Day 1:
High Performance Computing
CMT106

David W. Walker

Professor of High Performance Computing

Cardiff University

http://www.cardiff.ac.uk/people/view/118172-walker-david

Module Outline

- Seven all-day sessions held in S/1.29 from 9:30am to 5:00pm (apart from labs in C/2.08):
  - Week 5: Tuesday, 29 October 2019
  - Week 6: Tuesday, 5 November 2019
  - Week 7: Tuesday, 12 November 2019
  - Week 8: Tuesday, 19 November 2019
  - Week 9: Tuesday, 26 November 2019
  - Week 10: Tuesday, 3 December 2019
  - Week 11: Tuesday, 10 December 2019
- Sessions will be mix of lectures, informal labs, and discussion of papers and articles.

Assessment

- Exam (2 hours) = 70%
- 2 pieces of coursework each worth 15%:
  - Must do OpenMP coursework, and either MPI or CUDA coursework.
- Coursework set in weeks 6 (OpenMP), 8 (MPI), and 9 (CUDA)
- Coursework due at 9:30am on Friday of week 11
- Module description is at http://handbooks.data.cardiff.ac.uk/module/CMT106.html

Day 1

- 9:00-approx. 11:00: lecture on introduction to C
- 11:00-12:50: lecture on introduction to parallelism; motivation; types of parallelism; Flynn's taxonomy; shared and distributed memory

- Afternoon: self-study:
  - find 3 interesting/important things about the Top500 list
  - read "Back to thin-core massively parallel processors"

Topics Covered on Days 2-4

- *Day 2*: Programming with OpenMP; interconnection networks; network metrics; classification of parallel algorithms; speedup and efficiency.
- *Day 3*: Scalable algorithms; Amdahl's law; sending and receiving messages; programming with MPI; collective communication.
- Day 4: Integration example; regular computations and another simple example.

Topics Covered on Days 5-7

- *Day 5*: Programming GPUs with CUDA; regular two-dimensional problems and an example.
- *Day 6*: Dynamic communication and the molecular dynamics example; irregular computations; the WaTor simulation .
- *Day 7*: Load balancing strategies; message passing libraries; block-cyclic data distribution.

Books

- "Introduction to Parallel Programming," Peter Pacheco, published by Morgan Kaufmann, 2011. http://store.elsevier.com/product.jsp?isbn=9780123742605
- "Using MPI," Gropp, Lusk, and Skjellum, published by MIT Press, 1994.
- "Programming Massively Parallel Processors," David B. Kirk and Wen-mei W. Hwu, third edition, pub. Morgan Kaufmann, 2016. ISBN 978-0-12-811986-0. https://www.elsevier.com/books/programming-massively-parallel-processors/kirk/978-0-12-811986-0
- "Parallel Programming," B. Wilkinson and M. Allen, published by Prentice Hall, 1999. ISBN 0-13-671710-1.
- "Solving Problems on Concurrent Processors, Volume 1," Fox, Johnson, Lyzenga, Otto, Salmon, and Walker, published by Prentice-Hall, 1988.

Web Sites

- For the module: http://learningcentral.cf.ac.uk.
- For MPI: http://www.mcs.anl.gov/mpi/
- For OpenMP: https://computing.llnl.gov/tutorials/openMP/
- For information on the World's fastest supercomputers: http://www.top500.org/

What is Parallelism?

- *Parallelism* refers to the simultaneous occurrence of events on a computer.
- An event typically means one of the following:
  - An arithmetical operation
  - A logical operation
  - Accessing memory
  - Performing input or output (I/O)

Types of Parallelism 1

- Parallelism can be examined at several levels.
  - Job level: several independent jobs simultaneously run on the same computer system.
  - Program level: several tasks are performed simultaneously to solve a single common problem.

Types of Parallelism 2

- Instruction level: the processing of an instruction, such as adding two numbers, can be divided into sub-instructions. If several similar instructions are to be performed their sub-instructions may be overlapped using a technique called *pipelining*.
- Bit level: when the bits in a word are handled one after the other this is called a *bit-serial* operation. If the bits are acted on in parallel the operation is *bit-parallel.*

In this parallel processing course we shall be mostly concerned with parallelism at the program level.

Concurrent processing is the same as parallel processing.

Scheduling Example

Time	Jobs running	Utilisation
1	S, M	75%
2	L	100%
3	S, S, M	100%
4	L	100%
5	L	100%
6	S, M	75%
7	M	50%

- Average utilisation is 85.7%
- Time to complete all jobs is 7 time units.

A Better Schedule

- A better schedule would allow jobs to be taken out of order to give higher utilisation.

S M L S S M L L S M M

- Allow jobs to “float” to the front to the queue to maintain high utilisation.

Time	Jobs running	Utilisation
1	S, M, S	100%
2	L	100%
3	S, M, S	100%
4	L	100%
5	L	100%
6	M, M	100%

Notes on Scheduling Example

- In the last example:
  - Average utilisation is 100%.
  - Time to complete all jobs is 6 time units.
- Actual situation is more complex as jobs may run for differing lengths of time.
- Real job scheduler must balance high utilisation with fairness (otherwise large jobs may never run).

Parallelism Between Job Phases

- Parallelism also arises when different independent jobs running on a machine have several phases, e.g., computation, writing to a graphics buffer, I/O to disk or tape, and system calls.
- Suppose a job is executing and needs to perform I/O before it can progress further. I/O is usually expensive compared with computation, so the job currently running is suspended, and another is started. The original job resumes after the I/O operation has completed.
- This requires special hardware: I/O channels or extra processor for I/O.
- The *operating system* controls how different jobs are scheduled and share resources.

Program Level Parallelism

This is parallelism between different parts of the same job.

Example

A robot has been programmed to look for electrical sockets when it runs low on power. When it finds one it goes over to it and plugs itself in to recharge. Three subsystems are involved in this - the vision, manipulation, and motion subsystems. Each subsystem is controlled by a different processor, and they act in parallel as the robot does different things

Robot Example

Task	Vision	Manipulation	Motion
1. Looking for electrical socket	✕		✕
2. Going to electrical socket	✕		✕
3. Plugging into electrical socket	✕	✕	

Notes on Robot Example

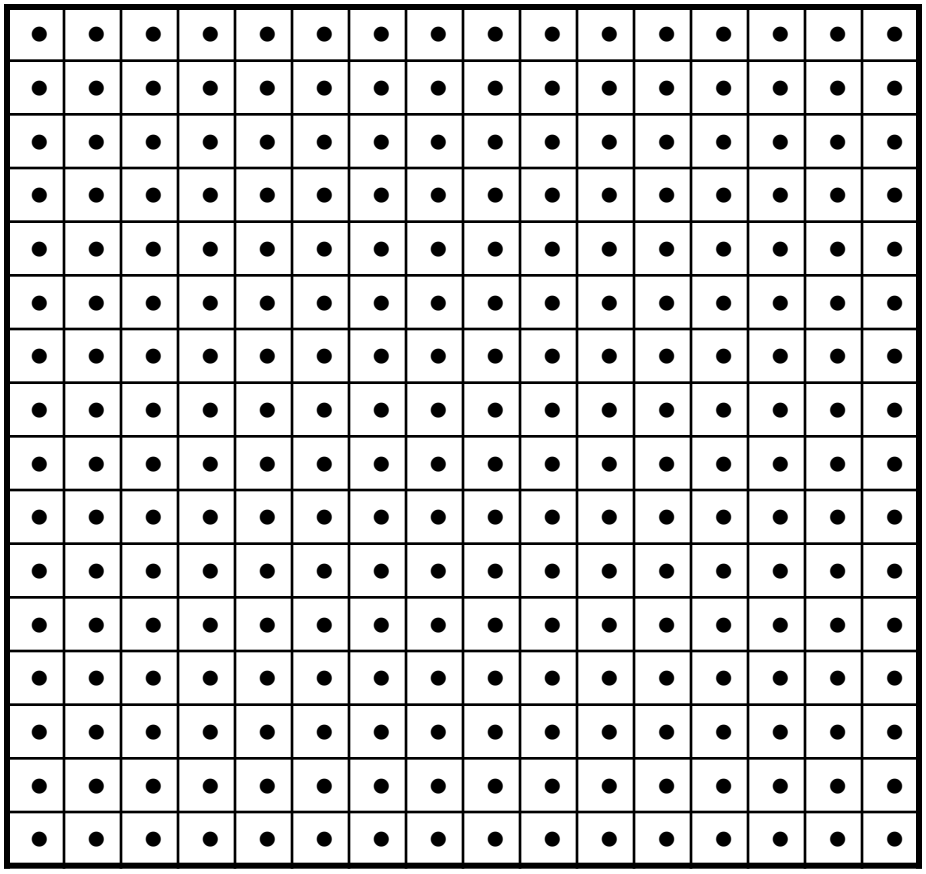
- The subsystems are fairly independent, with the vision subsystem guiding the others.
- There may also be a central “brain” processor.
- This is an example of *task parallelism* in which different tasks are performed concurrently to achieve a common goal.

Domain Decomposition

A common form of program-level parallelism arises from the division of the data to be programmed into subsets.

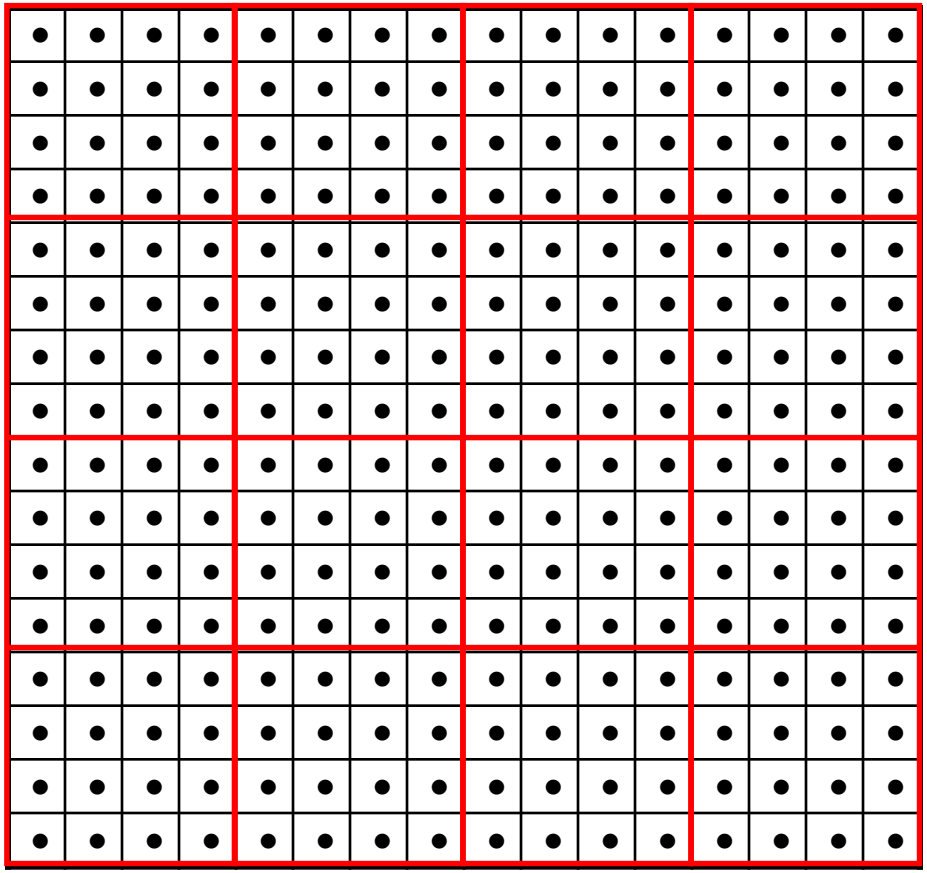
- This division is called *domain decomposition*.
- Parallelism that arises through domain decomposition is called *data parallelism*.
- The data subsets are assigned to different computational processes. This is called *data distribution*.
- Processes may be assigned to hardware processors by the program or by the runtime system. There may be more than one process on each processor.

Data Parallelism



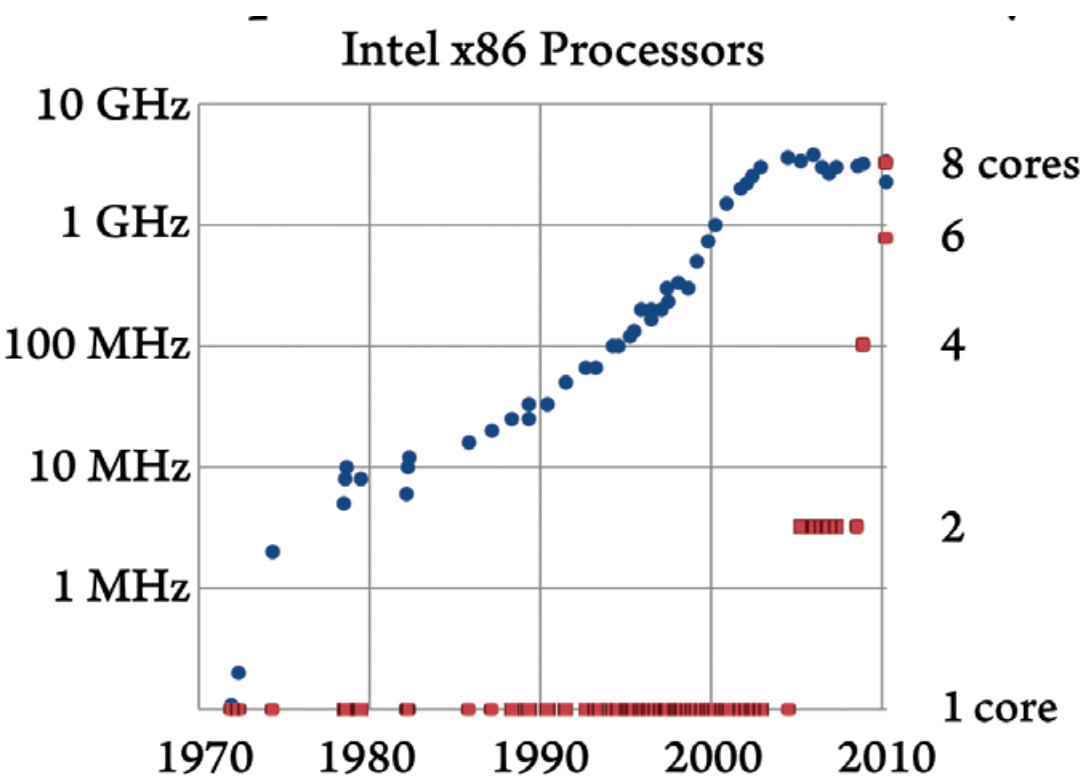
- Consider an image digitised as a square array of pixels which we want to process by replacing each pixel value by the average of its neighbours.
- The *domain* of the problem is the two-dimensional pixel array.

Domain Decomposition



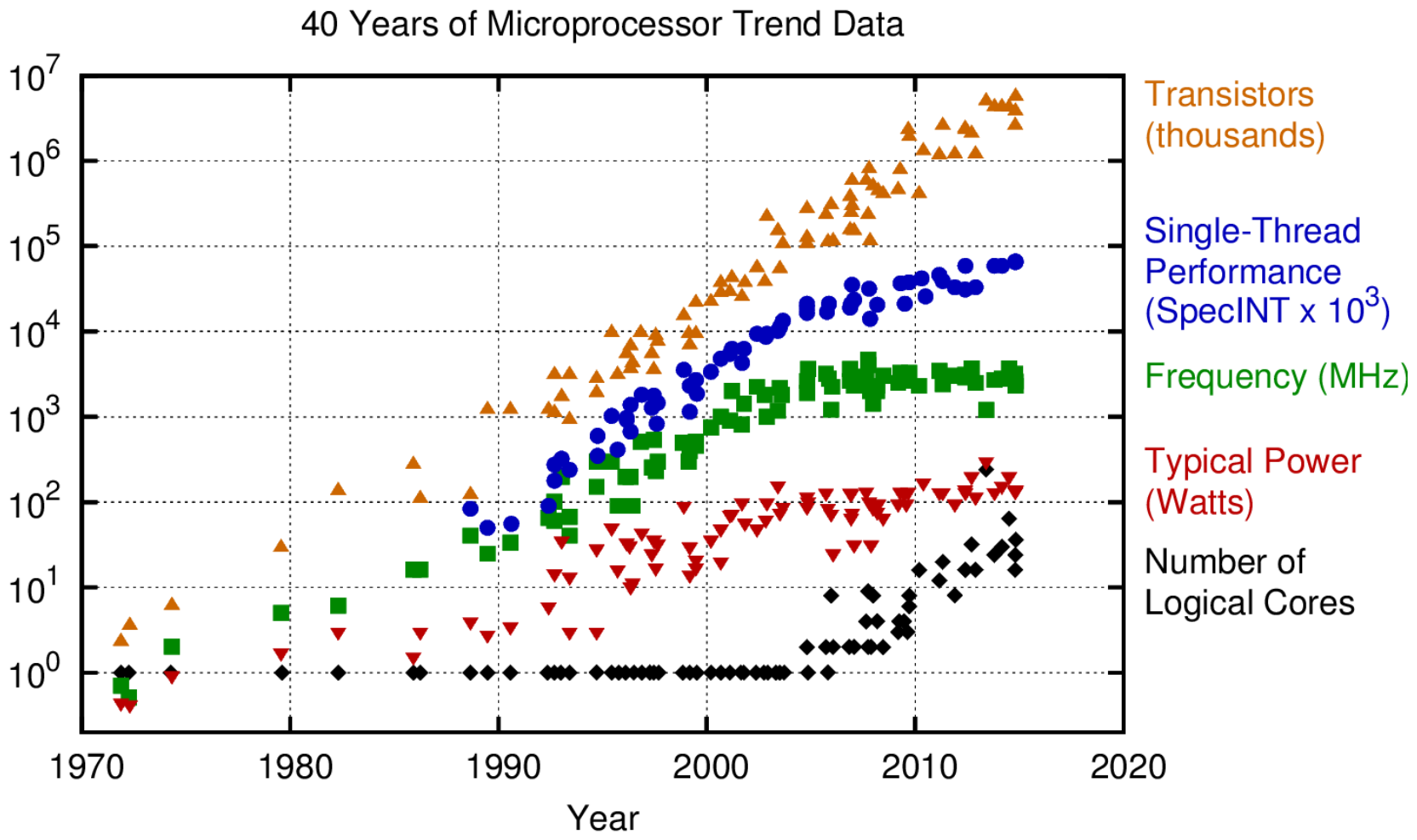
- Suppose we decompose the problem into 16 subdomains
- We then distribute the data by assigning each subdomain to a process.
- The pixel array is a *regular* domain because the geometry is simple.
- This is a homogeneous problem because each pixel requires the same amount of computation (almost - which pixels are different?).

Pervasive Parallelism



- Increases in clock frequency stopped in about 2003 due to power consumption and cooling problems.
- Chip designers now put multiple processing cores on a single chip.
- These cores can be used in parallel for a variety of tasks.
- GPUs can also be used to exploit parallelism.

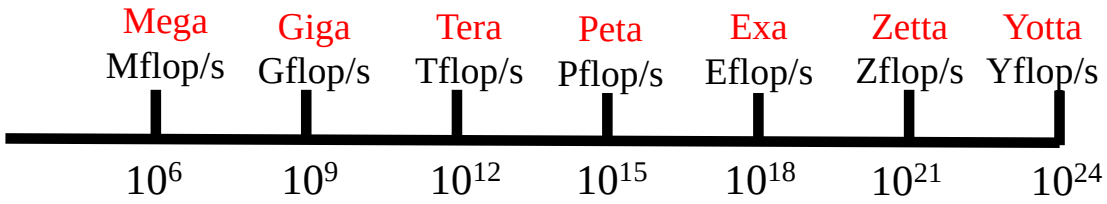
Moore's Law



Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten
New plot and data collected for 2010-2015 by K. Rupp

Why Use Parallelism?

- Better utilisation of resources. Want to keep hardware busy.
- Want to run programs faster by spreading work over several processors.
- Measure of speed in (most) scientific problems:
  - Floating point operations per second. 1 Mflop/s is one million floating point operations per second.
    - High performance workstation ≈ 200-500 Gflop/s
    - GPU ≈ 1-2 Tflop/s
    - Current best supercomputer ≈ 122.3 Pflop/s



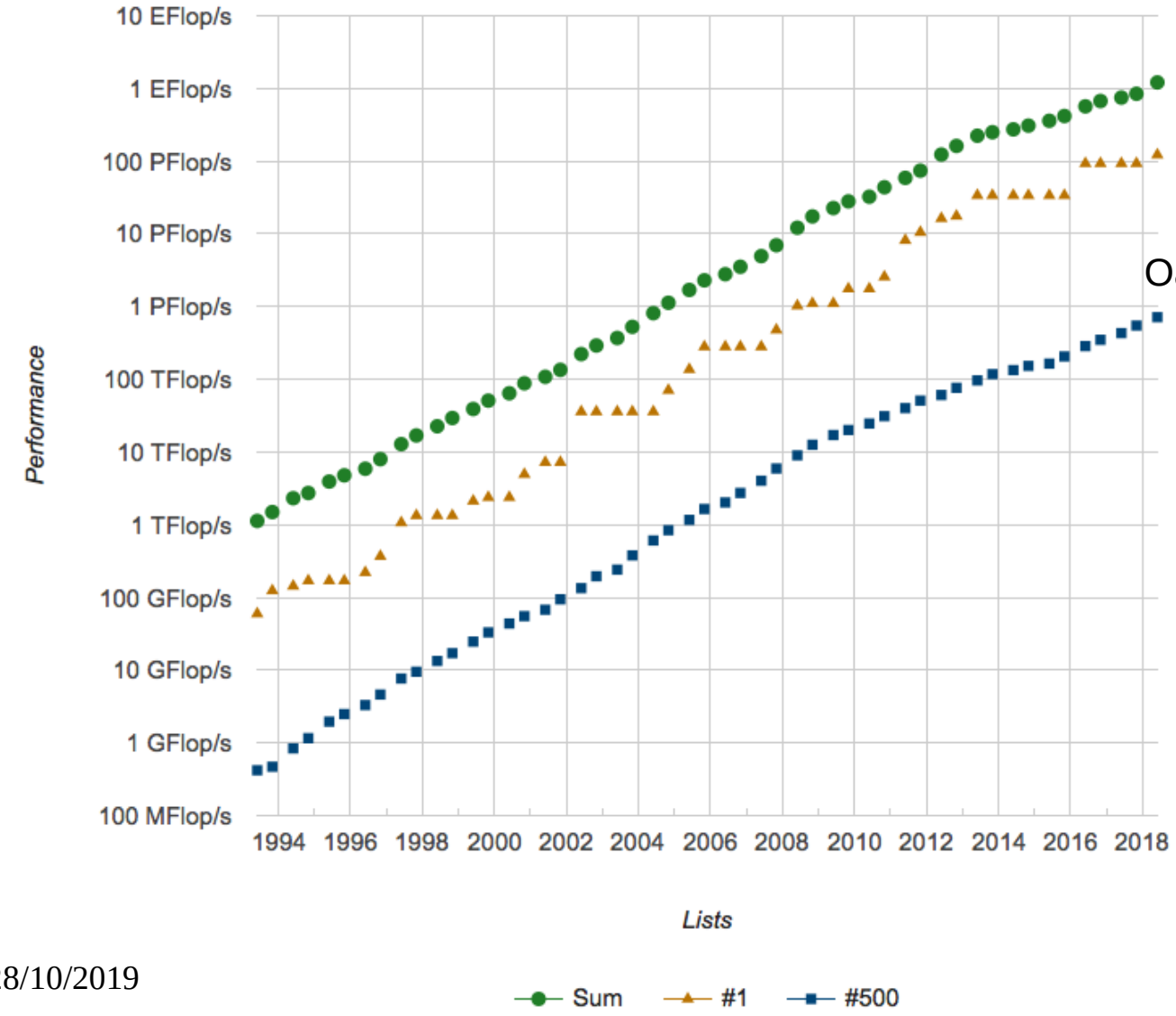
Parallelism and Memory

- Want more memory to solve bigger or more complex problems.
- Typical PCs/workstations have 4 Gbytes of RAM, expandable to 32 Gbytes. Can fit an 65,536 × 65,536 array of doubles into 32 Gbytes of memory.
- The "Summit" parallel computer at Oak Ridge National Lab in the USA has 2,414,592 cores with a total of 2.8 Pbyte of memory. Can fit a 18,708,286 × 18,708,286 array into memory. See https://www.top500.org/system/179397.

Parallelism and Supercomputing

- Parallelism is exploited on a variety of high performance computers, in particular *massively parallel processors* (MPPs) and *clusters*.
- MPPs, clusters, and high-performance vector computers are termed *supercomputers*.
- Currently supercomputers have peak performance in the range of 10-100 Pflop/s, and memory of 1 to 2 Pbytes. They cost about 100-200 million pounds.
- Supercomputers are leading to a new methodology in science called *computational science* joining theoretical and experimental approaches.

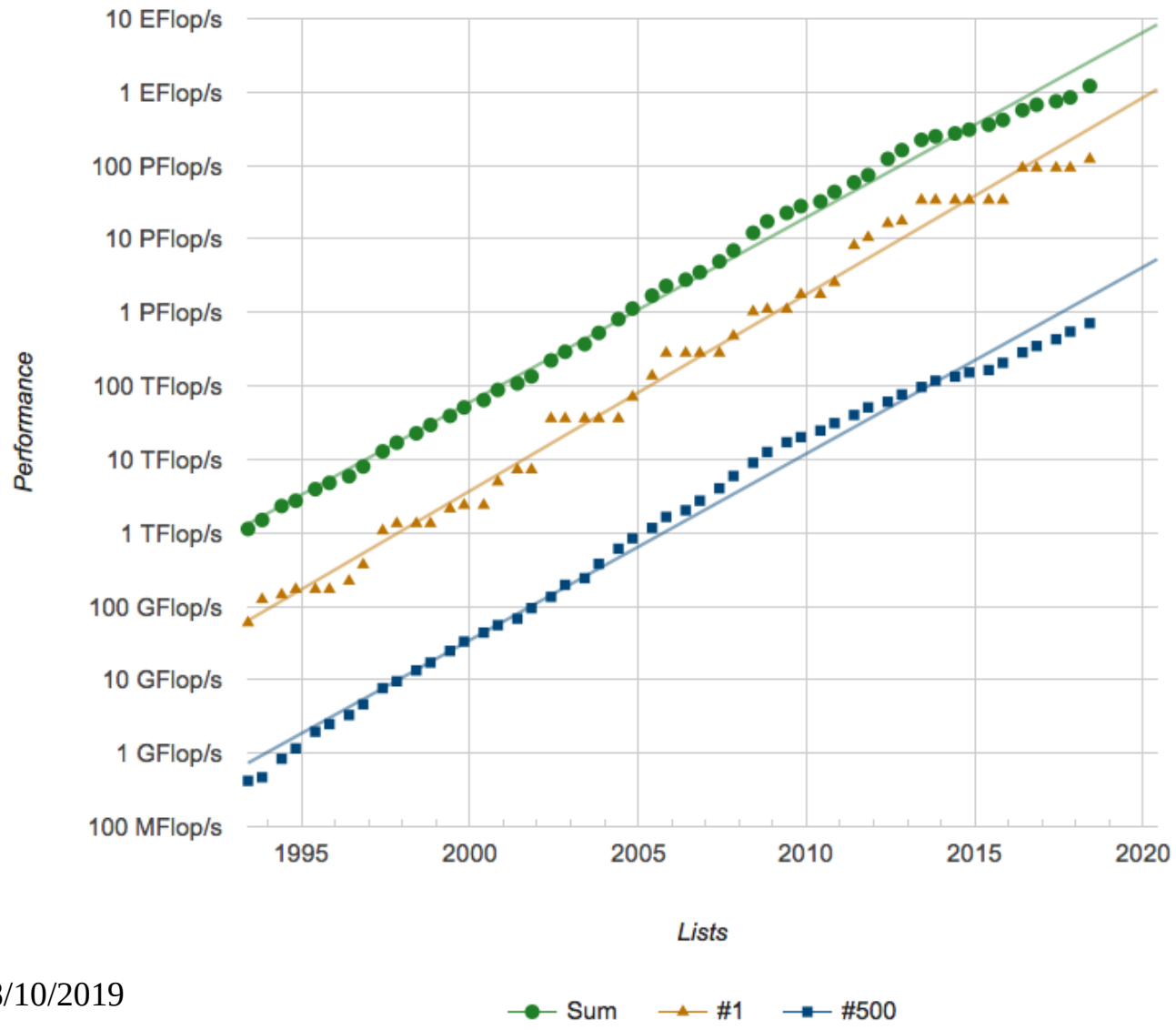
Performance Development



Peak at June 2019:
148.6 Pflop/s
2,414,592 cores
Oak Ridge National Lab, USA

Source: www.top500.org

Projected Performance Development



Source: www.top500.org

TOP500 List - June 2019

R_{max} and R_{peak} values are in TFlops. For more details about other fields, check the TOP500 description.

R_{peak} values are calculated using the advertised clock rate of the CPU. For the efficiency of the systems you should take into account the Turbo CPU clock rate where it applies.

previous | 1 | 2 | 3 | 4 | 5 | next

Rank	Site	System	Cores	Rmax (TFlop/s)	Rpeak (TFlop/s)	Power (kW)
1	DOE/SC/Oak Ridge National Laboratory United States	**Summit** - IBM Power System AC922, IBM POWER9 22C 3.07GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM	2,414,592	148,600.0	200,794.9	10,096
2	DOE/NNSA/LLNL United States	**Sierra** - IBM Power System S922LC, IBM POWER9 22C 3.1GHz, NVIDIA Volta GV100, Dual-rail Mellanox EDR Infiniband IBM / NVIDIA / Mellanox	1,572,480	94,640.0	125,712.0	7,438
3	National Supercomputing Center in Wuxi China	**Sunway TaihuLight** - Sunway MPP, Sunway SW26010 260C 1.45GHz, Sunway NRCPC	10,649,600	93,014.6	125,435.9	15,371
4	National Super Computer Center in Guangzhou China	**Tianhe-2A** - TH-IVB-FEP Cluster, Intel Xeon E5-2692v2 12C 2.2GHz, TH Express-2, Matrix-2000 NUDT	4,981,760	61,444.5	100,678.7	18,482
5	Texas Advanced Computing Center/Univ. of Texas United States	**Frontera** - Dell C6420, Xeon Platinum 8280 28C 2.7GHz, Mellanox InfiniBand HDR Dell EMC	448,448	23,516.4	38,745.9	
6	Swiss National Supercomputing Centre (CSCS) Switzerland	**Piz Daint** - Cray XC50, Xeon E5-2690v3 12C 2.6GHz, Aries interconnect , NVIDIA Tesla P100 Cray Inc.	387,872	21,230.0	27,154.3	2,384

Top500 Data

- It is interesting to use the sublist generator to look at different views of the data.
- By country – how does your country feature in the list?
- Who are the leading vendors of HPC systems?

SC14 Video about HPC

- https://www.youtube.com/watch?v=zJybFF6PqEQ

- Produced for the Supercomputing 2014 (SC14) conference. This is the premier HPC-related conference series. It attracts over 10000 attendees and takes place every November is the USA.

Uses of Parallel Supercomputers

- **Weather forecasting**. Currently forecasts are usually accurate up to about 5 days. This should be extended to 8 to 10 days over the next few years. Researchers would like to better model local nonlinear phenomena such as thunderstorms and tornadoes.
- **Climate modelling**. Studies of long-range behaviour of global climate. This is relevant to investigating global warming.
- **Engineering**. Simulation of car crashes to aid in design of cars. Design of aircraft in “numerical wind tunnels.‘”
- **Material science**. Understanding high temperature superconductors. Simulation of semiconductor devices. Design of lightweight, strong materials for construction.
- **Drug design**. Prediction of effectiveness of drug by simulation. Need to know configuration and properties of large molecules.

More Uses of Parallelism

- **Plasma physics**. Investigation of plasma fusion devices such as tokamaks as future source of cheap energy.
- **Economics**. Economic projections used to guide decision-making. Prediction of stock market behaviour. See http://spectrum.ieee.org/computing/it/financial-trading-at-the-speed-of-light
- **Defense**. Tracking of multiple missiles. Event-driven battlefield simulations. Code cracking.
- **Astrophysics**. Modeling internal structure of stars. Simulating supernova. Modeling the structure of the universe. See http://news.ucsc.edu/2011/09/bolshoi-simulation.html
- **Virtual worlds**. Simulation and rendering.

Computational Astrophysics

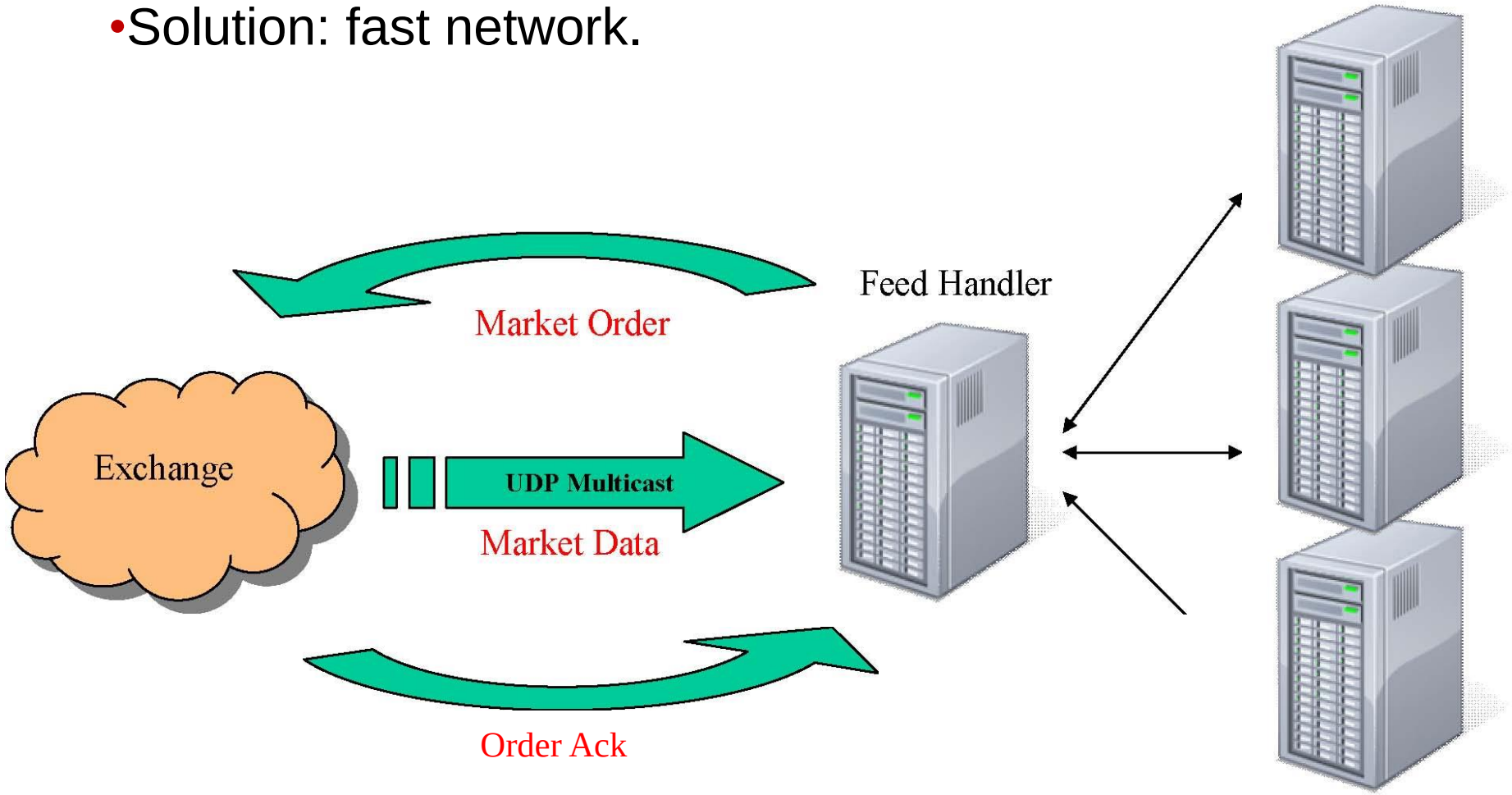
- **Astrophysics**. Modeling internal structure of stars. Simulating supernovae. Modeling the structure of the universe. See http://hipacc.ucsc.edu/Bolshoi/ and/or http://news.ucsc.edu/2011/09/bolshoi-simulation.html
- Professor Joel R. Primack, director of the High-Performance AstroComputing Center at the University of California at Santa Cruz:

> "Back in the early 1980's pencil and paper were the tools of choice for cosmologists. Now high performance computers have become vital. They've helped transform cosmology from philosophical speculation into what's almost an experimental science."

- View fly-through video of Bolshoi simulation: http://hipacc.ucsc.edu/Bolshoi/Movies.html#bsim

Financial Trading .

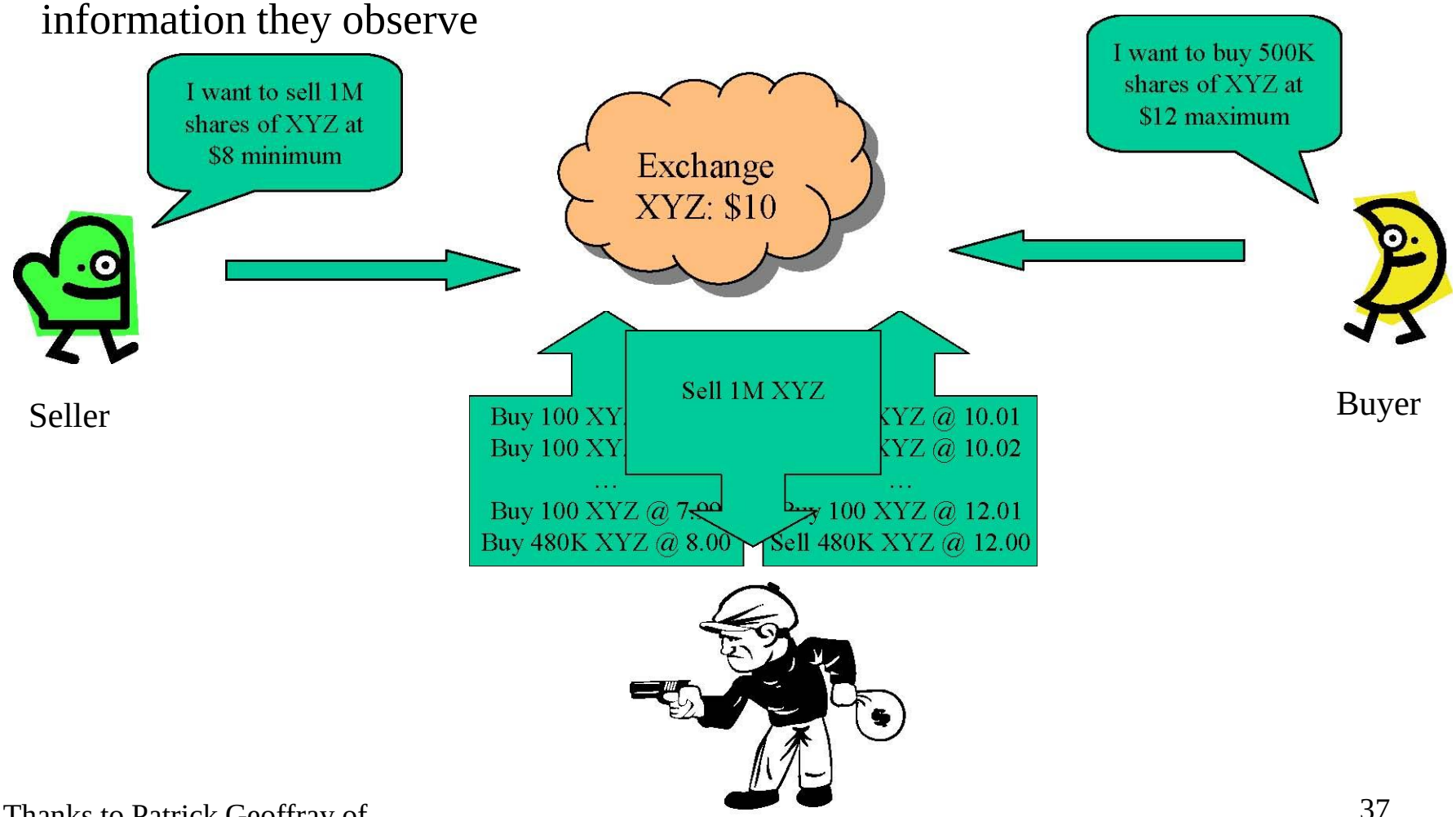
- Problem: trade faster than the other guys
- Metric: latency.
- Solution: fast network.



Thanks to Patrick Geoffray of Myricom for this slide.

High Frequency Trading

Computers make elaborate decisions to initiate orders based on information that is received electronically, before human traders are capable of processing the information they observe



Thanks to Patrick Geoffray of Myricom for this slide.

“Flash Crash” of May 2010

An automated trade execution system has been blamed for the 6 May 2010 stock market "flash crash" that affected trading worldwide.

Classification of Parallel Machines

- To classify parallel machines we must first develop a model of computation. The approach we follow is due to Flynn (1966).
- Any computer, whether sequential or parallel, operates by executing instructions on data.
  - a stream of **instructions** (the algorithm) tells the computer what to do.
  - a stream of **data** (the input) is affected by these instructions.

Classification of Parallel Machines

- Depending on whether there is one or several of these streams we have 4 classes of computers.
  - Single Instruction Stream, Single Data Stream: SISD
  - Multiple Instruction Stream, Single Data Stream: MISD
  - Single Instruction Stream, Multiple Data Stream: SIMD
  - Multiple Instruction Stream, Multiple Data Stream: MIMD

SISD Computers

This is the standard sequential computer.

A single processing unit receives a single stream of instructions that operate on a single stream of data

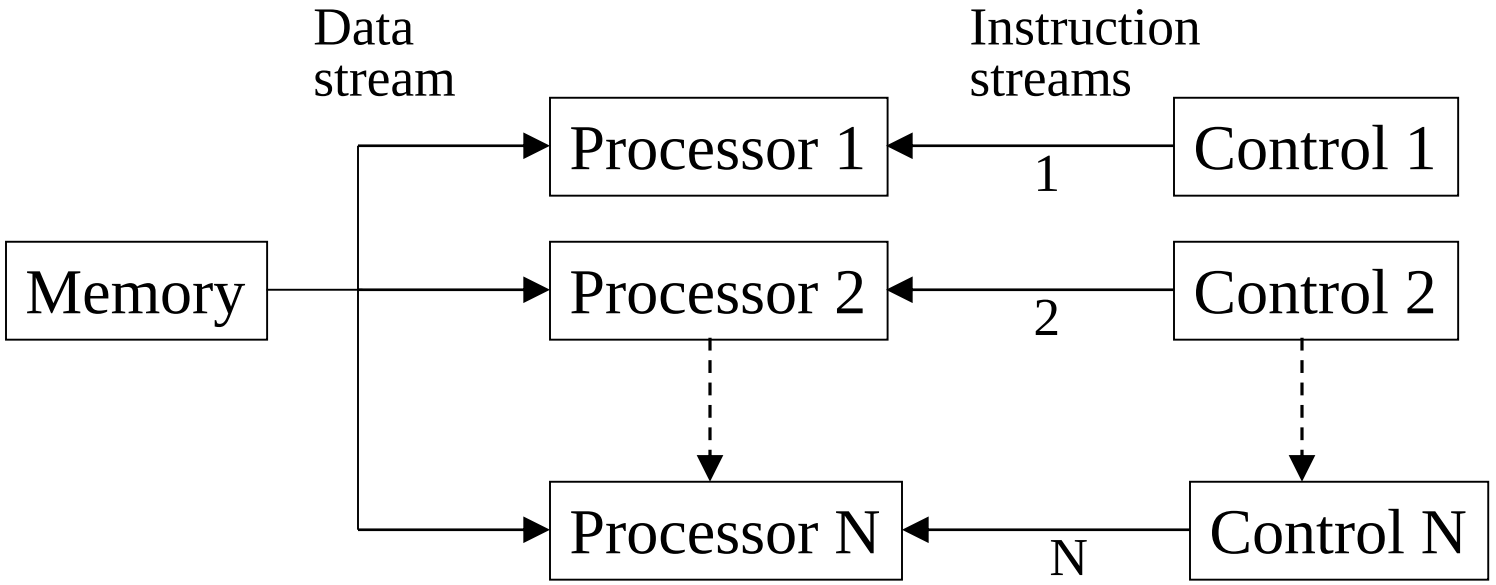
Example:

To compute the sum of N numbers $a_1,a_2,\ldots,a_N$ the processor needs to gain access to memory N consecutive times. Also N-1 additions are executed in sequence. Therefore the computation takes O(N) operations

Algorithms for SISD computers do not contain any process parallelism since there is only one processor.

MISD Computers

N processors, each with its own control unit, share a common memory.



MISD Computers (continued)

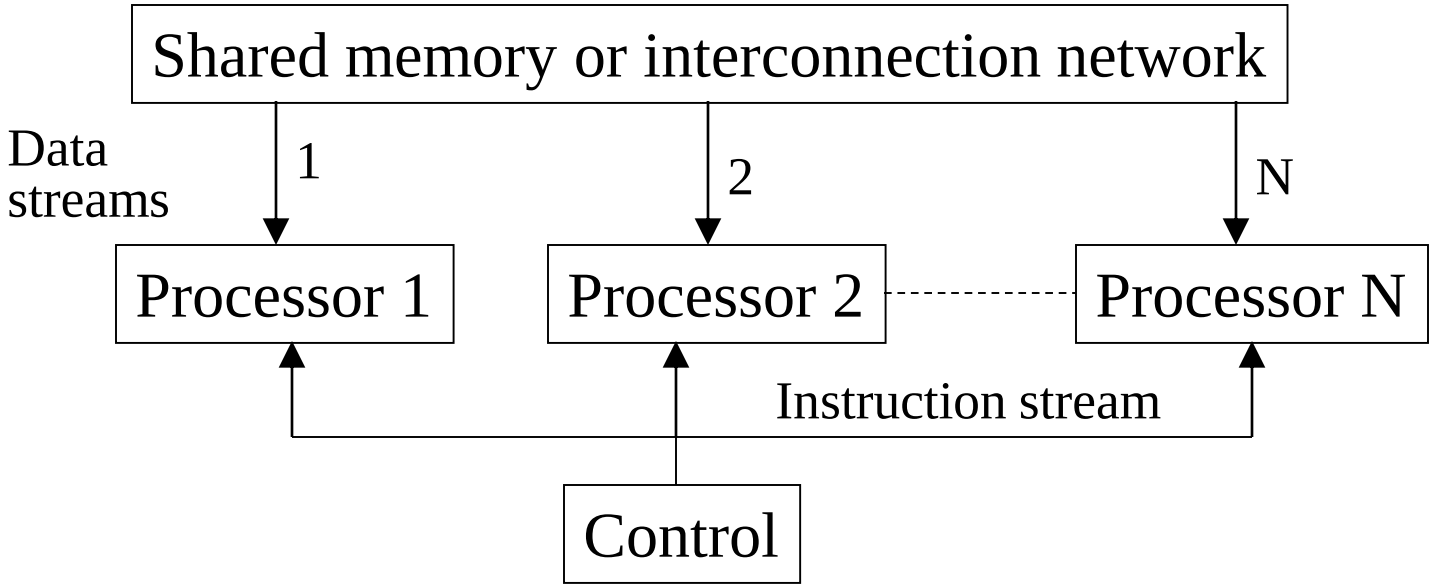
- There are N streams of instructions (algorithms/programs) and one stream of data. Parallelism is achieved by letting the processors do different things at the same time to the same data.
- MISD machines are useful in computations where the same input is to be subjected to several different operations.

MISD Example

- Checking whether a number Z is prime. A simple solution is to try all possible divisions of Z. Assume the number of processors is N=Z-2. All processors take Z as input and each tries to divide it by its associated divisor. So it is possible in one step to check if Z is prime. More realistically, if N<Z-2 then a subset of divisors is assigned to each processor.
- For most applications MISD computers are very awkward to use and no commercial machines exist with this design.

SIMD Computers

- All N identical processors operate under the control of a single instruction stream issued by a central control unit.
- There are N data streams, one per processor, so different data can be used in each processor.



Notes on SIMD Computers

- The processors operate *synchronously* and a global clock is used to ensure lockstep operation, i.e., at each step (global clock tick) all processors execute the same instruction, each on a different datum.
- Early array processors such as the ICL DAP, Connection Machine CM-200, and MasPar were SIMD computers (1980-1995).
- Later companies such as ClearSpeed marketed commercial SIMD products, for example, the CSX700 released in 2008 had 192 SIMD processors.

Notes on SIMD Computers

- SIMD machines are useful at exploiting data parallelism to solve problems having a regular structure in which the same instructions are applied to subsets of data, e.g., graphics processing.
- Advanced Vector Extensions (AVX): an SIMD instruction set extension to the x86 architecture from Intel, and introduced in 2008.
- Modern GPUs can be viewed as SIMD processors, resulting in the Single Instruction Multiple Threads (SIMT) execution model for GPUs.

SIMD Example

Problem: add two 2×2 matrices on 4 processors.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} + \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{pmatrix}$$

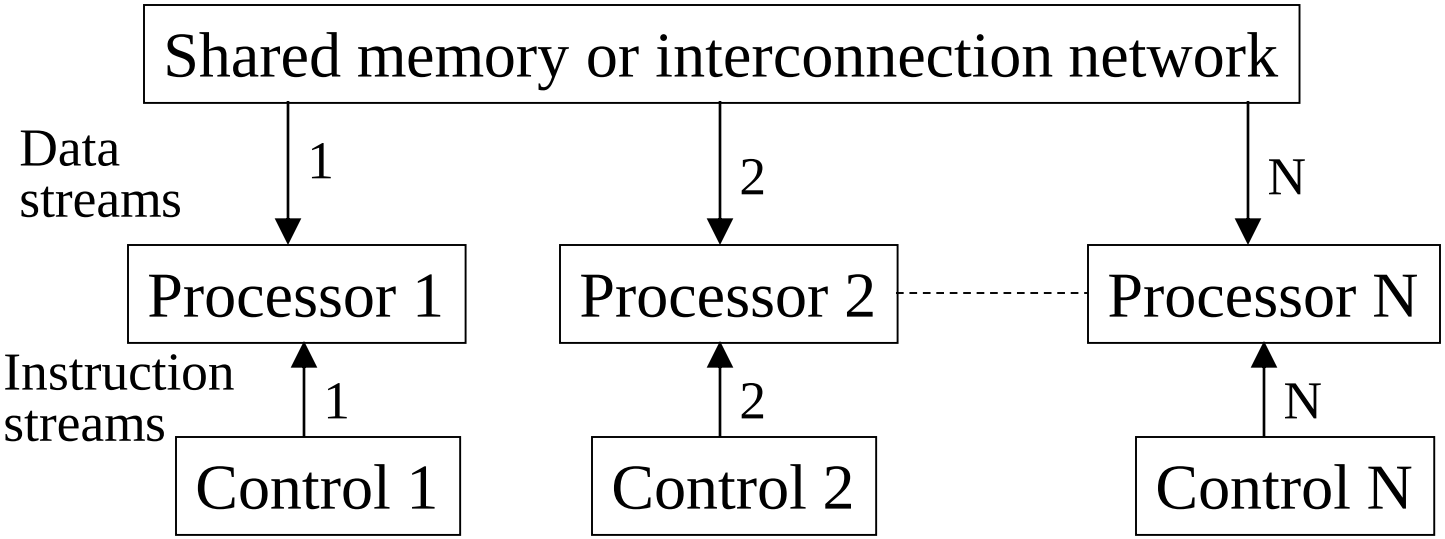
The same instruction is issued to all 4 processors (add two numbers), and all processors execute the instructions simultaneously. It takes one step to add the matrices, compared with 4 steps on a SISD machine.

Notes on SIMD Example

- In this example the instruction is simple, but in general it could be more complex such as merging two lists of numbers.
- The data may be simple (one number) or complex (several numbers).
- Sometimes it may be necessary to have only a subset of the processors execute an instruction, i.e., only some data needs to be operated on for that instruction. This information can be encoded in the instruction itself indicating whether
  - the processor is active (execute the instruction)
  - the processor is inactive (wait for the next instruction)

MIMD Computers

This is the most general and most powerful of our classification. We have N processors, N streams of instructions, and N streams of data.



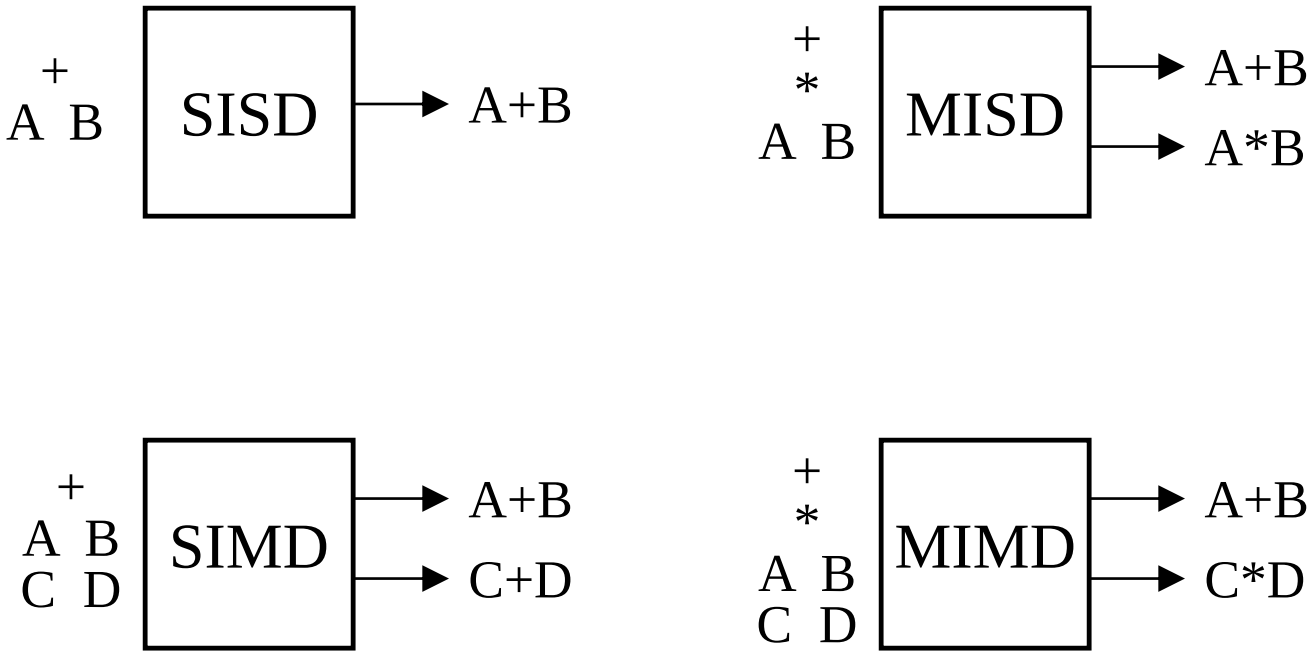
Notes on MIMD Computers

- The processors can operate asynchronously, i.e., they can do different things on different data at the same time.
- As with SIMD computers, communication of data or results between processors can be via shared memory or an interconnection network.

Notes on SIMD and MIMD

- In most problems to be solved on SIMD and MIMD computers it is useful for the processors to be able to communicate with each other to exchange data or results. This can be done in two ways
  - by using a shared memory and shared variables, or
  - using an interconnection network and message passing (distributed memory)
- MIMD computers with shared memory are known as *multiprocessors.* An example is the Onyx 300 produced by Silicon Graphics Inc.
- MIMD computers with an interconnection network are known as *multicomputers*. An example is the E6500 produced by Sun Microsystems.
- *Clusters* are multicomputers composed of off-the-shelf components

Potential of the 4 Classes



Single Program Multiple Data

- An MIMD computer is said to be running in SPMD mode if the same program is executing on each process.
- SPMD is not a hardware paradigm, so it is not included in our 4 classifications.
- It is a software paradigm for MIMD machines.
- Each processor executes an SPMD program on different data so it is possible that different branches are taken, leading to *asynchronous parallelism*. The processors no longer do the same thing (or nothing) in lockstep as they do on an SIMD machine. They execute different instructions within the same program.

SPMD Example

- Suppose X is 0 on processor 1, and 1 on processor 2. Consider

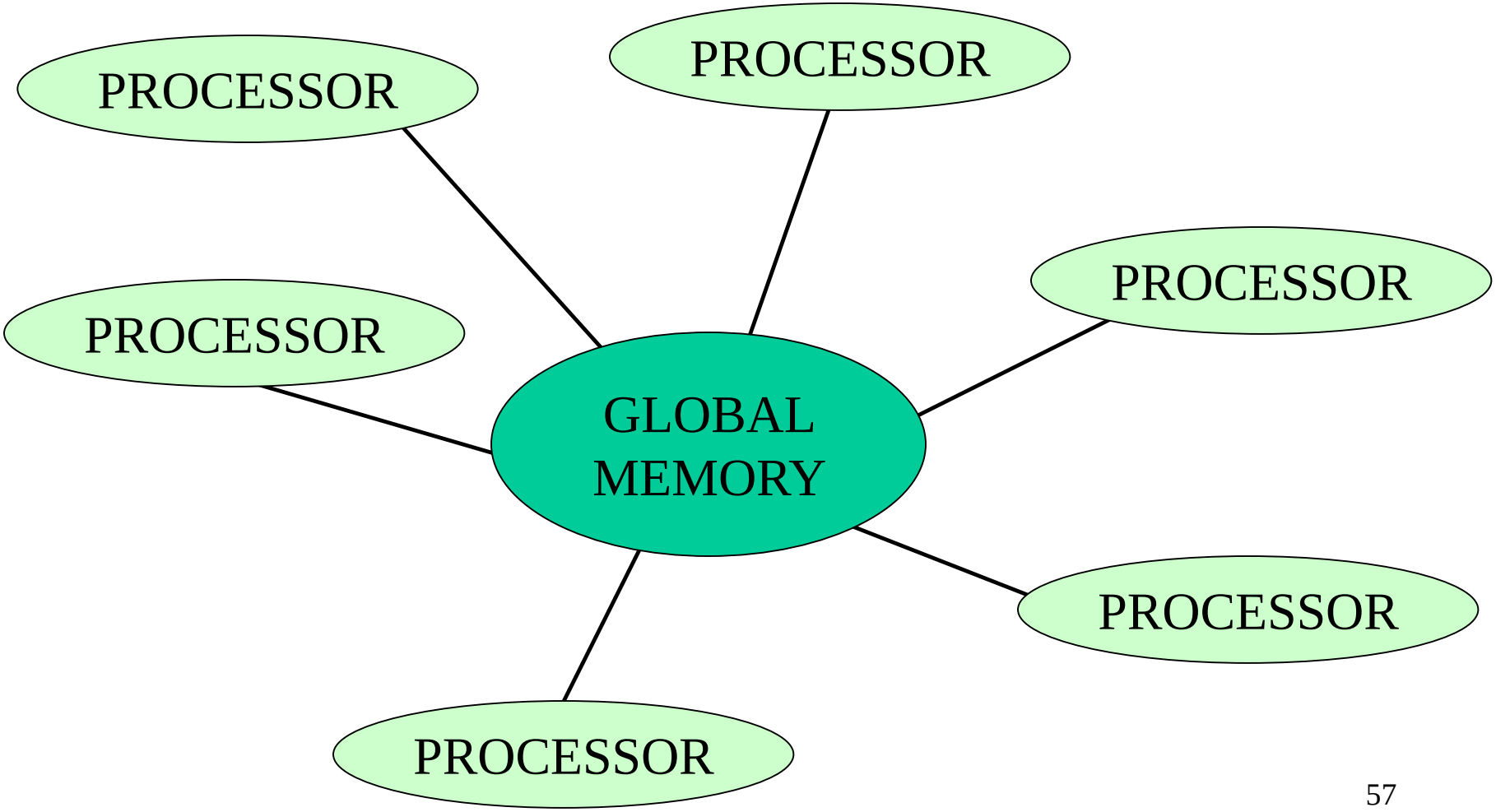
```
IF X = 0
   THEN S1
   ELSE S2
```

- Then processor 1 executes S1 *at the same time* that processor 2 executes S2.
- This could not happen on an SIMD machine.

Interprocessor Communication

- Usually a parallel program needs to have some means of sharing data and results processed by different processors. There are two main ways of doing this
  1. Shared Memory
  2. Message passing
- Shared memory consists of a global address space. All processors can read from and write into this global address space.

Global Shared Memory



Shared Memory Conflicts

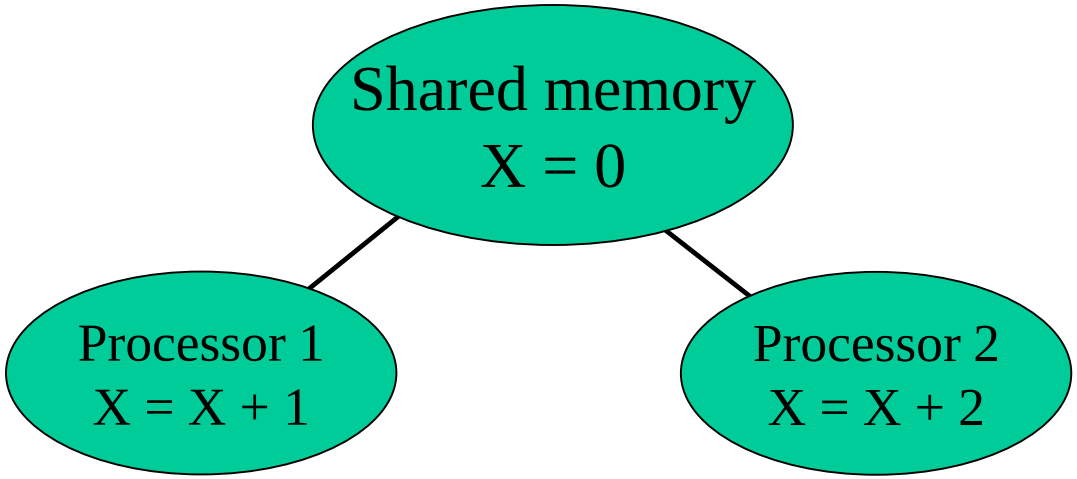
The shared memory approach is simple but can lead to problems when processors simultaneously access the same location in memory.

Example:

Suppose the shared memory initially holds a variable x with value 0. Processor 1 adds 1 to x and processor 2 adds 2 to x. What is the final value of x?

You should have met this problem before when studying locks and critical sections in the operating systems module.

Shared Memory Conflicts 2



This is an example of non-determinism or non-determinancy

The following outcomes are possible

1. If P1 executes and completes x=x+1 before P2 reads the value of x from memory then x is 3. Similarly, if P2 executes and completes x=x+2 before P1 reads the value of x from memory then x is 3.
2. If P1 or P2 reads x from memory before the other has written back its result, then the final value of x depends on which finishes last.
   - if P1 finishes last the value of x is 1
   - if P2 finishes last the value of x is 2

Non-Determinancy

- Non-determinancy is caused by *race conditions*.
- A race condition occurs when two statements in concurrent tasks access the same memory location, at least one of which is a write, and there is no guaranteed execution ordering between accesses.
- The problem of non-determinancy can be solved by synchronising the use of shared data. That is if x=x+1 and x=x+2 were mutually exclusive then the final value of x would always be 3.
- Portions of a parallel program that require synchronisation to avoid non-determinancy are called *critical sections*.

Locks and Mutual Exclusion

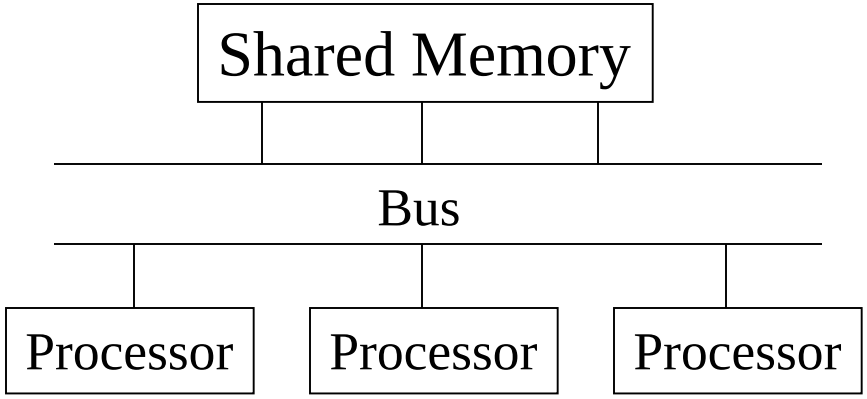
In shared memory programs *locks* can be used to give mutually exclusive access.

```
Processor 1:
        LOCK (X)
                X = X + 1
        UNLOCK (X)
```

```
Processor 2:
        LOCK (X)
                X = X + 2
        UNLOCK (X)
```

Limits on Shared Memory

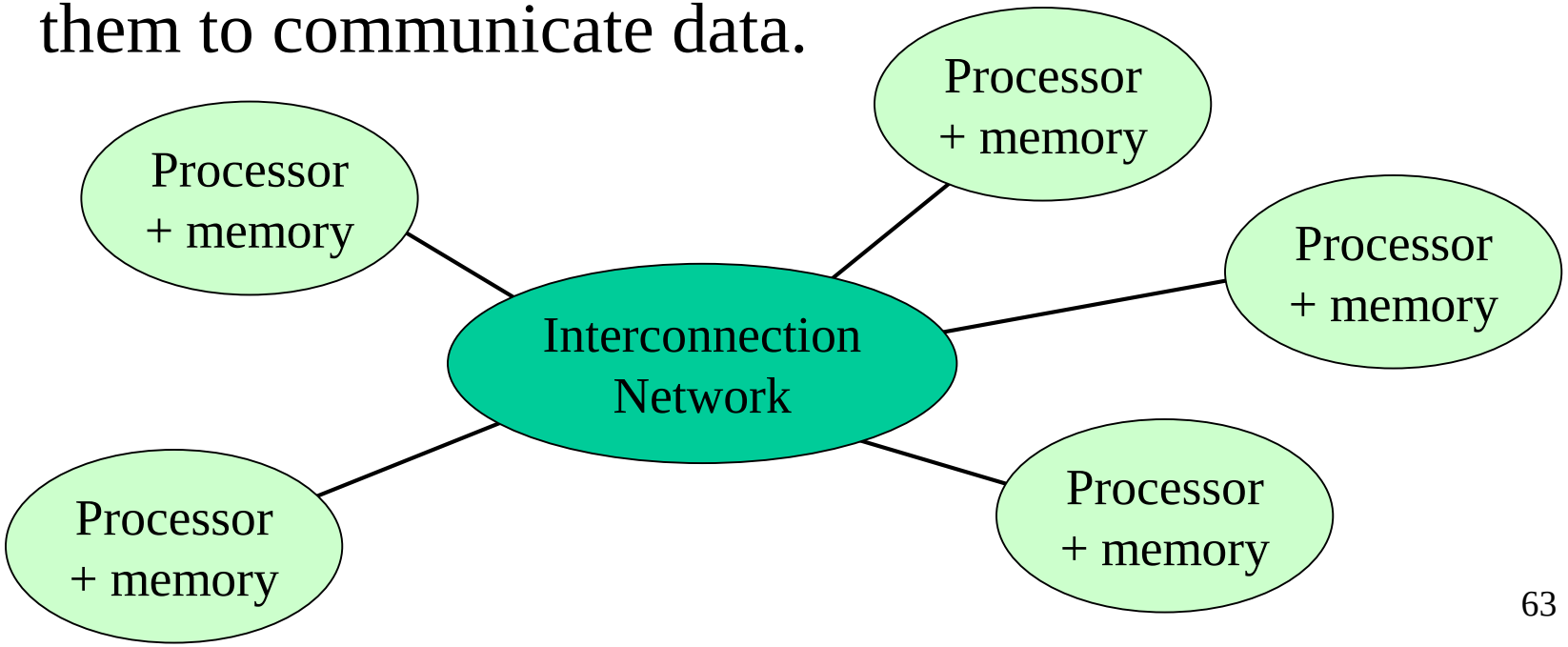
- Shared memory computers are often implemented by incorporating a fast bus to connect processors to memory.



- However, because the bus has a finite bandwidth, i.e., it can carry only a certain maximum amount of data at any one time, then as the number of processors increase the *contention* for the bus becomes a problem. So it is feasible to build shared memory machines with up to only about 100 processors.

Interconnection Networks and Message Passing

In this case each processor has its own private (local) memory and there is no global, shared memory. The processors need to be connected in some way to allow them to communicate data.



The Message Passing Paradigm

- Each process has its own address space.
- Processes cooperate to perform a task by independently computing with their local data, and communicating data between processes by explicitly exchanging messages.

Message Passing

- If a processor requires data contained on a different processor then it must be explicitly passed by using communication instructions, e.g., send and receive.

```
P1                          P2

receive (x, P2)     send (x, P1)
```

- The value x is explicitly passed from P2 to P1. This is known as *message passing*.

Communicating Sequential Processes

- CSP is a formal language used to described the interactions of concurrent processes.
- Described by Prof C. A. R. Hoare (University of Oxford) in 1978.
- CSP underpins the message passing paradigm.
- Inspired the development of the *Occam* language for parallel processing.
- Led to the *Transputer* family of processors manufactured by INMOS.

Basic CSP Concepts

- CSP program = parallel composition of a fixed number of sequential processes communicating with each other strictly through *synchronous message-passing*.
- Each process has an explicit name, and the source or destination of a message is defined by specifying the name of the sending or receiving process.

Introduction to Occam

- A coordination language based on CSP.
- Explicit message passing via channels
  - Uni-directional
  - Typed
  - Synchronous
- Assembly language of the transputer.
- Can check for deadlock and non-determinism.

The PAR Construct

- In Occam parallelism is expressed through the PAR construct:

```
PAR
    process 1
    process 2
    process 3
```

- This indicates that the 3 processes may be executed in parallel.
- Similarly, the SEQ construct is used to specify the sequential execution of a set of processes.

Simple Use of a Typed Channel

```
CHAN OF INT chan :
PAR
  chan ! 2
  INT x, z :
  SEQ
    chan ? x
    z := x
```

This defines the typed channel named chan

This is a primitive process that sends the number 2 on chan

This is a compound process

Receive a value on chan and store it in x

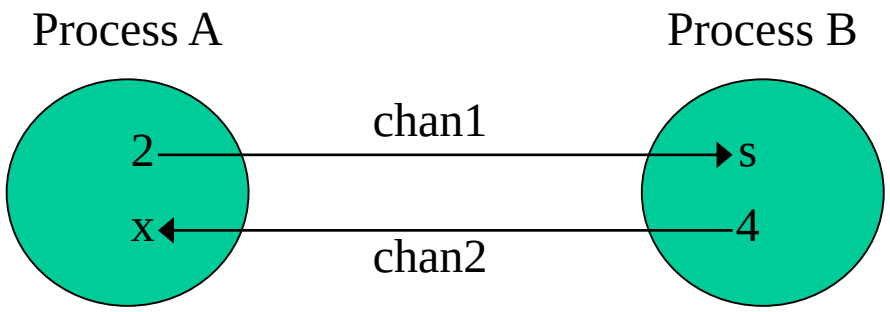
Set z equal to x

Deadlock

- Synchronous message passing can result in deadlock.

```
CHAN OF INT chan1, chan2 :   <-- This defines the typed channels chan1 and chan2
  PAR                        <-- This means processes defined below by SEQ run in parallel
    INT x :
    SEQ                      <-- This defines a sequential process
      chan1 ! 2              <-- This means 2 is sent on channel chan1
      chan2 ? x              <-- This means value received on channel chan2 is stored in x
    INT s :
    SEQ                      <-- This defines another sequential process
      chan2 ! 4
      chan1 ? s
```

Deadlock arises because Process A can't send 2 on chan1 until process B is ready to receive it into s. Similarly for chan2.



Non-Deterministic Communication

- This occurs if the receive on the receiving process can be matched by one of several potential send operations on other processes.
- Non-deterministic communication may be used to improve parallel performance by ensuring that the receiving process is supplied with enough data to keep it busy.
- However, non-determinism may be introduced into a program unintentionally, in which case it is a bug. It may cause intermittent failures.

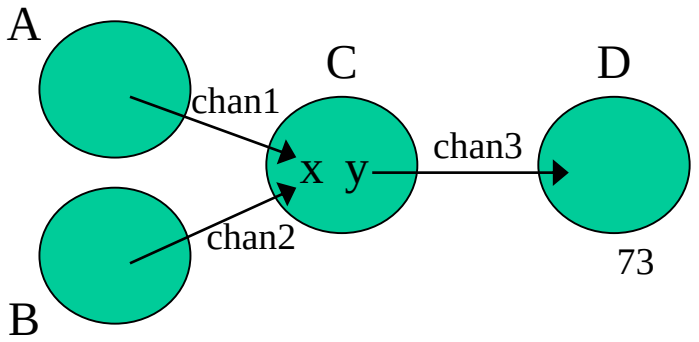
The ALT Construct

- In Occam non-deterministic communication is introduced using the ALT construct.

```
CHAN OF INT chan1, chan2, chan3 :   <-- This defines the typed channels chan1, chan2 and chan3
INT x, y :
ALT                                 <-- Monitor chan1 and chan2
  chan1 ? x                         <-- Waiting to receive input on chan1 into x
    SEQ                             <-- Process associated with chan1
      y := 2*x + 1                  <-- Compute y using x received on chan1
      chan3 ! y                     <-- Send y on chan3
  chan2 ? x                         <-- Waiting to receive input on chan2 into x
    SEQ                             <-- Process associated with chan2
      y := 2*x                      <-- Compute y using x received on chan2
      chan3 ! y                     <-- Send y on chan3
```

This only shows the code for process C

An ALT construct monitors several channels, each of which is associated with a process. Only the channel associated with the channel that first produces an input will be executed.



Occam Replicators

- Occam also has IF and WHILE constructs for controlling program flow.
- SEQ, PAR, ALT, and IF constructs can all be replicated. For example to get multiple similar sequential processes:

```
CHAN OF INT chan1 :
INT x :
VAL n IS 100 :
SEQ i = 0 FOR n-1
  chan1 ? x
```

Occam Replicators

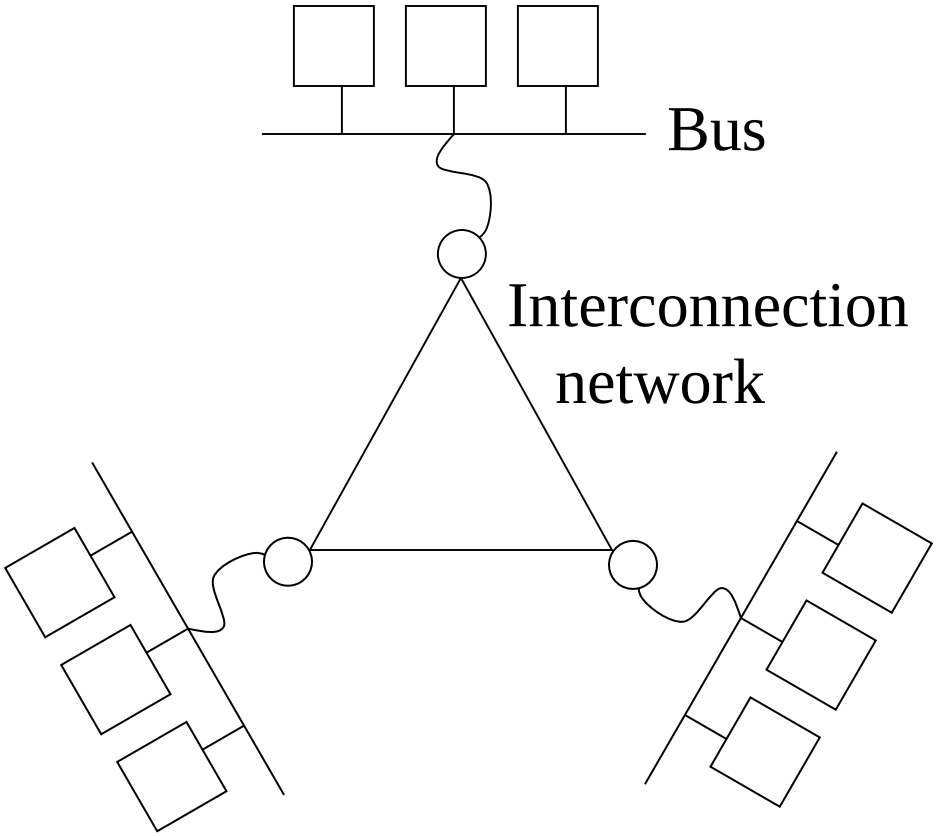
- Replicated PAR creates array of similar processes – used for SPMD style parallelism.
- Replicated ALT can be used to monitor array of channels and act according to which one receives input first.
- Replicated IF can be used to create conditional with similar choices.

Main Points of Occam

- Primitive input, output, and assignment processes.
- Basic constructs:
  - SEQ for sequential processes
  - PAR for parallel processes
  - ALT for non-determinism
  - IF conditional construct
  - WHILE loop construct
- Synchronous communication by typed channels
  - Single datatypes,
  - Can also handle mixed datatypes, fixed and variable length arrays,
- Replicators for SEQ, PAR, ALT, and IF.

Hybrid Computers

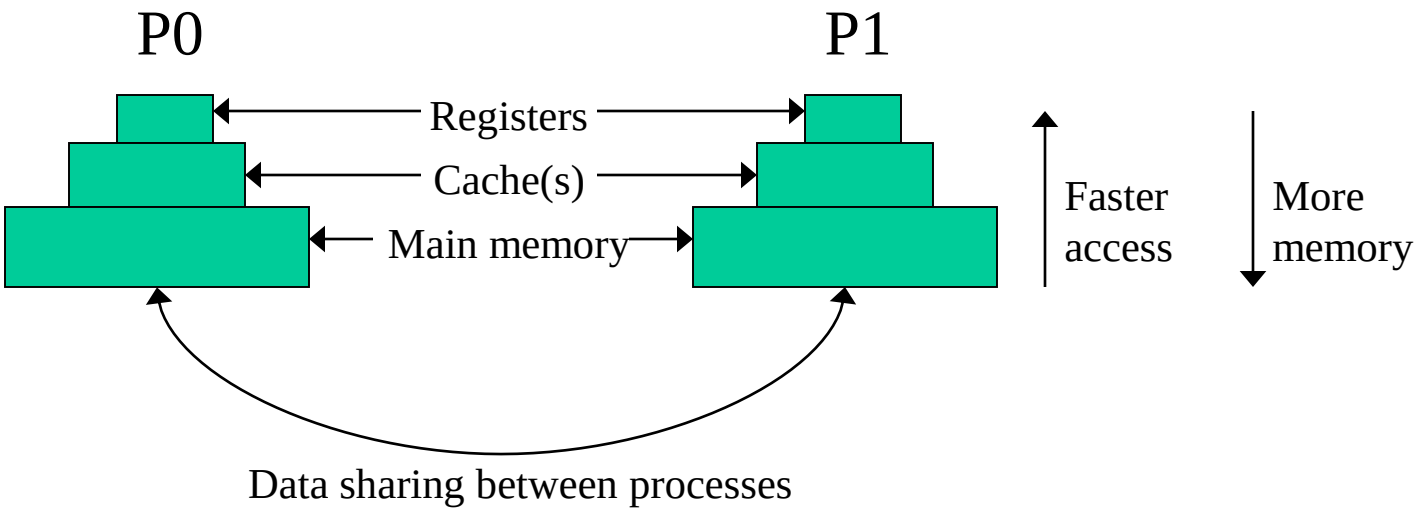
- In addition to the cases of shared memory and distributed memory there are possibilities for hybrid designs that incorporate features of both.
- Clusters of processors are connected via a high speed bus for communication within a cluster, and communicate between clusters via an interconnection network.



Comparison of Shared and Distributed Memory

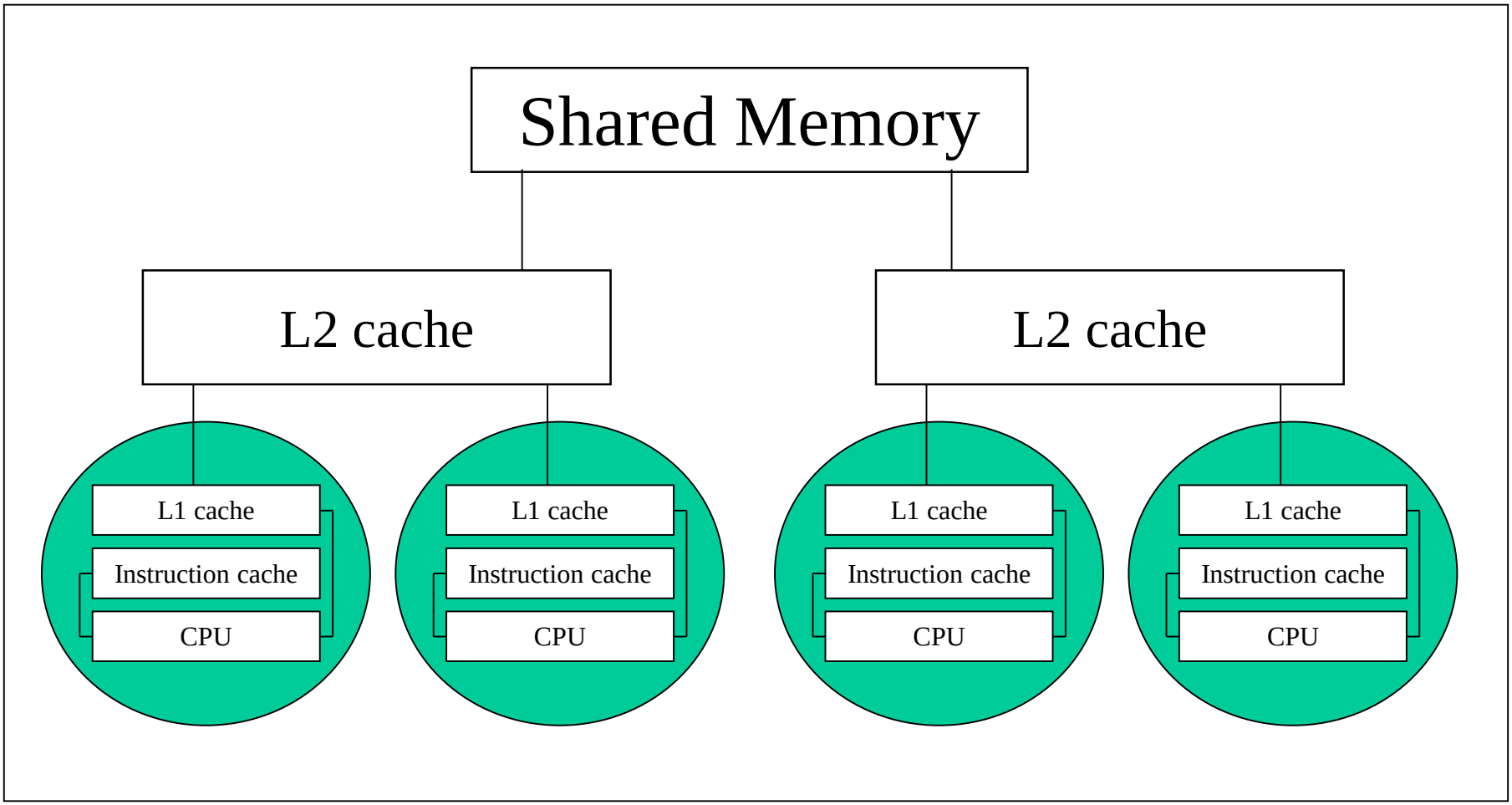
Distributed memory	Shared memory
Large number of processors (100's to 1000's, or more)	Moderate number of processor (10's to 100)
High peak performance	Modest peak performance
Unlimited expansion	Limited expansion
Difficult to fully utilise	Relatively easy to fully utilise
Revolutionary parallel computing	Evolutionary parallel computing

Memory Hierarchy 1



- In general, certain memory locations have a greater affinity for certain processes.
- Parallel programming language may make distinction between “near” and “far” memory, but will not fully represent the memory hierarchy.

Typical Quad-Core Chip



Summing m Numbers

Example: summing m numbers

On a sequential computer we have,

```
sum = a[0];
for (i=1;i<m;i++) {
        sum = sum + a[i];
}
```

We would expect the running time be roughly proportional to m. We say that the running time is $\Theta(m)$.

Summing m Numbers in Parallel

- What if we have N processors, with each calculating the m/N numbers assigned to it?
- We must add these partial sums together to get the total sum.

Summing Using Shared Memory

The m numbers, and the global sum, are held in global shared memory.

```
global_sum = 0;
for (each processor){
        local_sum = 0;
        calculate local sum of m/N numbers
        LOCK
                global_sum = global_sum + local_sum;
        UNLOCK
}
```

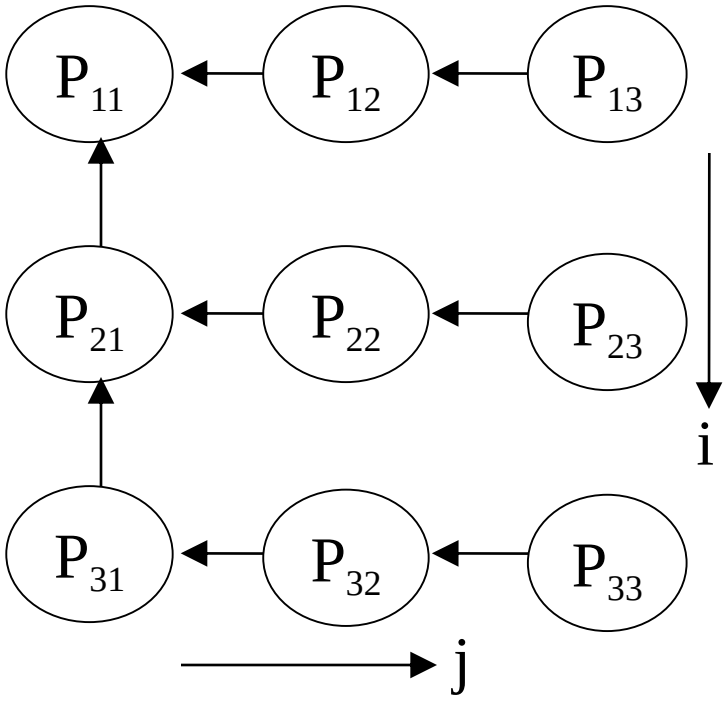
Notes on Shared Memory Algorithm

- Since global_sum is a shared variable each processor must have mutually exclusive access to it – otherwise the final answer may be incorrect.
- The running time (or *algorithm time complexity*) is

  $\Theta(m/N) + \Theta(N)$
- where
  - m/N comes from finding the local sums in parallel
  - N comes from adding N numbers in sequence

Summing Using Distributed Memory

Suppose we have a square mesh of N processors.



The algorithm is as follows:

1. Each processor finds the local sum of its m/N numbers
2. Each processor passes its local sum to another processor in a coordinated way
3. The global sum is finally in processor P_{11}.

Distributed Memory Algorithm

The algorithm proceeds as follows:

1. Each processor finds its local sum.
2. Sum along rows:
   a) If the processor is in the rightmost column it sends its local sum to the left.
   b) If the processor is not in the rightmost or leftmost column it receives the number form the processor on its right, adds it to its local sum, and sends the result to the processor to the left.
   c) If the processor is in the leftmost column it receives the number from the processor on its right and adds it to its local sum to give the row sum.
3. Leftmost column only – sum up the leftmost column:
   a) If the processor is in the last row send the row sum to the processor above
   b) If the processor is not in the last or first row receive the number from the processor below, add it to the row sum, and send result to processor above
   c) If the processor is in the first row receive the number from the processor below and add it to the row sum. This is the global sum.

Summing Example

There are $\sqrt{N}-1$ additions and $\sqrt{N}-1$ communications in each direction, so the total time complexity is

$$\Theta(m/N) + \Theta(\sqrt{N}) + C$$

where C is the time spent communicating.

10	12	7
6	9	17
9	11	18

Initially

10	19	
6	26	
9	29	

Shift left and sum

29		
32		
38		

Shift left and sum

29		
70		

Shift up and sum

99		

Shift up and sum