# COMS3008A: Parallel Computing

Hairong Bau

School of Computer Science & Applied Mathematics

Semester two 2022

WITS UNIVERSITY

## Contents

## Outline

## Objectives

- Understand the basics of parallel computers, parallel computing, the motivation of parallel computing, and the classification of parallel computers.
- Understand and apply the simple quantitative modelling for parallel program performance.

## Outline

## Course admin

- Relevant course information is given in course outline (uploaded on ulwazi course site).
- Course related communications will be announced primarily through the announcement via ulwazi course site. It is important for you to check such announcements regularly to keep informed timely.

## Outline

## Outline

## Parallel computing

- Parallel Computer: A parallel computer is a computer system that uses multiple processing elements simultaneously in a cooperative manner to solve a computational problem.
- Parallel computing (or processing): Parallel processing includes techniques and technologies that make it possible to compute in parallel.
  - Hardware, networks, operating systems, parallel libraries, languages, compilers, algorithms, tools etc.
- Parallel computing is an evolution of serial computing.
  - Parallelism is natural.
  - Computing problems differ in level or type of parallelism.

## Serial computing

- A problem is broken into a discrete series of instructions;
- Instructions are executed sequentially one after another;
- Executed on a single processor;
- Only one instruction may execute at any moment in time.

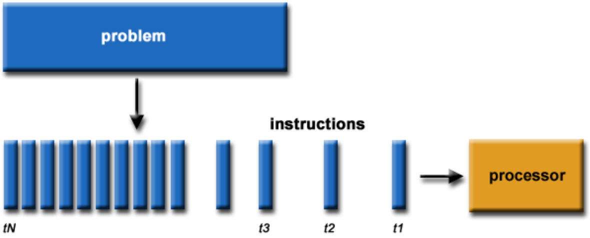


Figure: Serial computing

## Parallel computing

- A problem is broken into discrete parts that can be solved concurrently;
- Each part is further broken down to a series of instructions;
- Instructions from each part execute simultaneously on different processors;
- An overall control/coordination mechanism is employed.

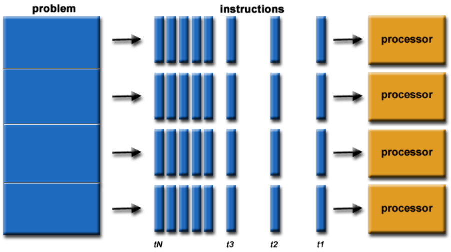


Figure: Parallel computing

## Parallel computing cont.

An example of parallelizing an addition of two vectors is shown in Figure 3.

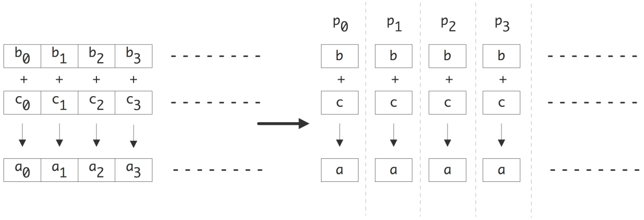
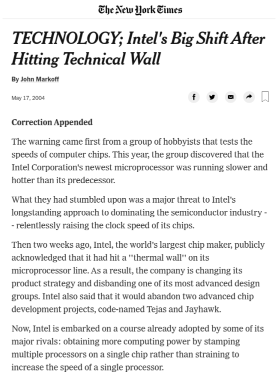


Figure: Parallelization of a vector addition

## Outline

## Why parallelism?

- In 2004, Intel changed its course from traditional chip design approach (single core processor) to embrace “dual core” processor structure.



The New York Times

*TECHNOLOGY; Intel's Big Shift After Hitting Technical Wall*

By John Markoff

May 17, 2004

**Correction Appended**

The warning came first from a group of hobbyists that tests the speeds of computer chips. This year, the group discovered that the Intel Corporation's newest microprocessor was running slower and hotter than its predecessor.

What they had stumbled upon was a major threat to Intel's longstanding approach to dominating the semiconductor industry -- relentlessly raising the clock speed of its chips.

Then two weeks ago, Intel, the world's largest chip maker, publicly acknowledged that it had hit a ''thermal wall'' on its microprocessor line. As a result, the company is changing its product strategy and disbanding one of its most advanced design groups. Intel also said that it would abandon two advanced chip development projects, code-named Tejas and Jayhawk.

Now, Intel is embarked on a course already adopted by some of its major rivals: obtaining more computing power by stamping multiple processors on a single chip rather than straining to increase the speed of a single processor.

Figure: Intel’s shift in chip design to multi-core structure.

## A brief history of processor performance

- What had been happening before multi-processor? – Single-processor machines: they had been getting exponentially faster.

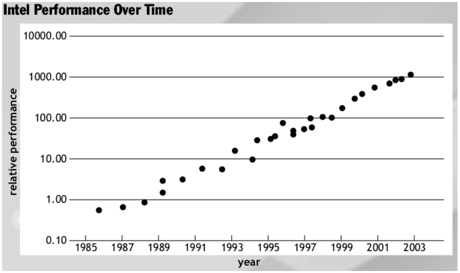


Figure: Intel single processor performance over time

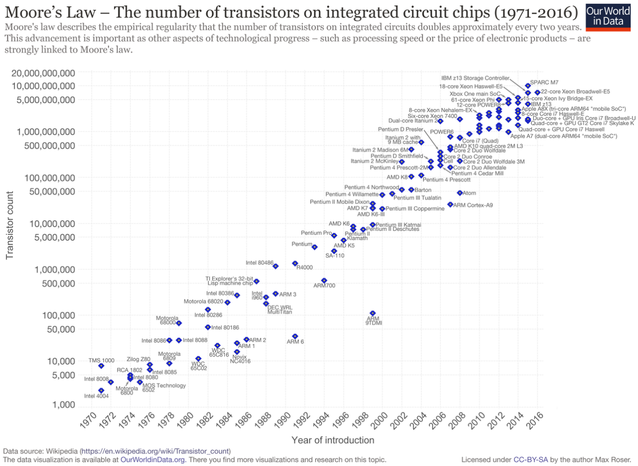
## A brief history of processor performance cont.

- Wider data paths
  - 4 bit $\rightarrow$ 8 bit $\rightarrow$ 16 bit $\rightarrow$ 32 bit $\rightarrow$ 64 bit
- More efficient pipelining
  - For example, from 3.5 cycles per instruction (CPI) $\rightarrow$ 1.1 CPI
- Exploiting instruction level parallelism (ILP)
  - " Superscale" processing: e.g., issues up to 4 instructions/cycle
- Faster clock rates
  - e.g., 10MHz $\rightarrow$ 100MHz $\rightarrow$ 1GHz $\rightarrow$ 3GHz

During 80s and 90s, computers had large exponential performance gains

# A brief history of processor performance cont.

- “The number of transistors on an integrated circuit doubles every two years.” — Gordon E. Moore



Moore’s Law – The number of transistors on integrated circuit chips (1971-2016)

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important as other aspects of technological progress – such as processing speed or the price of electronic products – are strongly linked to Moore's law.

Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Figure: Moore’s Law – transistor count 1971 - 2016

# A brief history of processor performance cont.

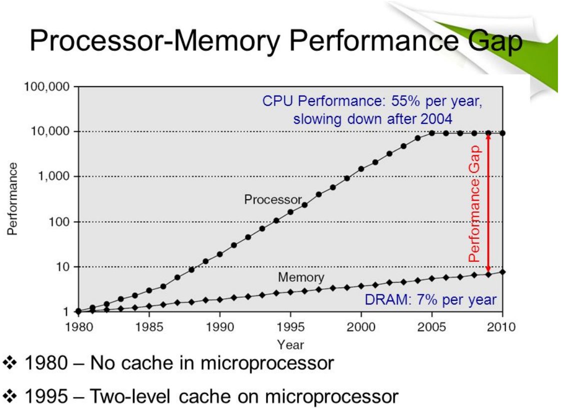


Figure: Processor-memory performance gap

## A brief history of processor performance cont.

- For example, Intel Itanium II
    - 6-way integer unit $<$ 2% die area;
    - Cache logic $>$ 50% die area.
- Most of chip there to keep these 6 integer units at ‘peak’ rate.
- Main issue is external DRAM latency (50ns) to internal clock rate (0.25ns) ratio is 200:1.

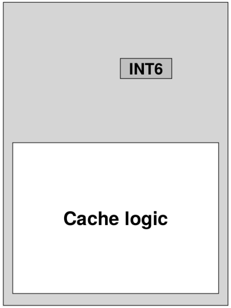


Figure: Illustration of the die area for integer unit and cache logic for Intel Itanium II.

## A brief history of parallel computing

- Greater clock frequency, greater electrical power
- Intel VP Patrick Gelsinger (ISSCC 2001): “If scaling continues at present pace, by 2005, high speed processors would have power density of nuclear reactor, by 2010, a rocket nozzle, and by 2015, surface of sun.”

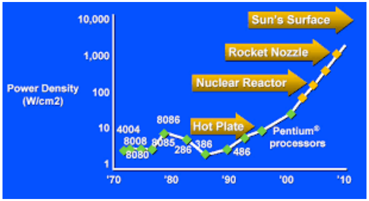
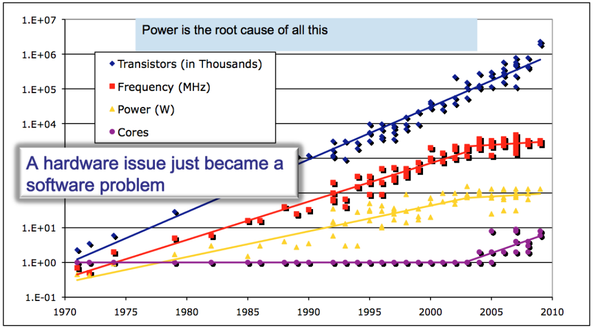


Figure: Intel VP Patrick Gelsinger (ISSCC 2001)

# A brief history of parallel computing cont.

- Add multiple cores to add performance, keep clock frequency the same or reduced.



Data from Kunle Olukotun, Lance Hammond, Herb Sutter, Burton Smith, Chris Batten, and Krste Asanoviç
Slide from Kathy Yelick

Figure: Growth of transistors, frequency, cores, and power consumption

## A brief history of parallel computing cont.

- What we can do? We can still pack more and more transistors on to a die, but we cannot make the individual transistor faster and faster.
- We have to make better use of those more and more transistors to increase the performance instead of making only the clock speed faster. One solution is to use parallelism.

WITS
UNIVERSITY

## Why parallelism – summary

- Serial machines have inherent limitations:
  - Processor speed, memory bottlenecks, ...
- Parallelism has become the future of computing.
- Two primary benefits of parallel computing:
  - Solve fixed size problem in shorter time
  - Solve larger problems in a fixed time
- Other factors motivate parallel processing:
  - Effective use of machine resources;
  - Cost efficiencies;
  - Overcoming memory constraints.
- Performance is still the driving concern.
- Technology push
- Application pull
  - Application performance demands hardware advances;
  - Hardware advances generate new applications;
  - New applications have greater performance demands.

WITS UNIVERSITY

## Outline

WITS UNIVERSITY

## No 1 Supercomputer in 6/2022 — Frontier

| | |
|---|---|
| Site | DOE/SC/Oak Ridge National Laboratory |
| Manufacturer | HPE (Frontier; Frontier URL) |
| Cores | 8,730,112 |
| Processor | AMD Optimized 3rd Generation EPYC 64C 2GHz |
| Interconnect | Slingshot-11 |
| Linpack Performance (Rmax) | 1,102.00 PFlop/s |
| Theoretical Peak (Rpeak) | 1,685.65 PFlop/s |
| Nmax | 24,440,832 |
| Power | 21,100.00 kW (Submitted) |
| Operating System | HPE Cray OS |

## No 1 Supercomputer in 11/2020 & 11/2021 — Fugaku

| | |
|---|---|
| Site | RIKEN Center for Computational Science |
| Manufacturer | Fujitsu (Fugaku; Fugaku virtual tour) |
| Cores | 7,630,848 |
| Linpack Performance (Rmax) | 442,010 TFlop/s |
| Theoretical Peak (Rpeak) | 537,212 TFlop/s |
| HPCG | 16,004.5 TFlop/s |
| Nmax | 21,288,960 |
| Power | 29,899.23 kW (Submitted) |
| Memory | 5,087,232 GB |
| Processor | A64FX 48C 2.2GHz |
| Interconnect | Tofu interconnect D |
| Operating System | RHEL |
| Compiler | Fujitsu software technical computing suite v4.0 |
| Math library | Fujitsu software technical computing suite v4.0 |
| MPI | Fujitsu software technical computing suite v4.0 |

## No 1 Supercomputer in 11/2020 & 11/2021 — Fugaku cont.

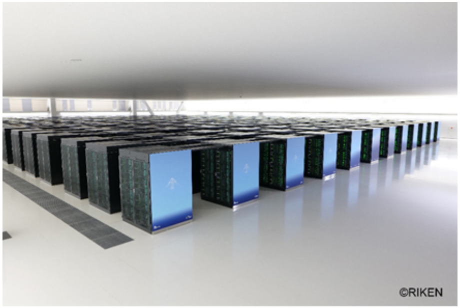


Figure: Fugaku, a supercomputer from RIKEN and Fujitsu Limited.

# No 1 Supercomputer in 11/2019 — Summit

| | |
|---|---|
| Site | DOE/SC/Oak Ridge National Laboratory |
| Manufacturer | IBM |
| Cores | 2,414,592 |
| Linpack Performance (Rmax) | 148,600 TFlop/s |
| Theoretical Peak (Rpeak) | 200,795 TFlop/s |
| HPCG | 2,925.75 TFlop/s |
| Nmax | 16,693,248 |
| Power | 9,783.00 kW (Submitted) |
| Memory | 2,801,664 GB |
| Processor | IBM POWER9 AC9 22C 3.07GHz |
| GPU | Nvidia Volta GV100 |
| Interconnect | Dual-rail Mellanox EDR Infiniband |
| Operating System | RHEL 7.4 |
| Compiler | IBM XLC, nvcc |
| Math library | ESSL, CUBLAS 9.2 |
| MPI | Spectrum MPI |

## No 1 Supercomputer in 11/2019 — Summit cont.

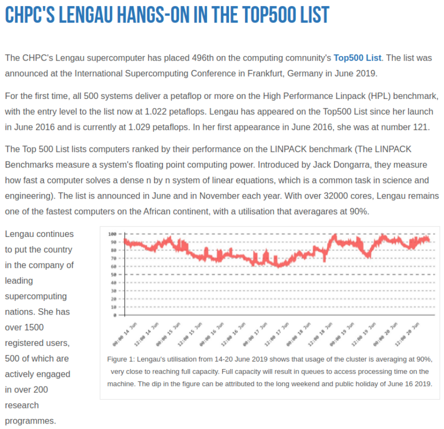
Figure: Summit, a supercomputer from IBM and the US Department of Energy's Oak Ridge National Laboratory (ORNL).

## Lengau from CHPC SA — Rank 400 (11/2018)

| | |
|---|---|
| Site | Centre for High Performance Computing |
| System URL | `http://www.chpc.ac.za/` |
| Manufacturer | Dell |
| Cores | 32,856 |
| Linpack Performance (Rmax) | 1,029.3 TFlop/s |
| Theoretical Peak (Rpeak) | 1,366.8 TFlop/s |
| Nmax | 3,105,408 |
| Power | 685 kW (Submitted) |
| Memory | 175,232 GB |
| Processor | Xeon E5-2690v3 12C 2.6GHz |
| Interconnect | Infiniband FDR |
| Operating System | CentOS |
| Compiler | Intel |
| Math Library | Intel MKL |
| MPI | Intel MPI |

The same machine was ranked 161 and 165 in 11/2016 and 11/2017, respectively.

# Lengau from CHPC SA



CHPC'S LENGAU HANGS-ON IN THE TOP500 LIST

The CHPC's Lengau supercomputer has placed 496th on the computing community's **Top500 List**. The list was announced at the International Supercomputing Conference in Frankfurt, Germany in June 2019.

For the first time, all 500 systems deliver a petaflop or more on the High Performance Linpack (HPL) benchmark, with the entry level to the list now at 1.022 petaflops. Lengau has appeared on the Top500 List since her launch in June 2016 and is currently at 1.029 petaflops. In her first appearance in June 2016, she was at number 121.

The Top 500 List lists computers ranked by their performance on the LINPACK benchmark (The LINPACK Benchmarks measure a system's floating point computing power. Introduced by Jack Dongarra, they measure how fast a computer solves a dense n by n system of linear equations, which is a common task in science and engineering). The list is announced in June and in November each year. With over 32000 cores, Lengau remains one of the fastest computers on the African continent, with a utilisation that averagares at 90%.

Lengau continues to put the country in the company of leading supercomputing nations. She has over 1500 registered users, 500 of which are actively engaged in over 200 research programmes.

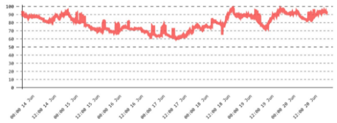


Figure 1: Lengau's utilisation from 14-20 June 2019 shows that usage of the cluster is averaging at 90%, very close to reaching full capacity. Full capacity will result in queues to access processing time on the machine. The dip in the figure can be attributed to the long weekend and public holiday of June 16 2019.

Figure: Lengau – CHPC

## The Clusters at Mathematical Sciences Lab

- stampede: Up to sixteen nodes, each with two Xeon E5-2680 CPUs, two GTX1060 GPUs (6GB per GPU, 12GB per node), and 32GB of system AM. For general purpose use or jobs that can leverage InfiniBand, however, InfiniBand has not yet been enabled yet.
- batch: Up to sixteen nodes each with a single Intel Core i9-10940X CPU (14 cores), NVIDIA RTX3090 GPU (24GB), and 128GB of system RAM. For bigger jobs that can leverage a bigger GPU and additional system RAM.
- biggpu: Each node has two Intel Xeon Platinum 8280L CPUs (28 cores per CPU, 56 cores per node) with two NVIDIA Quadro RTX8000 GPUs (48GB per GPU, 96GB per node), and 1TB of system RAM. For mature code that can meaningfully leverage large amounts of GPU and system RAM. While the system will allow jobs to use all three nodes, please try to use only a single node.

## The Clusters at Mathematical Sciences Lab cont.

- ranger: Up to twelve nodes, each with two AMD CPUs, 32GB of RAM, and InfiniBand. No GPUs! For large CPU based jobs. Partition only available later in 2021.
- mia: For the exclusive use of researchers in MIA. Up to twelve nodes with InfiniBand.

## Outline

## Outline

## Flynn’s taxonomy

Flynn’s taxonomy is widely used since 1966 for classification of parallel computers. The classification is based on two independent dimensions of *instruction stream* and *data stream* with two possible states: *single* or *multiple*.

- SISD: Single instruction stream single data stream. This is the traditional CPU architecture: at any one time only a single instruction is executed, operating on a single data item.

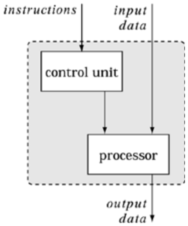


Figure: The SISD architecture

## SIMD

- SIMD: Single instruction stream multiple data stream. In this computer type there can be multiple processors, each operating on its own data item, but they are all executing the same instruction on that data item. SIMD computers excel at operations on arrays, such as

$$for(i = 0; i < N; i++) \quad a[i] = b[i] + c[i];$$

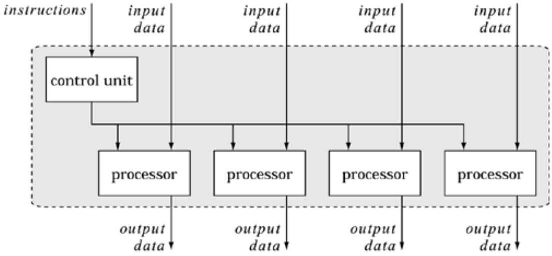


Figure: The SIMD architecture

## MIMD

- MISD: Multiple instruction stream single data stream. Each processing unit executes different instruction streams on a single data stream. Very few computers are in this type.
- MIMD: Multiple instruction stream multiple data stream. Multiple processors operate on multiple data items, each executing independent, possibly different instructions. Most current parallel computers are of this type.

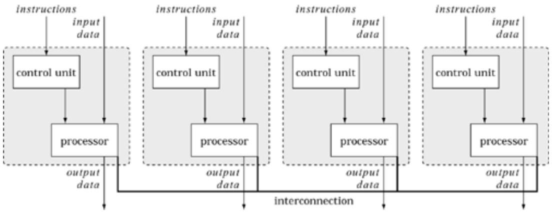


Figure: The MIMD architecture

## MIMD cont.

- Most of MIMD machines operate in *single program multiple data* (SPMD) mode, where the programmers starts up the same executable on the parallel processors.

## A Further Decomposition of MIMD

The MIMD category is typically further decomposed according to memory organization: shared memory and distributed memory.

- **Shared memory:** In a shared memory system, all processes share a single address space and communicate with each other by writing and reading shared variables.
- One class of shared-memory systems is called SMPs(symmetric multiprocessors).

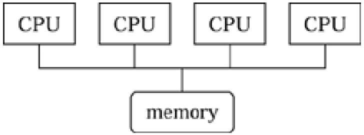


Figure: The SMP architecture

## NUMA

- The other main class of shared-memory systems is called non-uniform memory access (NUMA). The memory is shared, it is uniformly addressable from all processors, but some blocks of memory may be physically more closely associated with some processors than others.
- To mitigate the effects of non-uniform access, each processor has a cache, along with a protocol to keep cache entries coherent — cache-coherent non-uniform memory access systems (ccNUMA).

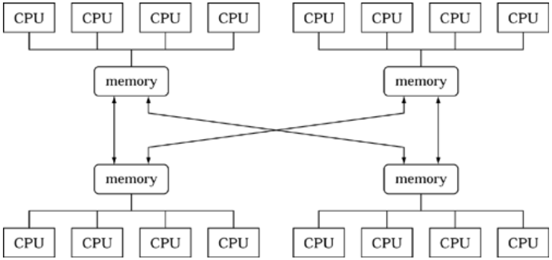


Figure: The NUMA architecture

## Distributed memory systems

- Each process has its own address space and communicates with other processes by message passing (sending and receiving messages).

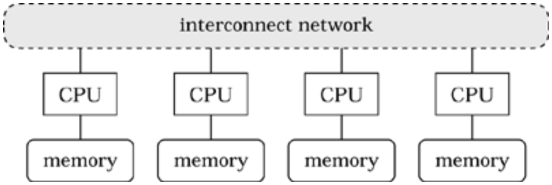


Figure: The distributed memory architecture

## Clusters

- Clusters are distributed memory systems composed of off-the-shelf computers connected by an off-the-shelf network.

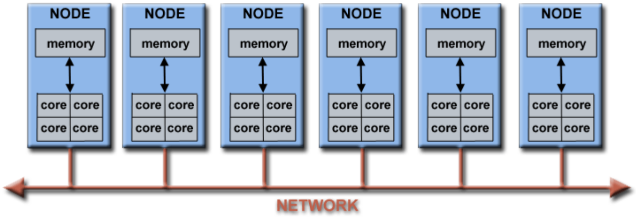


Figure: A cluster

## Outline

WITS UNIVERSITY

## A simple performance modelling

- Consider a computation consisting of three parts: a setup section, a computation section, and a finalization section.
- The total running time of this program on one processing element (PE) is given as:

$$T_{total}(1) = T_{setup} + T_{compute} + T_{finalization} \tag{1}$$

- What happens when we run this computation on a parallel computer with multiple PEs?

$$T_{total}(P) = T_{setup} + \frac{T_{compute}(1)}{P} + T_{finalization} \tag{2}$$

## Speedup

- An important measure of how much additional PEs help is the relative speedup S, which describes how much faster a problem runs:

$$S(P) = \frac{T_{total}(1)}{T_{total}(P)} \tag{3}$$

- A related measure is the efficiency E, which is the speedup normalized by the number of PEs.

$$E(P) = \frac{S(P)}{P} = \frac{T_{total}(1)}{PT_{total}(P)} \tag{4}$$

- Ideally, we would want the speedup to be equal to P, the number of PEs. This is sometimes called perfect linear speedup.

## Amdahl's Law

- The terms that cannot be run concurrently are called the serial terms. Their running times represent some fraction of the total, called the serial fraction, denoted $\gamma$.

$$\gamma = \frac{T_{setup} + T_{finalization}}{T_{total}(1)} \tag{5}$$

- The fraction of time for parallelizable part is then $1 - \gamma$. The total computation time with P PEs becomes

$$T_{total}(P) = \gamma T_{total}(1) + \frac{(1-\gamma)T_{total}(1)}{P} \tag{6}$$

## Amdahl's Law cont.

- Then we obtain the well-known Amdahl's law:

$$S(P) = \frac{T_{total}(1)}{T_{total}(P)} = \frac{T_{total}(1)}{(\gamma + \frac{1-\gamma}{P})T_{total}(1)} = \frac{1}{\gamma + \frac{1-\gamma}{P}} \tag{7}$$

  Taking the limit as P goes to infinity in Eq. 7,

$$S(P) = \frac{1}{\gamma} \tag{8}$$

  Eq. 8 gives an upper bound on the speedup.

- Amdahl's Law says the speedup you can achieve is limited by the fraction for serial computation in your problem (i.e., the number of PEs does not determine the upper bound of speedup you can achieve).

## Amdahl's Law cont.

### Example 1

Suppose we are able to parallelize 90% of a serial program. Further suppose the speedup of this part is $P$, the number of processes we used (which is a perfect linear speedup). If the serial time is $T_{serial} = 20$s, then the runtime of the parallelized part is $(0.9 \times T_{serial})/P = 18/P$. The runtime of unparallelized part is $0.1 \times 20 = 2$s. The overall parallel runtime will be

$$T_{parallel} = 18/P + 2,$$

and the speedup will be

$$S = \frac{T_{serial}}{T_{parallel}} = \frac{20}{18/p + 2}.$$

As $P$ gets larger, $18/P$ gets close to 0, and the denominator gets close to 2, in turn, $S$ gets close to 10. That means $S \leq 10$, no matter how many number of processes you use in your program.

## Gustafson's Law

- In contrast to Eq. 2, we obtain $T_{total}(1)$ from the serial and parallel parts when executed on P PEs.

$$T_{total}(1) = T_{setup} + PT_{compute}(P) + T_{finalization} \tag{9}$$

- Now, we define the so-called scaled serial fraction, denoted $\gamma_{scaled}$, as

$$\gamma_{scaled} = \frac{T_{setup} + T_{finalization}}{T_{total}(P)}, \tag{10}$$

and then

$$T_{total}(1) = \gamma_{scaled} T_{total}(P) + P(1 - \gamma_{scaled}) T_{total}(P). \tag{11}$$

## Gustafson’s Law cont.

- Using Eq. 11, we obtain the scaled speedup, sometimes known as Gustafson’s law.

$$\begin{aligned} S(P) &= \frac{T_{total}(1)}{T_{total}(P)} = \frac{\gamma_{scaled} T_{total}(P) + P(1-\gamma_{scaled})T_{total}(P)}{T_{total}(P)} \\ &= P(1-\gamma_{scaled}) + \gamma_{scaled} = P + (1-P)\gamma_{scaled}. \end{aligned} \quad (12)$$

- Suppose we take the limit in P while holding $T_{compute}$ and $\gamma_{scaled}$ constant. That is, we are increasing the size of the problem so that the total running time remains constant when more processors are added. In this case, the speedup is linear in P.

## Gustafson's Law cont.

**Example 2**

Now, for the previous example, let's assume the scaled serial fraction of the same problem is 0.1, that is $\gamma_{scaled} = 0.1$, and $p = 16$. Then the scaled speedup is $S = 16(1 - 0.1) + 0.1 = 14.5$, which is greater than the upper bound determined by Amdahl's Law (10).

WITS UNIVERSITY

## Efficiency

- There are another two performance metrics we frequently use in the process of the course: efficiency and scalability.
- Efficiency:

  $$E = \frac{S}{P}, \tag{13}$$

  where $S$ is the speedup, and $P$ is the number of processes used to achieve the speedup. Note $0 < E \leq 1$. Efficiency is better when $E$ is closer to 1, which simply means you utilized the $P$ processors efficiently.

## Scalability

- Scalability: In general, a technology is scalable if it can handle ever-increasing problem size.
- In parallel program performance, scalability refers to the following measure. Suppose we run a parallel program using certain number of processors and with a certain problem size, and obtained an efficiency $E$. Further suppose that now we want to increase the number of processors. In this case, if we can find a rate at which the problem size increases so that we can still maintain the efficiency $E$, we say the parallel program scalable.

## Scalability cont.

### Example 3

Suppose $T_{serial} = n$, where $n$ is also the problem size. Also suppose $T_{parallel} = n/p + 1$. Then

$$E = \frac{n}{p(n/p+1)} = \frac{n}{n+p}.$$

To see if the problem is scalable, we increase $p$ by a factor of $k$, then we get

$$E = \frac{n}{n+p} = \frac{xn}{xn+kp}.$$

If $x = k$, then we have the same efficiency. That is, if we increase the problem size at the same rate that we increase the number of processors, then the efficiency remain constant, hence, the program is scalable.

UNIVERSITY

## Scalability cont.

- **Strong scalability:** When we increase the number of processes, we can keep the efficiency fixed without increasing the problem size, the program is strongly scalable.
- **Weak scalability:** If we can keep the efficiency fixed by increasing the problem size at the same rate as we increase the number of processes, then the program is said to be weakly scalable.

## Summary

# COMS3008A: Parallel Computing Lecture 2: Modelling Parallel Computation and Interconnection Networks

Hairong Wang

School of Computer Science & Applied Mathematics
University of the Witwatersrand, Johannesburg

Semester 2 2022

## Contents

## Outline

## Objectives

- Understand the physical architecture of parallel computers.

## Outline

## Outline

## RAM

- A computation model abstracts relevant properties of a computation from the irrelevant ones.
- Random access machine (RAM) is a sequential computation model. It consists of
  - A processing element (PE) (or processing unit (PU))
  - A memory

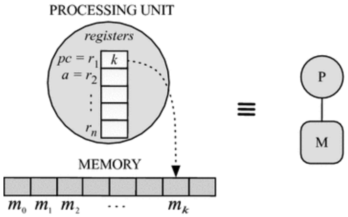


Figure: RAM model of computation: memory M - contains program instructions and data; processing unit P - execute instructions on data.

## PRAM

- A natural extension of RAM to parallel computation consists of multiple processing elements and a global memory of unbounded size that is uniformly accessible to all PEs.
- The generalization of RAM to parallel computing can be done in 3 different ways:
    - Parallel RAM (PRAM)
    - Local memory machine (LMM)
    - Modular memory machine (MMM)

## PRAM cont.

- PRAM

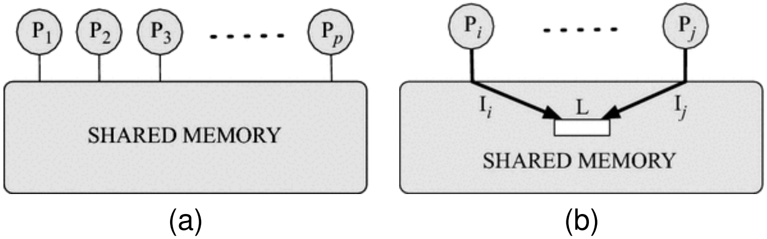


Figure: (a) PRAM model for parallel computation; (b) multiple PEs try to access the same memory location simultaneously.

## PRAM cont.

- What happens if multiple PEs need to access the same memory location? It could be both read, both write, or one read and the other write?
- A solution could be to serialize such contending accesses; however, we then have another issue about the uncertainty of which one will happen first — uncertainty.
- Problem with PRAM: simultaneous accesses to a memory location could lead to unpredictable data in PEs, as well as in the memory location accessed.
- A number of variants of PRAM are proposed, they differ in the ways in simultaneous access, and the ways in avoiding unpredictability.

## PRAM cont.

- Exclusive read exclusive write PRAM (EREW-PRAM): It does not support simultaneous access to the same memory location - any access to any memory location must be exclusive.
- Concurrent read exclusive write PRAM (CREW-PRAM): Allows simultaneous reads from the same memory location, but writing to a memory location must be exclusive.
- Concurrent read concurrent write (CRCW-PRAM): Supports simultaneous reads from the same memory location; simultaneous writes to the same memory location, and simultaneous reads and writes to the same memory location.

## PRAM cont.

- CRCW-PRAM: The unpredictability is handled in different ways:
  - Consistent CRCW-PRAM: PEs may simultaneously write to the same memory location, but they need to write the same value;
  - Abstract CRCW-PRAM: PEs may simultaneously try to write to the same memory location (not necessarily the same value), but only one of them will succeed, and it is unpredictable which one will succeed.
  - Priority CRCW-PRAM: There is a priority order imposed on PEs.
  - Fusion CRCW-PRAM: PEs may simultaneously try to write to the same memory location, but it is assumed that a particular operation is first performed on fly, and only the result of such operation will be written. Such operation should be associative and commutative, which includes sum, product, max, min, and logical AND and logical OR.

## PRAM cont.

- Note that the restriction of simultaneous access is relaxed from EREW-PRAM, to CREW-PRAM, and to CRCW-PRAM.
- This leads to some power gain from EREW-PRAM to CRCW-PRAM gradually, but not much, it is only in the order of logarithm.

## Outline

## LMM and MMM

- LMM: Each PE has its own local memory; accessing such memory is fast; A PE can access non-local memory via interconnect network
- MMM: No local memory to PEs;

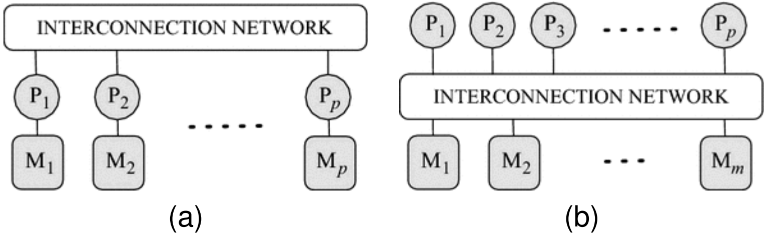


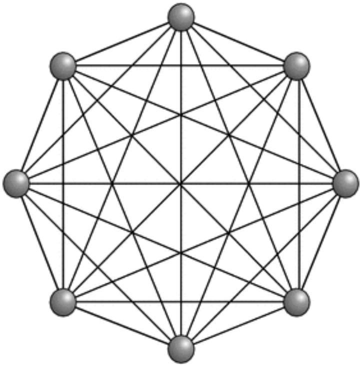
Figure: (a) LMM; (b) MMM.

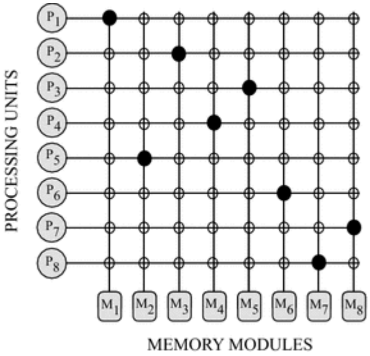
## Outline

## Outline

## Interconnection networks

- An important factor for the efficiency and scalability of parallel computers or programs: interconnection networks.
- Provide mechanisms for data communication between processing nodes or between processors and memory modules.


(a)



(b)

Figure: (a) Fully connected network; (b) A fully connected crossbar network.

## Important factors of interconnection networks

- Routing: the process for choosing a path in an interconnection network traffic;
- Flow control: the process of managing the rate of data transmission between two nodes to avoid scenario where a fast sender could overwhelm a slow receiver;
- Network topology: the arrangement of various elements, such as communication nodes and channels, of an interconnection network.

## Some basic properties of interconnection networks

- An interconnection network can be represented as a graph $G(V, E)$, where $V$ is the set of nodes, and $E$ is the set of links (or edges).
- Topological properties:
  - Node degree: the number of edges connecting a node
    - An interconnection network is regular if all nodes have the same node degree
  - Diameter
    - The number of nodes traversed by a packet from the source to the destination (a path) is called *hop count*
    - If there are multiple paths between a pair of source and destination nodes, the path with the shortest hop counts gives the minimum hop count, $l$
    - Average distance, $l_{avg}$: the average of all $l$s taken over all possible pairs of nodes.
    - Diameter: The maximum hop count $l$ taken over all possible pairs of source and destination nodes.

## Some basic properties of interconnection networks cont.

- Topological properties:
  - Path diversity: Multiple paths between a pair of communication nodes
  - Scalability: i) the capability of a network that handles growing amount of workload; ii) the potential of a network to be enlarged to accommodate growing amount of work.

## Some basic properties of interconnection networks cont.

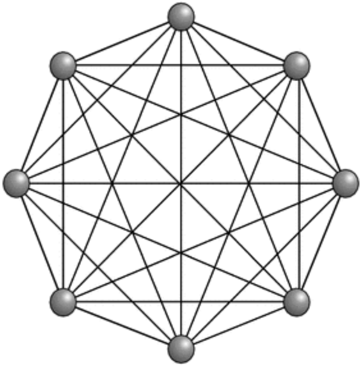
- Performance properties:
    - Bisection width: the minimum number of communication links that must be removed to partition the network into two equal parts (or almost equal parts)
    - Channel bandwidth: the peak rate at which data can be communicated over a communication link (channel), e.g., if the transfer time of a word is $t_w$, then the bandwidth is $1/t_w$.
    - Bisection bandwidth: the minimum volume of data communication allowed between any two halves of the network. It is the product of bisection width and channel bandwidth.
    - Cost: One way of defining the cost of a network is in terms of the number of communication links required by the network.

## Outline

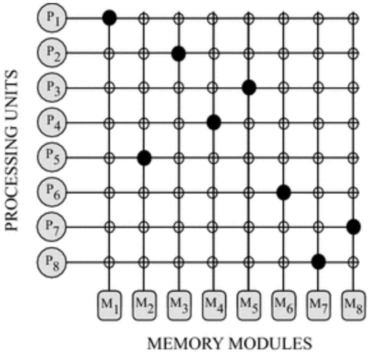
## The classification of interconnection networks

- Direct networks (static): Each node is directly connected to its neighbours. It has point-to-point communication links (network interface) between nodes.
    - Fully connected network: If the number of nodes is $n$, then such a network has $\frac{1}{2}n(n-1)$ connections.
- Indirect networks (dynamic): Connect nodes and memory modules via switches and communication links. A cross point is a switch that can be opened or closed. Uses switches to establish paths among nodes.
    - Fully connected crossbar switch: On one end is nodes, and the other end memory modules. The fully connected crossbar has too large complexity to be used for connecting large numbers of input and output ports. For example, if we have 1000 nodes and 1000 memory modules, then we need one million switches to build the fully connected crossbar switches.

## The classification of interconnection networks cont.



(a)



(b)

Figure: (a) Fully connected network; (b) A fully connected crossbar network.

## Popular topologies of interconnection networks

- Bus. Used in both LMMs and MMMs. At one time, only one process is allowed to use the bus for communication.
  - Advantages: simple to build; buses are ideal for broadcasting data among nodes.
  - Disadvantages: unscalable in terms of performance (but scalable in terms of cost)

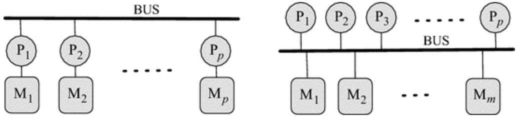


Figure: Bus topology.

## Popular topologies of interconnection networks cont.

- Linear array. Used in LMMs. Every node (except the two nodes at the ends) is connected to two neighbours, see Figure 7. Simple. If a node index is $i$, its two neighbours can be found using indices $(i+1) \bmod n$ and $(i-1) \bmod n$, where $n$ is the total number of nodes in the linear array.

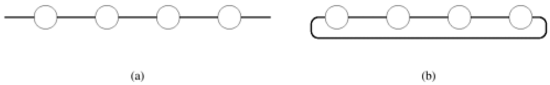


Figure: (a) Linear array without wraparound link; (b) linear array with wraparound link (also called ring, see next slide).

## Popular topologies of interconnection networks cont.

- Ring. Used in LMMs. Every node is connected to two neighbours, see Figure 8. Simple. If a node index is $i$, its two neighbours can be found using indices $(i+1) \bmod n$ and $(i-1) \bmod n$, where $n$ is the total number of nodes in the ring.

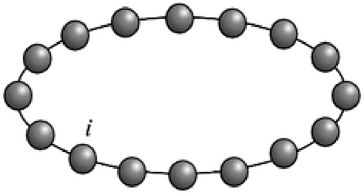
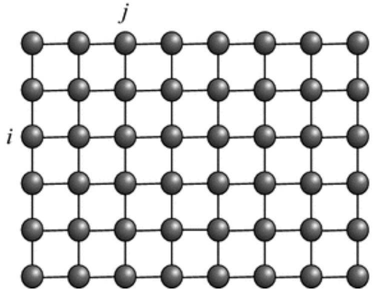


Figure: Ring topology. Each node represents a processing element with local memory.

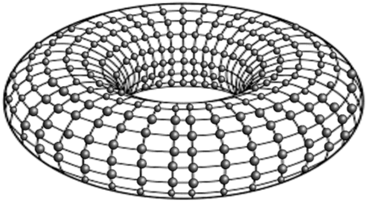
## Popular topologies of interconnection networks cont.

- 2D mesh. Can be used in LMMs. Each node is connected to a switch. The number of switches can be determined by the lengths of the two sides. Every switch, except those along the 4 borders, has 4 neighbours.
- 2D torus. Similar to 2D mesh, however, each pair of corresponding border switches is connected. Every switch has 4 neighbours.

## Popular topologies of interconnection networks cont.



(a)



(b)

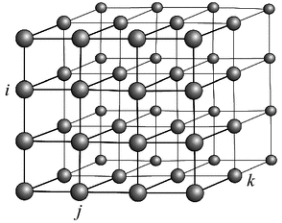
Figure: (a) 2D mesh topology; (b) 2D torus topology. Each node represents a processing element with local memory.

## Popular topologies of interconnection networks cont.

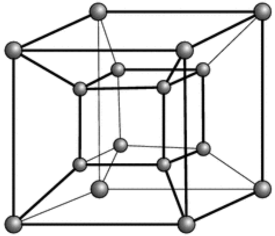
- 3D mesh. Similar to 2D mesh, however, in 3 dimension. Every switch except the border ones, has 6 neighbours.
- 3D torus, Every pair of opposite switches are connected in 3D mesh.
- Hypercube: An interconnection network that has $n = 2^d$ nodes, where $d$ is the number of dimensions. Each node has a distinct label consisting of $d$ binary bits. For example, $d = 3$, then $n = 8$. Two nodes are connected via a link if and only if their labels differ in only one bit location. Used in LMMs.

WITS
UNIVERSITY

## Popular topologies of interconnection networks cont.



(a)



(b)

Figure: (a) 3D mesh, (b) Hypercube topology Each node represents a processing element with local memory.

# Popular topologies of interconnection networks cont.

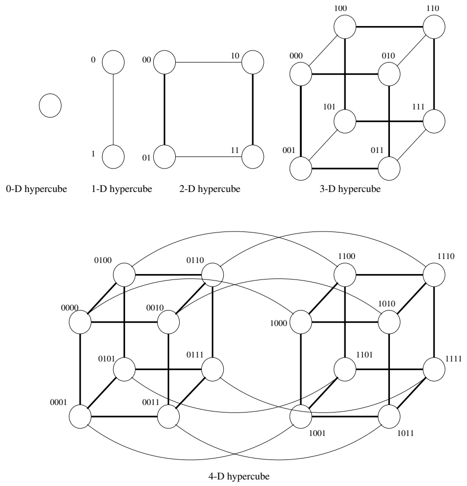


Figure: Hypercubes of 1D, 2D, 3D, and 4D.

## Popular topologies of interconnection networks cont.

- Multistage network: used in MMM, where input switches are connected to PEs, and output switches are connected to memory modules.

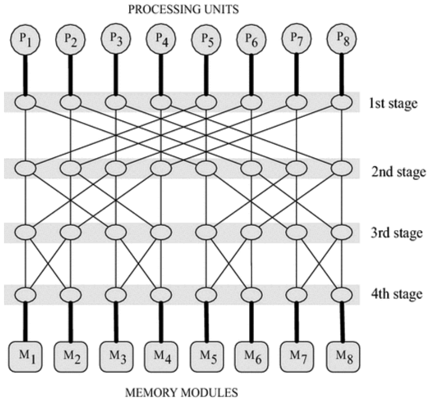


Figure: Multistage network topology. A 4-stage interconnection network capable of connecting 8 PEs to 8 memory modules. Each switch can establish a connection between a pair of input and output channels.

## Popular topologies of interconnection networks cont.

- Fat tree: used in constructing LMM, where PEs with their local memories are attached to the leaves.

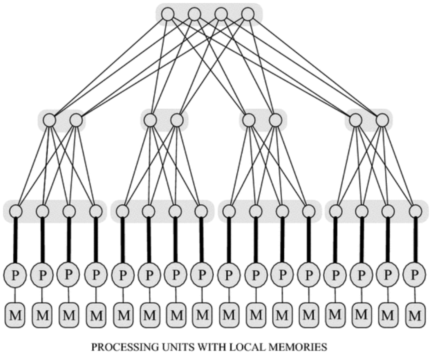


Figure: Fat tree topology.

## Popular topologies of interconnection networks cont.

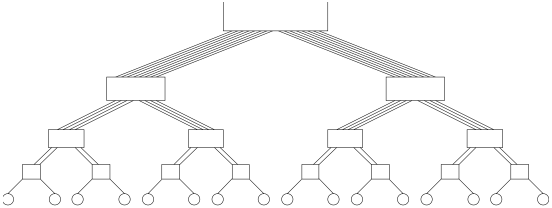


Figure: Fat tree topology. Each switch can establish a connection between arbitrary pair of incidents. Edges closer to the root are thicker. The idea is to increase the number of communication links and switching nodes closer to the root.

## Outline

## Evaluating the interconnect network

- Diameter
- Bisection width
- Cost

Table: Quantitative characteristics of various interconnect networks

| | Network | Diameter | bisection width | Cost |
|---|---|---|---|---|
| Static | Fully connected | 1 | $p^2/4$ | $p(p-1)/2$ |
| | Linear array | p-1 | 1 | p-1 |
| | 2D mesh | $2(\sqrt{p}-1)$ | $\sqrt{p}$ | $2(p-\sqrt{p})$ |
| | Ring | $\lfloor p/2 \rfloor$ | 2 | p |
| | 2D torus | $2\lfloor \sqrt{p}/2 \rfloor$ | $2\sqrt{p}$ | 2p |
| | Hypercube | $\log p$ | $p/2$ | $(p \log p)/2$ |
| Dynamic | Crossbar | 1 | p | $p^2$ |
| | Fat tree | $2 \log p$ | the # of links | p-1 (or the # of links) |

*With p nodes in the above networks

## Summary & References

- Summary
  - The extension of RAM to parallel computing
  - Interconnection networks: properties and topologies
- Bibliography:

  - Introduction to Parallel Computing: From Algorithms to Programming on State-of-the-Art Platforms, by Roman Trobec, Boštjan Slivnik, Patricio Bulić, Borut Robič, Springer, 2018,
  - Introduction to Parallel Computing, second edition, by Ananth Grama, Anshul Gupta, George Karypis, Vipin Kumar. Addison Wesley Publisher, 2003.

# COMS3008A:
# Parallel Computing
# Lecture 3: Parallel Algorithm Design

Hairong Wang

School of Computer Science & Applied Mathematics
University of the Witwatersrand, Johannesburg

2022-8-11

## Contents

## Outline

## Objectives

- Learn how to design a parallel program given a problem (often starting from a serial solution)

## Outline

1. Objectives
2. Parallel Program Design
   - Shared memory vs distributed memory
   - Shared memory
   - Distributed memory
   - Parallel program design

## Outline

## Shared memory vs distributed memory

- Physical memory in parallel computers is either shared or local.
- Shared memory parallel computer: parallel computers that share memory space
- Distributed memory parallel computer: parallel computers that do not share memory space

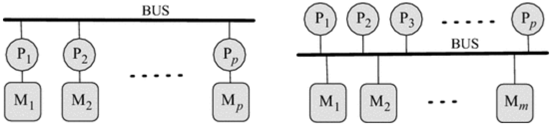


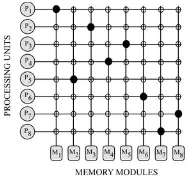
Figure: Bus based parallel computers.

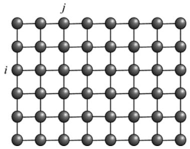
## Shared memory vs distributed memory

- From programming point of view: shared memory programming and distributed memory programming
  - Shared memory programming: All PEs have equal access to shared memory space. Data shared among PEs are via shared variables.
  - Distributed memory programming: A PE has access only to its local memory. Data (message) shared among PEs are communicated via the communication channel, i.e., interconnection network.

## Shared memory vs distributed memory cont.

- In terms of programming, programming shared memory parallel computers is easier than programming distributed memory systems.
- In terms of scalability, shared memory systems and distributed memory systems display different performance and cost characteristics. For example,
  - Bus, scalable in terms of cost, but not performance
  - Fully connected crossbar switch, scalable in terms of performance , but not cost
  - Hypercube, scalable both in performance and cost
  - 2D mesh, scalable both in performance and cost
- In terms of problem size, distributed memory systems are more suitable for problems with vast amount of data and computation





WITS UNIVERSITY

## Outline

## Shared memory

- In shared memory programming, variables have two types: shared variable and private variable.
- Shared variable can be read and write by any thread; and private variables can usually only be accessed by one thread.
- In shared memory systems, communication among threads happen via shared variables — this means communication is implicit.
- Non-determinism in shared memory programming.

## Shared memory cont.

- Non-determinism: A computation is non-deterministic if a given input can result in different outputs.
- In a shared memory system, non-determinism arises when multiple threads are executing independently, and at different rates. Then these threads could complete at different orders for different runs, and hence could give different outputs from run to run. This is a typical case of non-determinism.
- Non-determinism can be harmless for some problem, and can cause issues for some problems.

## Shared memory cont.

### Example 1

Suppose we have two threads in a parallel program, one with thread ID 0, and the other with thread ID 1. Suppose also that each thread has a private variable `my_x`; thread 0 stores a value 5 for `my_x`, and thread 1 a value 9 for its `my_x`.

| Thread ID | Variable | Value |
|---|---|---|
| 0 | `my_x` | 5 |
| 1 | `my_x` | 9 |

Now, what will be the output like if both threads execute the following line of code?

```
...
printf("Thread %d > my_x = %d\n", my_ID, my_x);
...
```

## Shared memory cont.

### Example 1 cont.

The output could be

```
Thread 0 > my_x = 5
Thread 1 > my_x = 9
```

or

```
Thread 1 > my_x = 9
Thread 0 > my_x = 5
```

## Shared memory cont.

### Example 2

Suppose we have a shared variable `min_x` (with initial value $\infty$) and two threads in our program. Each thread has a private variable `my_x`, thread 0 stores a value 5 for `my_x`, and thread 1 a value 9 for its `my_x`. What will happen if both threads update `min_x = my_x` simultaneously?

```
/*each thread tries to update variable min_val as
    follows*/
if (my_x < min_x)
  min_x = my_x;
```

## Shared memory cont.

### Example 2 cont.

- This is the situation where two threads more or less try to update a shared variable (`min_x`) simultaneously.
- **Race condition:** When threads attempt to access a resource simultaneously, and the access can result in error, we often say the program has a race condition.
- In such case, we can serialize the contending activities by setting a critical section where the section can only be accessed by one thread at a time.

## Outline

## Distributed memory

- In distributed memory programming, the processes can only access their own private (or local) memory. Message-passing programming model (or APIs) is most commonly used. For example, MPI.
- In message passing, each process is identified by its rank.
- Processes communicate with each other by explicitly sending and receiving messages.
- Message passing APIs often provide basic send and receive functions, they also provide more powerful communication functions such as broadcast and reduction.
  - Broadcast, a single process transmits the same data to all processes;
  - Reduction, the results computed by individual processes are combined to a single results, e.g., summation.

## Outline

## Parallel program design

The following steps are often taken to solve a problem in parallel [1]:

- Decomposition (or partitioning) of the computation into tasks;
- Assignment of tasks to processes;
- Orchestration of the necessary data access, communication, synchronization among processes;
- Mapping of processes to processors.

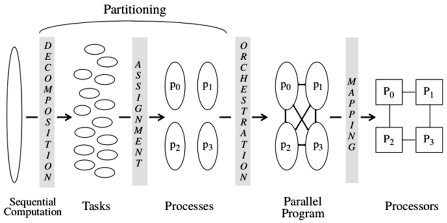


Figure: Steps in parallelization, relationship between PEs, tasks and processors.

[1] Culler, D. E., Singh, J. P., and Gupta, A. (1998). *Parallel Computer Architecture: A Hardware/Software Approach*.

## Parallel program design cont.

- Decomposition involves decomposing a problem into finer tasks such that these tasks could be executed in parallel.
- The major goal of decomposition is to expose enough concurrency to keep processes busy all the time, yet not so much that overhead of managing tasks become substantial.
- Assignment involves assigning fine tasks to available processes, such that each process has approximately similar workload. Load balancing is an challenging issue in parallel programming.
- The primary performance goals of assignment are to achieve balanced workload, reduce the runtime overhead of managing assignment.

## Parallel program design cont.

- The third step involves necessary orchestration, could involve communication among processes and synchronization.
- The major performance goals in orchestration:
    - reducing the cost of communication and synchronization
    - preserving locality of data reference
    - scheduling tasks
    - reducing the overhead of parallelism management

## Parallel program design cont.

- Finally, the mapping is to map processes to processors. The number of processes and the number of processors are not necessarily need to be matched. That is, for example, you can have 8 processes on a 4 processor machines. In such a case, a processor may handle more than one processes by techniques such as space sharing and time sharing.
- The mapping process can be taken care of by OS in order to optimize resource allocation and utilization. The program may also control the mapping of course.

## Parallel program design — simple example

**Example 3**

Consider a simple example program with two phases.

- In the first phase, a single operation is performed independently on all points of a 2-dimensional $n$ by $n$ grid;
- in the second phase, the sum of $n^2$ grid point values is computed.

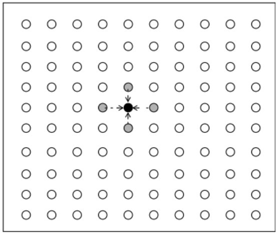


Figure: 2D grid points.

## Parallel program design — simple example cont.

**Example 3 cont.**

If we have $p$ processes,

- we can assign $n^2/p$ points to each process and complete the first phase in time $n^2/p$.
- In the second phase, each process can add each of its assigned $n^2/p$ values to a global sum variable.

## Parallel program design — simple example cont.

### Example 3 cont.

- Issue: the second phase is in serial where it takes $n^2$ time regardless of multiple processes. The total time is $n^2/p + n^2$. Then the speedup, compared to the sequential time $2n^2$, is

$$s = \frac{2n^2}{\frac{n^2}{p} + n^2} = \frac{2}{\frac{1}{p} + 1},$$

  which is at most 2, even if a large number of processes is used.
- Can we expose more concurrency?

## Parallel program design — simple example cont.

### Example 3 cont.

We can improve the performance:

- We can first compute local sums of $n^2/p$ values on all processes simultaneously.
- After $n^2/p$ time, we have $p$ number of local sums.
- We then add these $p$ local sums into a global sum one at a time, which takes $p$ units of time.
- Now the total time is $n^2/p + n^2/p + p$. The speedup is

$$s = \frac{2n^2}{\frac{2n^2}{p} + p} = p \times \frac{2n^2}{2n^2 + p^2}.$$

This speedup is almost linear in $p$, the number of processors used, when $n$ is large compared to $p$.

# Parallel program design — simple example cont.

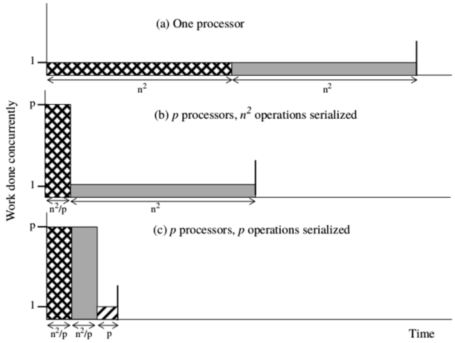


Figure: Impact of limited concurrency.

## Parallel program design — simple example cont.

**Example 4**

Suppose we have an array with large quantities of floating point data stored in it. In order to have a good feeling of the distribution of the data, we can find the histogram of the data. To find the histogram of a set of data, we can simply divide the range of data into equal sized subintervals, or bins, determine the number of values in each bin, and plot a bar graphs showing the sizes of the bins.

## Parallel program design — simple example cont.

### Example 4 cont.

As a very small example, suppose our data are

$$A = [1.3, 2.9, 0.4, 0.3, 1.3, 4.4, 1.7, 0.4, 3.2, 0.3,$$
$$4.9, 2.4, 3.1, 4.4, 3.9, 0.4, 4.2, 4.5, 4.9, 0.9]$$

- For $A$, $A_{min} = 0.3$, $A_{max} = 4.9$, $A_{count} = 20$.
- Let's set the number of bins to be 5, and the bins are the $[0, 1.0), [1.0, 2.0), [2.0, 3.0), [3.0, 4.0), [4.0, 5.0)$, $bin_{width} = (5.0 - 0)/5 = 1.0$.
- Then $bin_{count} = [6, 3, 2, 3, 6]$ is the histogram — the output is an array the number of elements of data that lie in each bin.

## Parallel program design — simple example cont.

### Example 4 cont.

```
for (i = 0; i < data_count; i++){
    bin = Find_bin(data[i], ...);
    bin_count[bin]++;
}
```

The `Find_bin` function returns the bin that `data[i]` belongs to.

## Parallel program design — simple example cont.

### Example 4 cont.

- Now if we want to parallelize this problem, first we can decompose the dataset into subsets. Given the small dataset, we divide it into 4 subsets, so that each subset has 5 elements.
  - Identify tasks (decompose): i) finding the bin to which an element of data belongs; ii) increment the corresponding entry in `bin_count`.
  - The second task can only be done once the first task has been completed.
  - If two processes or threads are assigned elements from the same bin, then both of them will try to update the count of the same bin. Assuming the `bin_count` is shared, this will cause **race condition**.

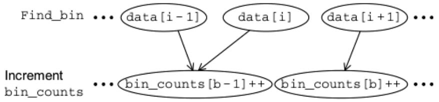


Figure: Tasks and their communications.

## Parallel program design — simple example cont.

### Example 4 cont.

- A solution to race condition in this example is to create local copies of `bin-count`, each process updates their local copies, and at the end add these local copies into the global `bin_count`.

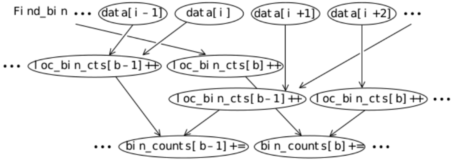


Figure: Tasks and communications.

## Parallel program design — simple example cont.

### Example 4 cont.

- In summary, the parallelization approach is to
  - Elements of data are assigned to processes/threads so that each process/thread gets roughly the same number of elements;
  - Each process/thread is responsible for updating its `local_bin_counts` array based on the assigned data elements.
  - The local `local_bin_counts` are needed to be aggregated into the global `bin_counts`.
    - If the number of bins is small, the final aggregation can be done by a single process/thread.
    - If the number of bins is large, we can apply parallel addition in this step too.

## Summary

- Aspects of parallel program design
  - Decomposition to create concurrent tasks
  - Assignment of works to workers
  - Orchestration to coordinate processing of tasks by processes/threads
  - Mapping to hardware

  We will look more into making good decisions in these aspects in the coming lectures.