
    - Trojans, viruses, brute force attacks, …
  - People stealing the physical hardware

Erroneous Assumptions of Distributed Systems

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous
9. Time is ubiquitous

Topology doesn't change

- Node x is always *there*
- Really?
  - There = ?
    - Latency
    - IP
    - Route
    - Services
  - Typically can’t guarantee the route taken and hence latency
    - unless specific routing protocols selected, e.g. differentiated services with reservation protocols
    - Mostly not available (TCP/IP)
      - Next hop behaviour =?

Erroneous Assumptions of Distributed Systems

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous
9. Time is ubiquitous

There is one administrator

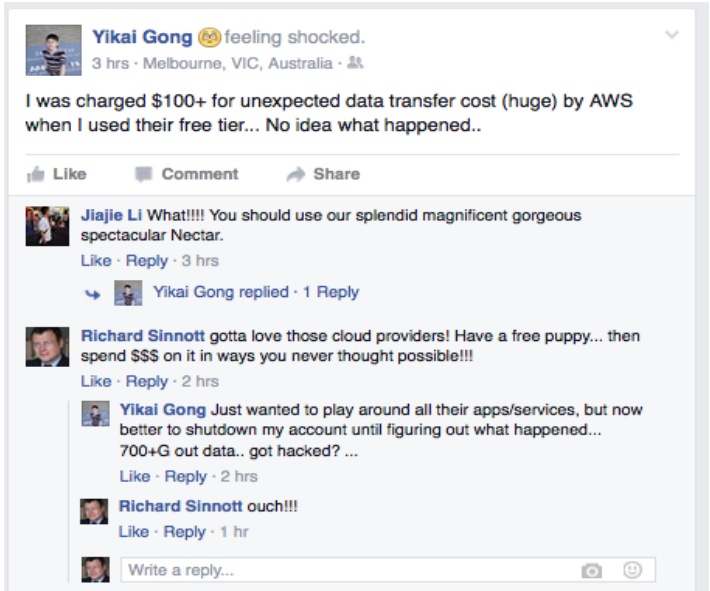
- Really?
  - Pretty much never the case
  - Who is administrator for University of Melbourne
  - Firewall changes, server reconfigurations, services, access control (students/staff/ others…)
  - Inter-organisational administrators?

Erroneous Assumptions of Distributed Systems

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous
9. Time is ubiquitous

Transport cost is zero

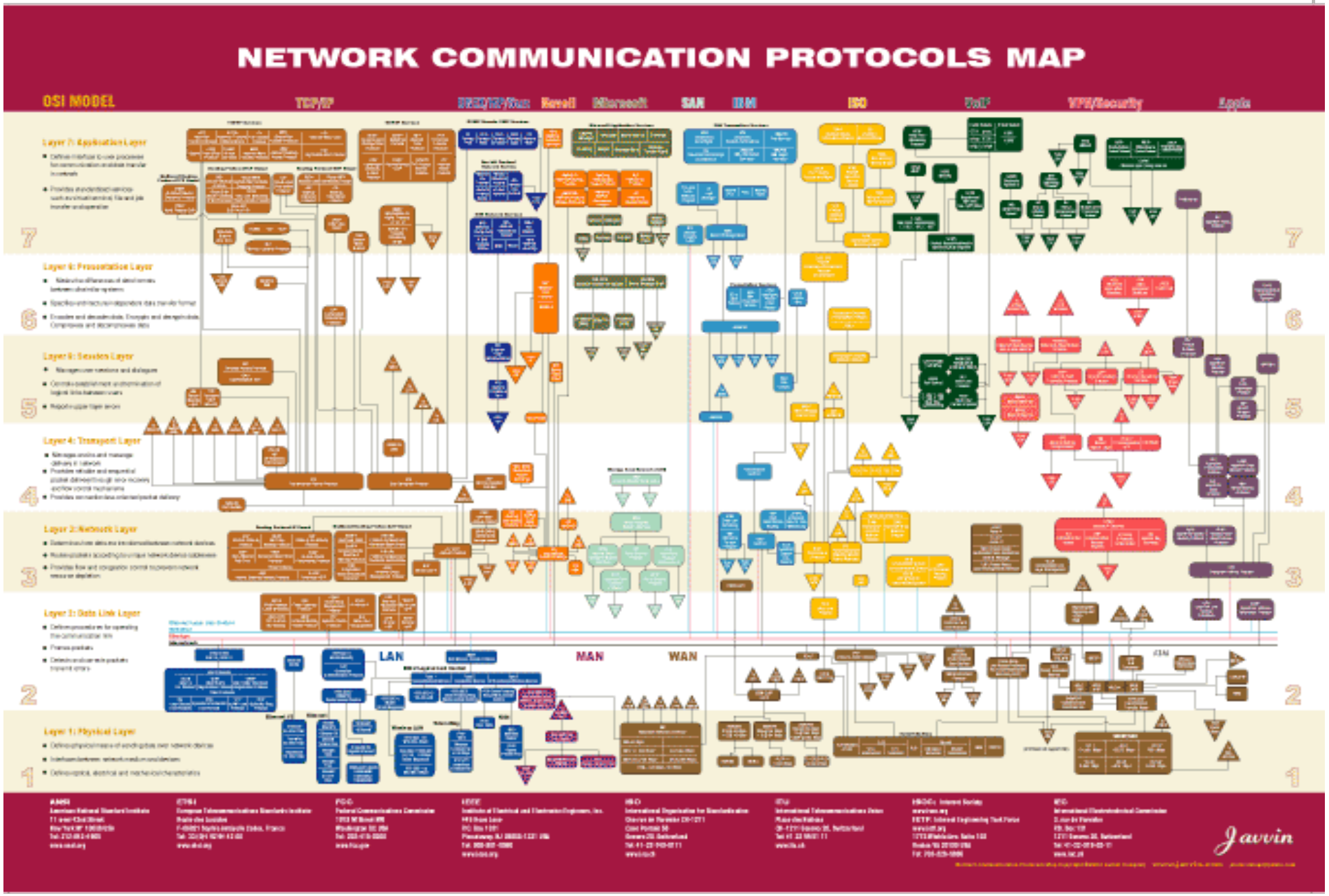
- I can send as much data as I like for free
- Really?
  - Can appear so in some places, but…
    - Australia…!?!?!?!
      - Capped uploads/downloads
      - UniMelb researchers with $50k bill for data access that wasn't over AARNET
      - Amazon EC2/S3…$$$$$
  - It is NEVER free!



Erroneous Assumptions of Distributed Systems

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous
9. Time is ubiquitous

The network is homogeneous

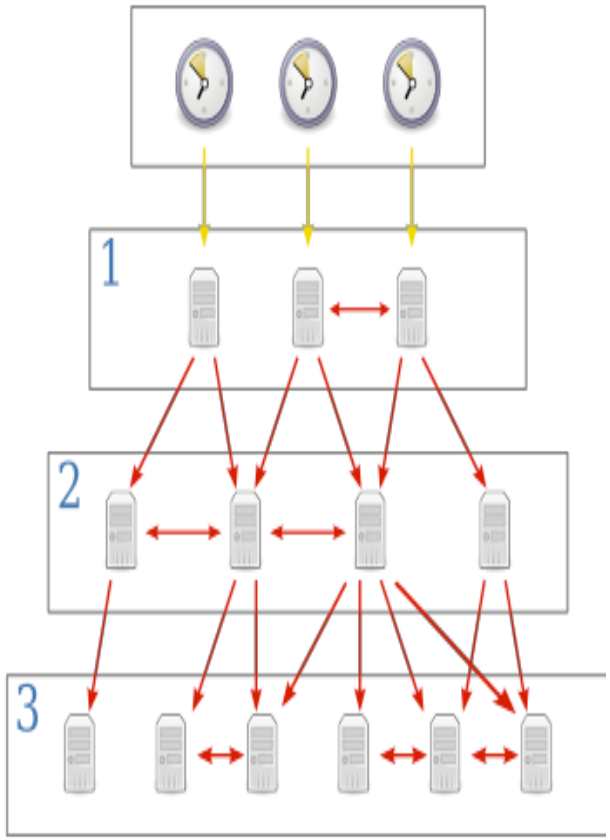


Erroneous Assumptions of Distributed Systems

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous
9. Time is ubiquitous

Time is ubiquitous

- Is same across all computers in network
- Really?
  - Clocks, clock cycles
    - Clock drift, skew, …
  - Protocols
    - NTP synchronizes participating computers to within a few milliseconds of Coordinated Universal Time (UTC)
  - Spaghetti westerns
    - Lip-synchronisation algorithms
    - Many approaches to reduce latency, jitter, …
  - For some applications this is a MAJOR issue, e.g. finance!



Despite this…

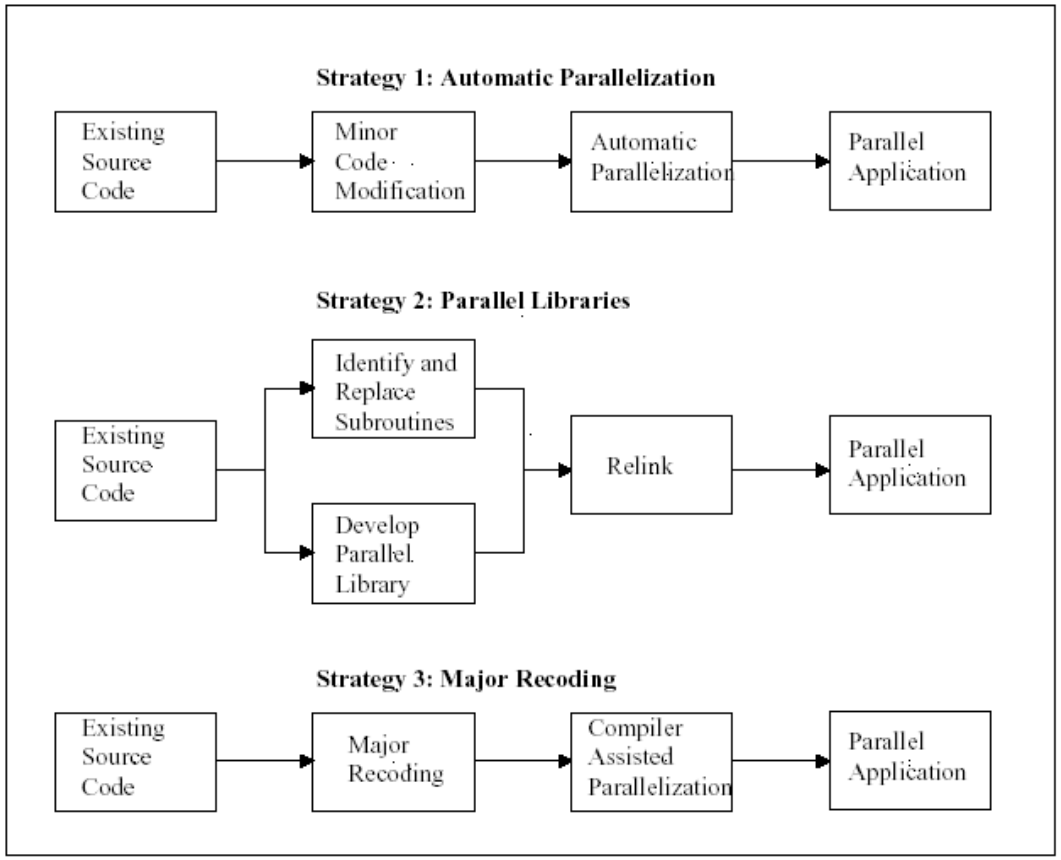
- …and issues of heterogeneity of compute, data, security, …
    - (Remember lecture 1)

  – Distributed systems are widespread
    - The Internet!

  – Designing parallel or distributed systems
    - Many approaches
      – No single algorithm
      – No single technical solution
      – Eco-system of approaches explored over time and many open research questions/challenges
      – Flavour of some of these…

Break

Strategies for Development of Parallel/Distributed Systems



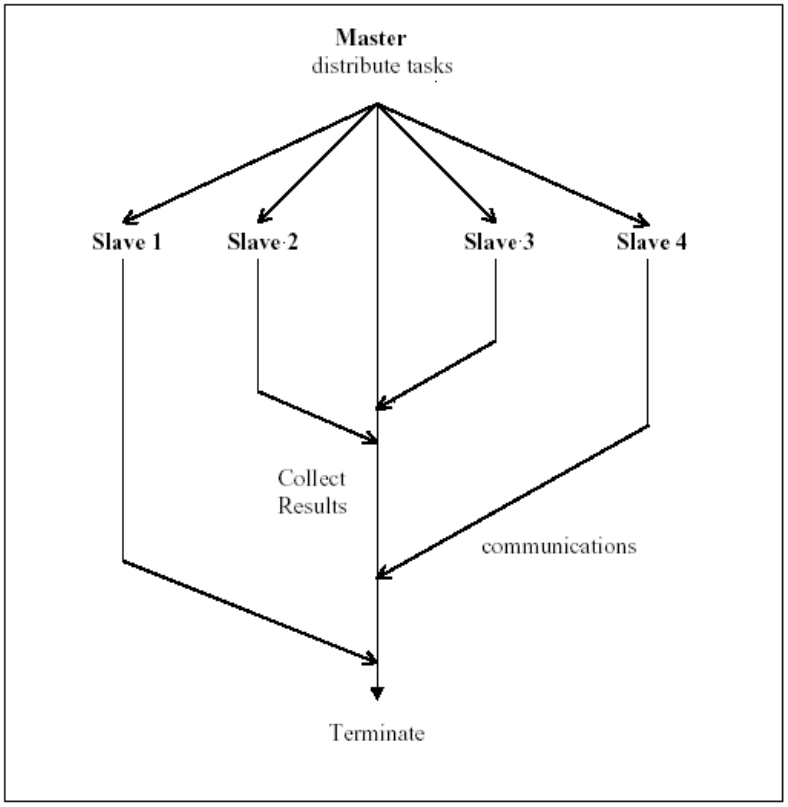
- Challenges
  - dependence analysis is hard for code that uses pointers, recursion, …; loops can have unknown number of iterations; access to global resources, e.g. shared variables

Design Stages of Parallel Programs

- Partitioning
  - Decomposition of computational activities and data into smaller tasks
  - Numerous paradigms:
    - Master-worker, SPMD, pipeline, divide and conquer, speculation
- Communication
  - Flow of information and coordination among tasks that are created in the partitioning stage
- Agglomeration
  - Tasks and communication structure created in the above stages are evaluated for performance and implementation cost
  - Tasks may be grouped into larger tasks to improve communication
  - Individual communications can be bundled
- Mapping / Scheduling
  - Assigning tasks to processors such that job completion time is minimized and resource utilization is maximized.

Master Worker/Slave Model

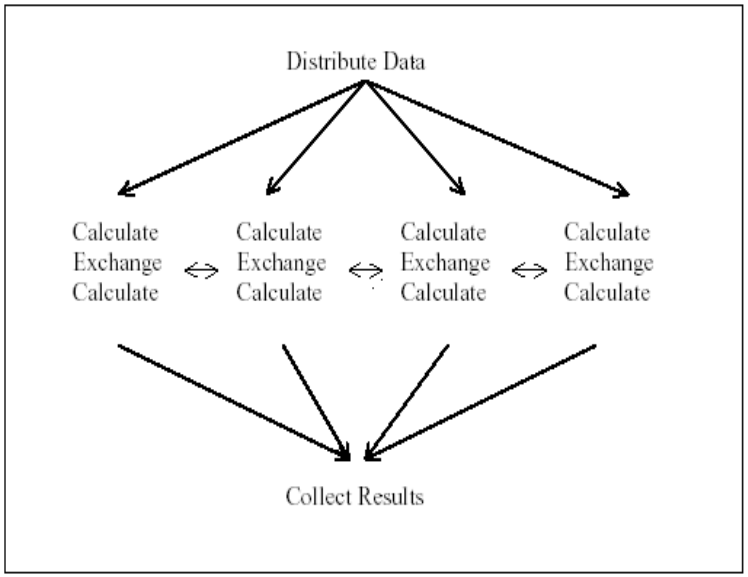
- Master decomposes the problem into small tasks, distributes to workers and gathers partial results to produce the final result
- Realised in many ways of different levels of granularity, e.g. threads through to web service workflow definition and enactment



Single-Program Multiple-Data

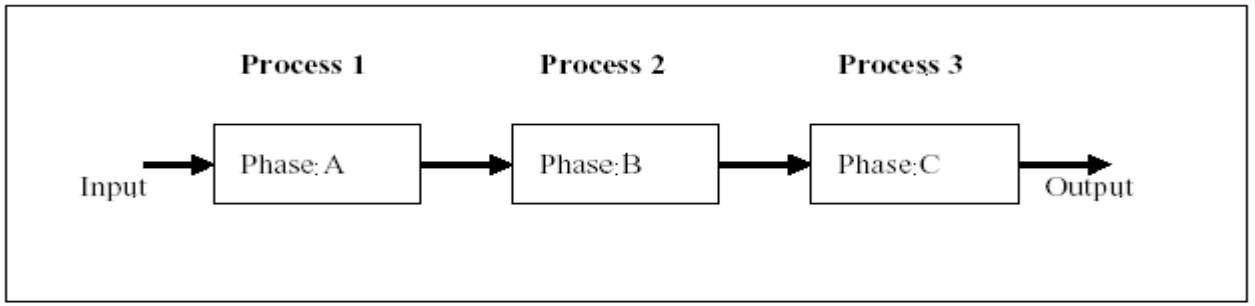
- Commonly exploited model
  - Bioinformatics, MapReduce, …
- Each process executes the same piece of code, but on different parts of the data
- Data is typically split among the available processors
- Data splitting and analysis can be done in many ways

  (Assignment 1!!!)



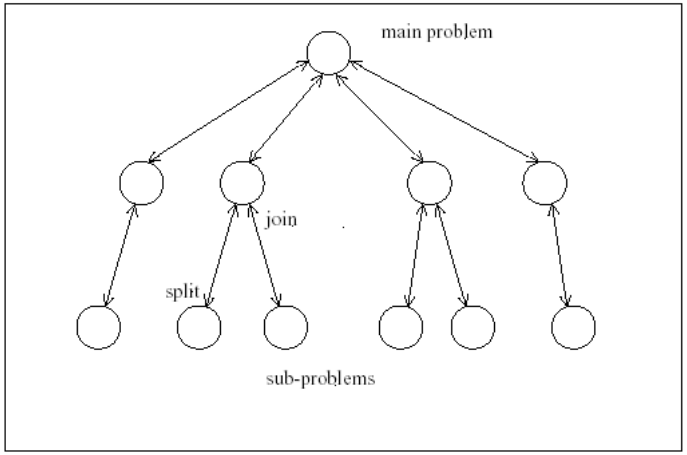
Data Pipelining

- Suitable for applications involving multiple stages of execution, that typically operate on large number of data sets.



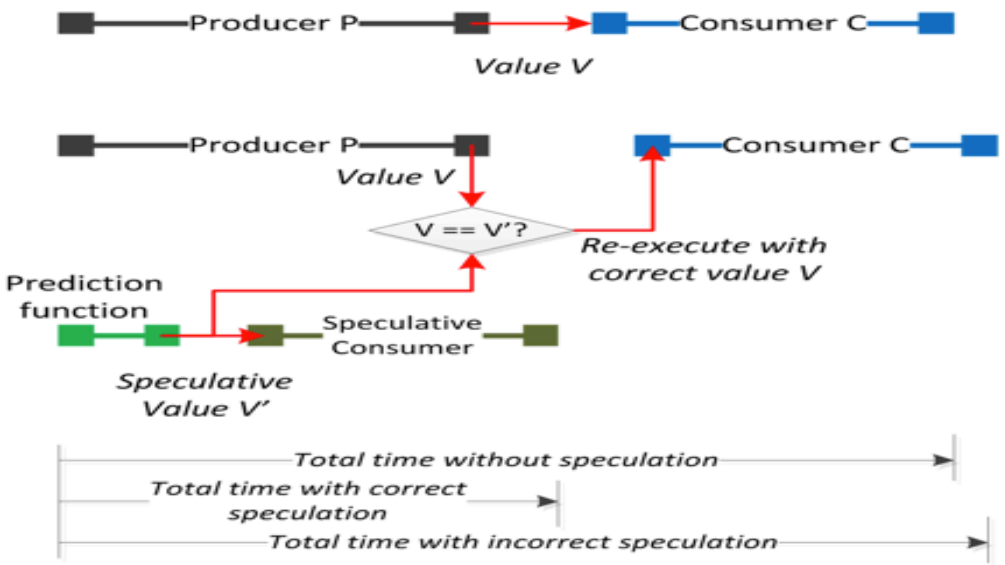
Divide and Conquer

- A problem is divided into two or more sub problems, and each of these sub problems are solved independently, and their results are combined
- 3 operations: split, compute, and join
- Master-worker/task-farming is like divide and conquer with master doing both split and join operation



Speculative Parallelism

- Used when it is quite difficult to achieve parallelism through the previous paradigms
  - Problems with complex dependencies – use "*look ahead* "execution
  - Consider a (long running) producer P and a consumer C such that C depends on P for the value of some variable V. If the value of V is *predictable*, we can execute C speculatively using a predicted value in parallel with P.
    - If the prediction turns out to be correct, we gain performance since C doesn't wait for P anymore.
    - If the prediction is incorrect (which we can find out when P completes), we have to take corrective action, cancel C and restart C with the right value of V again.



Summary

- Lots of open issues with design, implementation and verification of distributed and parallel systems
- Hardware refinements/evolution still happening
  - FPGAs, GPUs, …
  - Flavours of data storage
- Software challenges/algorithms
- Design issues – often challenged by domain needs
  - Reliability vs performance
  - Security vs scale
  - …

Questions