

# HELP US ALL STAY HEALTHY

## SIX SIMPLE TIPS



Maintain 1.5 metres distance  
between yourself and others  
where possible



Cough and sneeze into  
your elbow or a tissue  
(not your hands)



Avoid shaking hands



Put used tissues  
in the bin



Wash hands with soap and  
warm water or use an alcohol-  
based hand sanitiser after you  
cough or sneeze



Do not touch  
your face

### IF YOU ARE UNWELL AND WORRIED ABOUT COVID-19:

- Call the National  
Coronavirus Helpline:  
1800 020 080
- Call your usual GP for advice
- Call the UWA Medical Centre  
for advice: 6488 2118

UWA FAQs:  
[uwa.edu.au/coronavirus](https://uwa.edu.au/coronavirus)

Report COVID-19 hazards  
and suspected/confirmed  
cases via RiskWare:  
[uwa.edu.au/riskware](https://uwa.edu.au/riskware)



THE UNIVERSITY OF  
**WESTERN  
AUSTRALIA**

# High-Performance Computing

## Lecture 10 Parallel Program Design and Performance Modelling

---

CITS5507

---

Zeyi Wen

---

Computer Science and  
Software Engineering

---

School of Maths, Physics  
and Computing

Acknowledgement: The lecture slides are adapted from many online sources.

- Review of Parallel Programing
  - ✓ Why Parallel Programing
  - ✓ Why Parallel Code Slower
  - ✓ Improving Programs with Parallelism
  - ✓ The Structure of MPI and Common Error
- Performance Modelling
  - ✓ Why and What is Performance Modelling?
  - ✓ Performance Metrics in Parallel Systems

- Exploiting parallelism is often difficult
- Hardware moves fast, problems grow fast, writing code takes time
- parallelism is used in many applications

## **Engineering**

- Airfoil design
- Internal combustion engines  
/ electric drivetrains
- High variable problems
- Industrial processes
- ...

## **Science**

- Drug design
- Human genome sequencing
- ...

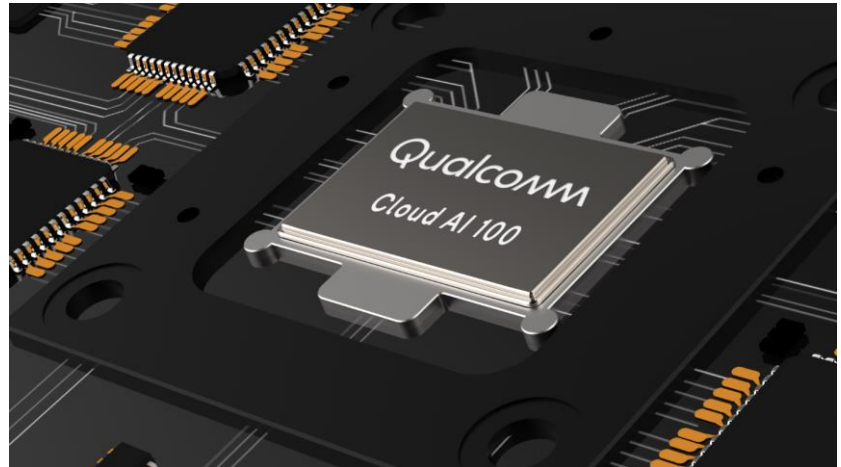
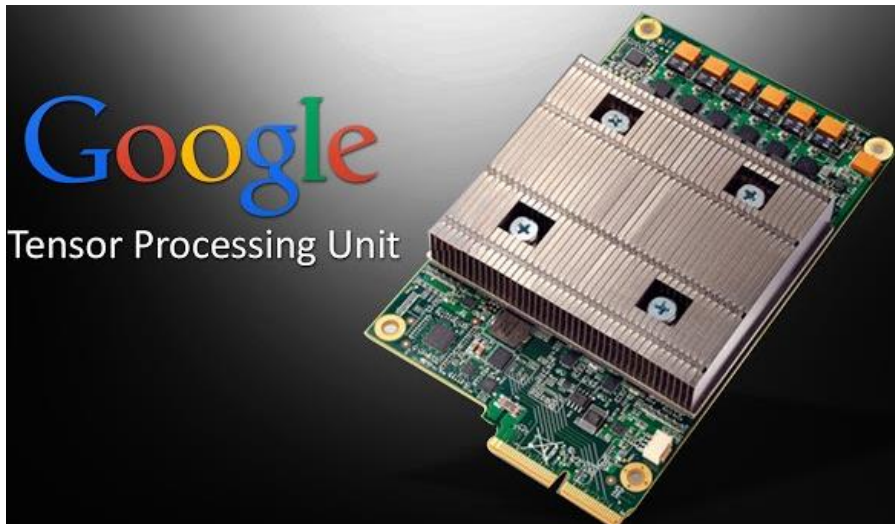
## **Industry**

- Serving the largest websites
- Financial trading
- ...



# Specialised Hardware and Software

- The fastest **software** for particular **hardware** will be written specifically for that **hardware**.
- The fastest **hardware** for a particular **software** will be built specifically for that **software**.
- Two main options (we've now seen both)
  - ✓ Shared memory address space
  - ✓ Message passing



- Review of Parallel Programing
  - ✓ Why Parallel Programing
  - ✓ Why Parallel Code Slower
  - ✓ Improving Programs with Parallelism
  - ✓ The Structure of MPI and Common Error
- Performance Modelling
  - ✓ Why and What is Performance Modelling?
  - ✓ Performance Metrics in Parallel Systems

- Sometime parallel code can be slower than serial code
- **A few key factors** that determinate the performance of the parallel code include:
  - ✓ **parallel task granularity,**
  - ✓ **communication overhead,**
  - ✓ **load balancing among processes,**
  - ✓ **false sharing memory/cache.**

- In parallel computing, **granularity** (or grain size) of a task is a measure of the amount of work (or computation) which is performed by that task.
- The **granularity** of the parallel task must be enough to overleap the parallel model overheads (parallel task creation and communication between them).
- In order to reduce the communication overhead, granularity can be increased. Coarse grained tasks have **less communication overhead** but they often **cause load imbalance**.



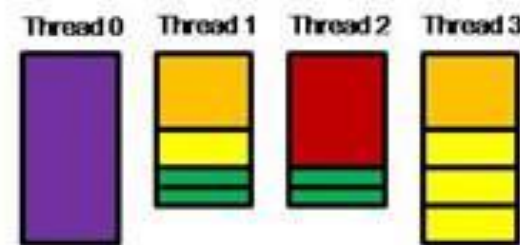
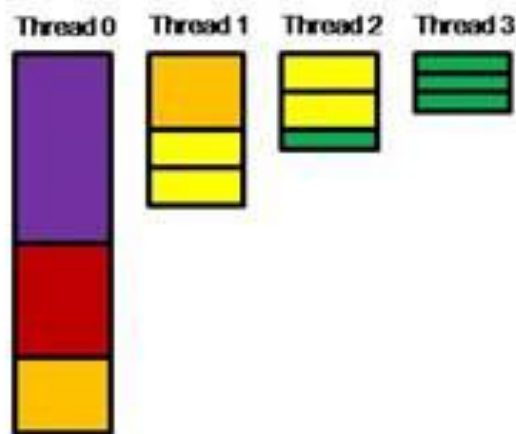
- Every time one process intends to communicate with others, it has the cost of creating/sending the message and in case of using a synchronous communication routine there is also the cost of waiting for the other processes to receive the message.
- Reduce the amount of communication and synchronisation between parallel tasks.
- Potential solutions:
  - ✓ **computation instead of communication,**
  - ✓ **asynchronous communications,**
  - ✓ **collective communications,**
  - ✓ **faster communication hardware.**

# Communication Overhead

L1 cache reference/hit	1.5 ns	4 cycles
Floating-point add/mult/FMA operation	1.5 ns	4 cycles
L2 cache reference/hit	5 ns	12 ~ 17 cycles
L3 cache hit	16-40 ns	40-300 cycles
256MB main memory reference E5-2690v4	75-120 ns	TinyMemBench on "Broadwell"
Read 1MB sequentially from disk (seek time would be additional latency)	5,000,000 ns	5,000 us
Random Disk Access (seek+rotation)	10,000,000 ns	10,000 us
Send packet CA->Netherlands->CA	150,000,000 ns	150,000 us

# Load Balancing among Processes/Threads

- A good load balancing can maximise the work done in parallel.
- Each process/thread should take approximately the same time to finish their work.
- If processors have different speeds, then it might need a more complex task distribution



- False sharing is a well-known performance issue on SMP systems, where each processor has a local cache.
- It occurs when threads on different processors **modify** variables that reside on the same cache line.
- This circumstance is called false sharing because each thread is not actually sharing access to the same variable.

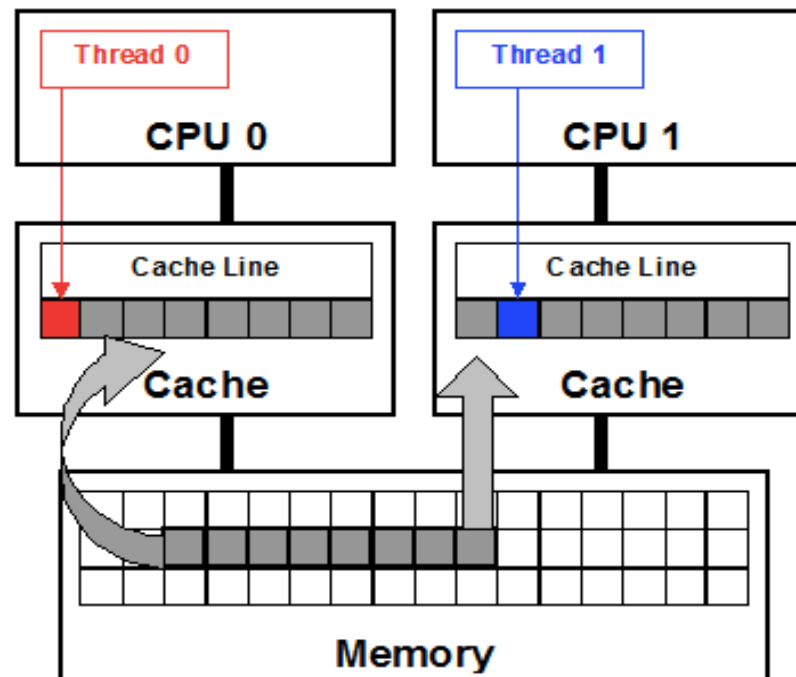
# False Sharing Memory/Cache (Example)

- This code block cause false sharing

```
double sum=0.0, sum_local[NUM_THREADS];
#pragma omp parallel num_threads(NUM_THREADS)
{
    int me = omp_get_thread_num();
    sum_local[me] = 0.0;
    #pragma omp for
    for (i = 0; i < N; i++)
        sum_local[me] += x[i] * y[i];
    #pragma omp atomic
    sum += sum_local[me];
}
```

# False Sharing Memory

- There is a potential for false sharing on array **sum\_local**.
- This array is dimensioned according to the number of threads and is small enough to fit in a single cache line.
- When executed in parallel, the threads **modify** different, but adjacent, elements of **sum\_local** which invalidates the cache line for all processors.





- Review of Parallel Programing
  - ✓ Why Parallel Programing
  - ✓ Why Parallel Code Slower
  - ✓ Improving Programs with Parallelism
  - ✓ The Structure of MPI and Common Error
- Performance Modelling
  - ✓ Why and What is Performance Modelling?
  - ✓ Performance Metrics in Parallel Systems

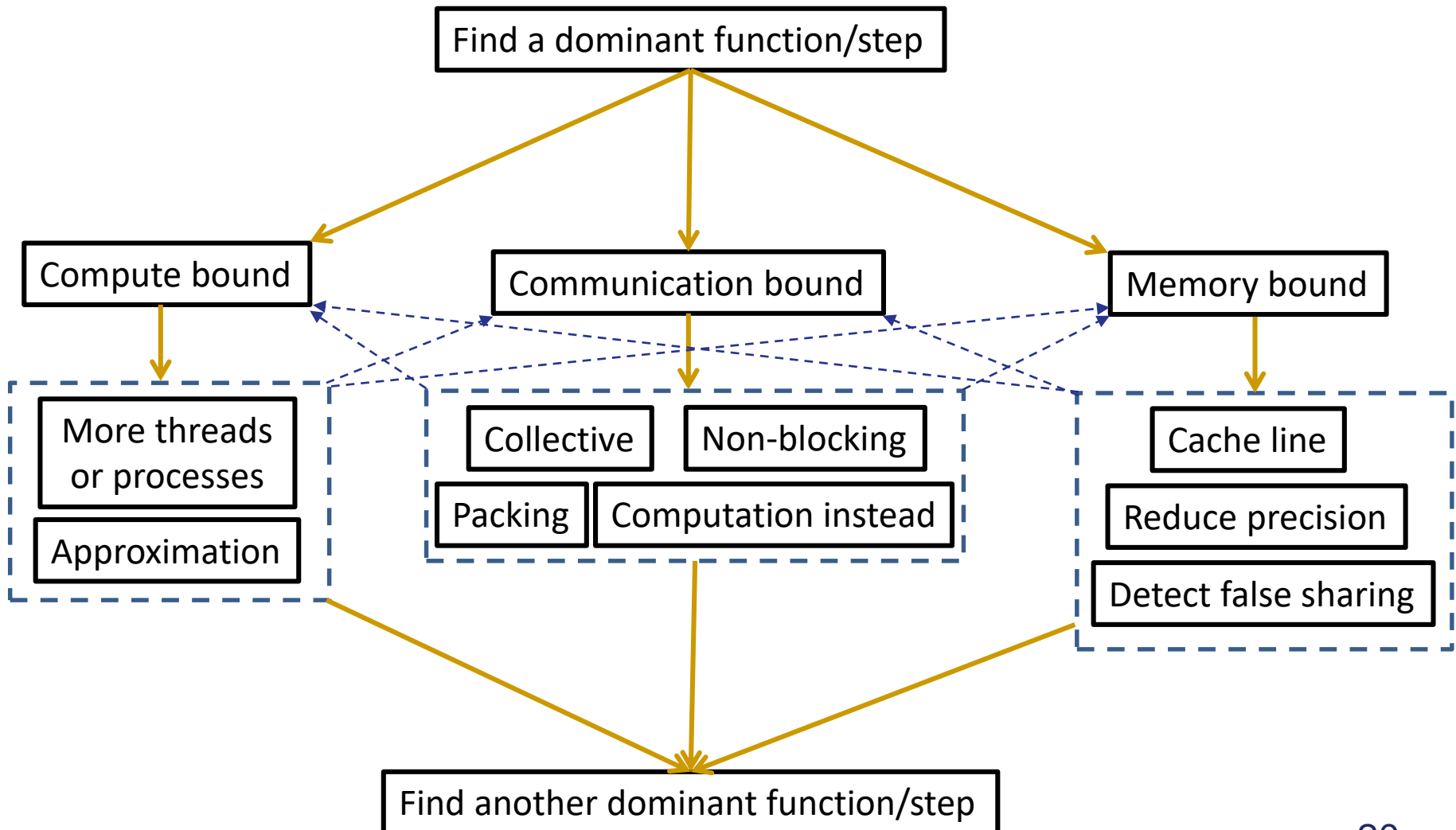
- Install OpenMP and MPI locally
  - ✓ You can simulate running multiple nodes on a single machine with `mpiexec/mpirun` calls
- Write a **serial version** of your solution first
  - ✓ Serial code is vastly easier to parallelise
- Write a **parallel version** of your solution
  - ✓ Break the serial code into multiple steps/components
  - ✓ Measure the elapsed time of each step/component
  - ✓ Optimise the most time consuming step/component
  - ✓ Optimise all the steps/components that can be parallelised

- Write a 'dummy' parallelised version of your code first
  - ✓ Have each process/thread compute its bounds first and print them out
  - ✓ Test your solution for different configurations (e.g. different number of threads) and check correctness
  - ✓ Gradually implement actual parallel computation
- Read documentation
  - ✓ there may be a function to help make your life easier
- Test your parallel code in serial or 2 processes/threads
  - ✓ A parallelised piece of code often still work if only one process/thread is available

- Use the sysadmins
  - ✓ If you end up using a commercial HPC system (e.g. Pawsey Supercomputing Centre) use the helpdesk
- Invest in some tests
  - ✓ Writing a few small examples and making them easy to run will make testing changes easier
- Good printouts are invaluable
  - ✓ It helps to quickly find out where your code may be bugged and what values are changing

- Make writing code easy for you
  - ✓ Many IDEs / editors (CLion, VSCode, etc.) allow for a full remote mode. Write code directly on the supercomputer using your own editor
  - ✓ Otherwise
    - write code locally
    - test locally
    - commit with git
    - pull the edits on the supercomputer and run

# Workflow of Accelerating Programs





- Only major difference vs. sequential design is deciding what should be in parallel
- Identifying separate portions of work
- Mapping concurrent tasks onto multiple processes/threads
- Distributing input, output and intermediate data
- Managing access to shared data
- Synchronising processes at various stages

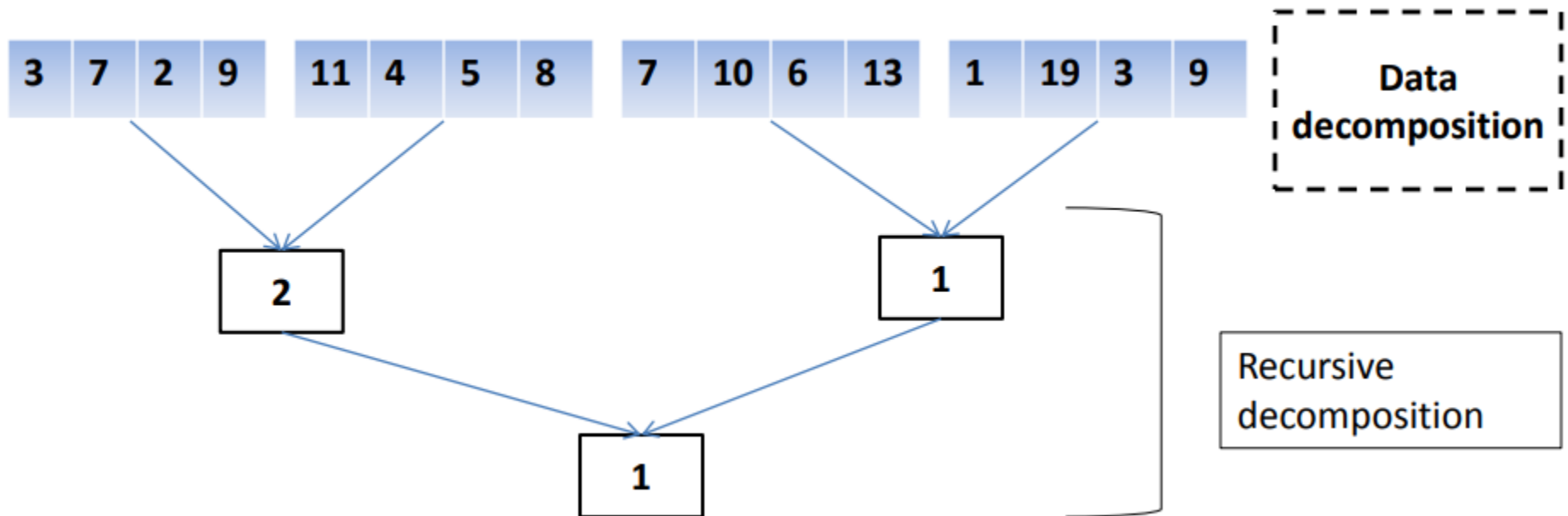
- We define **granularity** in task decomposition
  - ✓ A large number of small tasks is fine-grained
  - ✓ A small number of large tasks is coarse-grained
- The degree of concurrency is the maximum number of parallel tasks in your program
- **Coarse grained** - Divide the problem into **big tasks**, run many at the same time, coordinate when necessary.
- **Fine grained** - For each “operation”, divide across functional units such as floating point units.

- Coarse grained example (a research project)
  - ✓ Set students on different problems in a research area,
  - ✓ give each person a list of task, and
  - ✓ have them do everything
- Fine grained example (writing a list of letters)
  - ✓ send out lists of letters break into steps,
  - ✓ make everyone write letter text,
  - ✓ stuff envelope, write address, and apply stamp,
  - ✓ then collect and mail.
- There is no clear separation of coarse and fine grained

- Recursive
  - ✓ a large task is broken into smaller sub-tasks
  - ✓ the sub-tasks can be further divided until small enough
  - ✓ e.g. sorting: partition a large array into smaller arrays
- Data
  - ✓ useful when a problem is built on a large data structure
  - ✓ divide the structure (and thus the computation)
  - ✓ decomposition can apply to: input, output, intermediate
- Exploratory
  - ✓ useful when problems computations correspond to searching a state space of solutions
  - ✓ e.g. computing chess moves
- Speculative
  - ✓ useful when a program can take one of many possible paths
  - ✓ speedup is related to the number of speculative paths

# Hybrid Decomposition

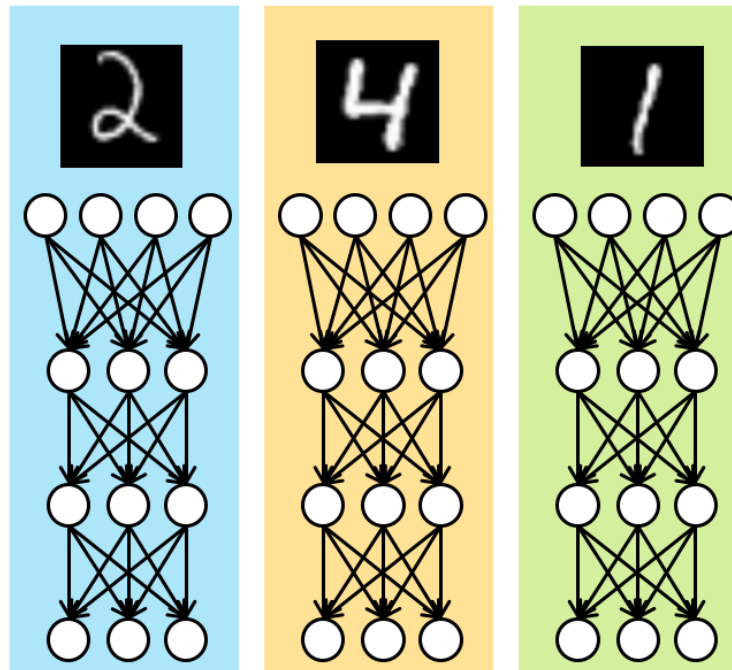
- Use several decomposition methods together
- Example: finding the minimum of any array of size 16 using 4 tasks.



- Data parallel
  - ✓ identical steps performed to different data elements
  - ✓ parallelism scales with problem size
- Task graph
  - ✓ parallel task can be described by task dependencies
  - ✓ useful for complex, interactive or recursive problems
- Work pool
  - ✓ dynamic task mapping to processes for load-balancing
  - ✓ work available up-front or dynamically generated
- Master-worker architecture
  - ✓ a node is selected to generate work for worker nodes
  - ✓ a common parallel computing architecture
- Pipelining
  - ✓ a stream of data is passed to a set of processes
  - ✓ can be linear or more complicated

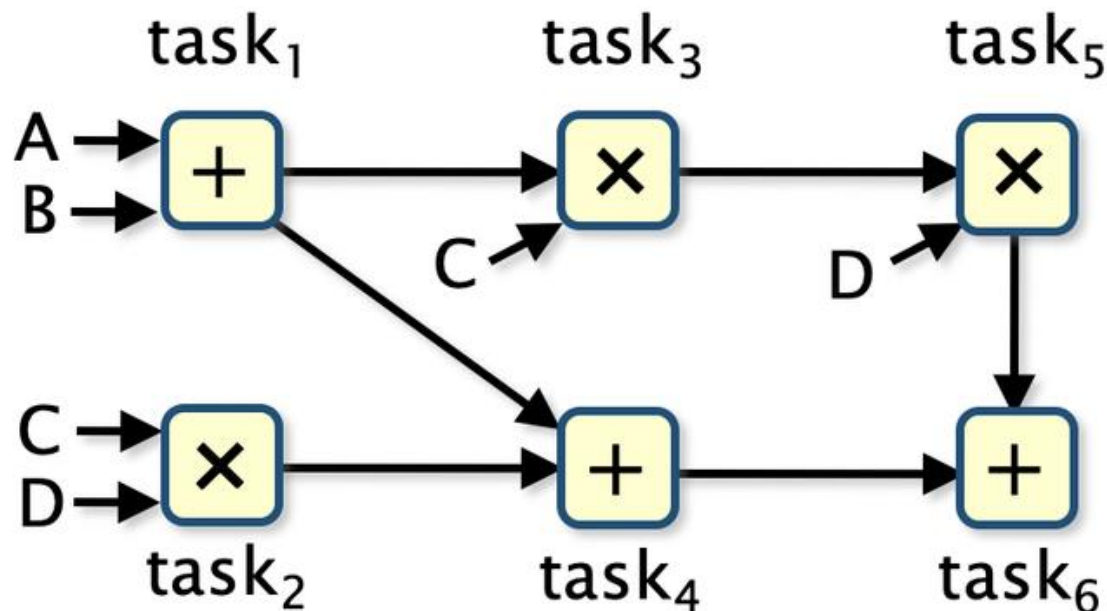


- Data parallel
  - ✓ identical steps performed to different data elements
  - ✓ parallelism scales with problem size
- Task graph
- Work pool
- Master-worker architecture
- Pipelining

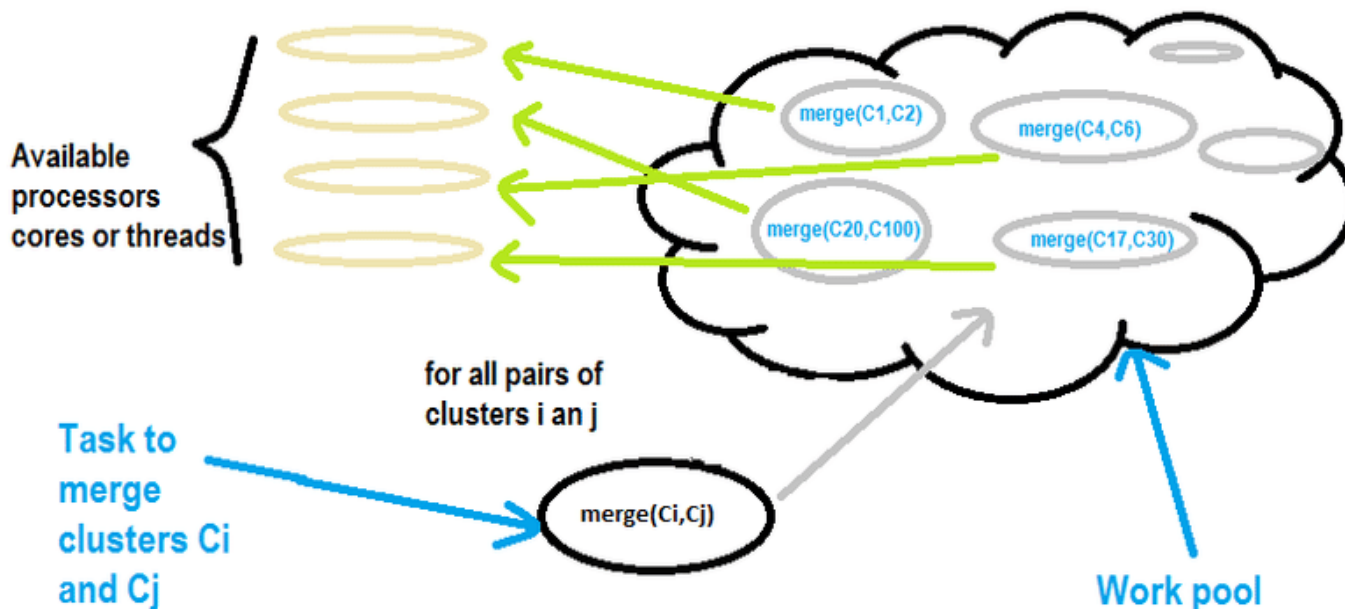


# Task Graph

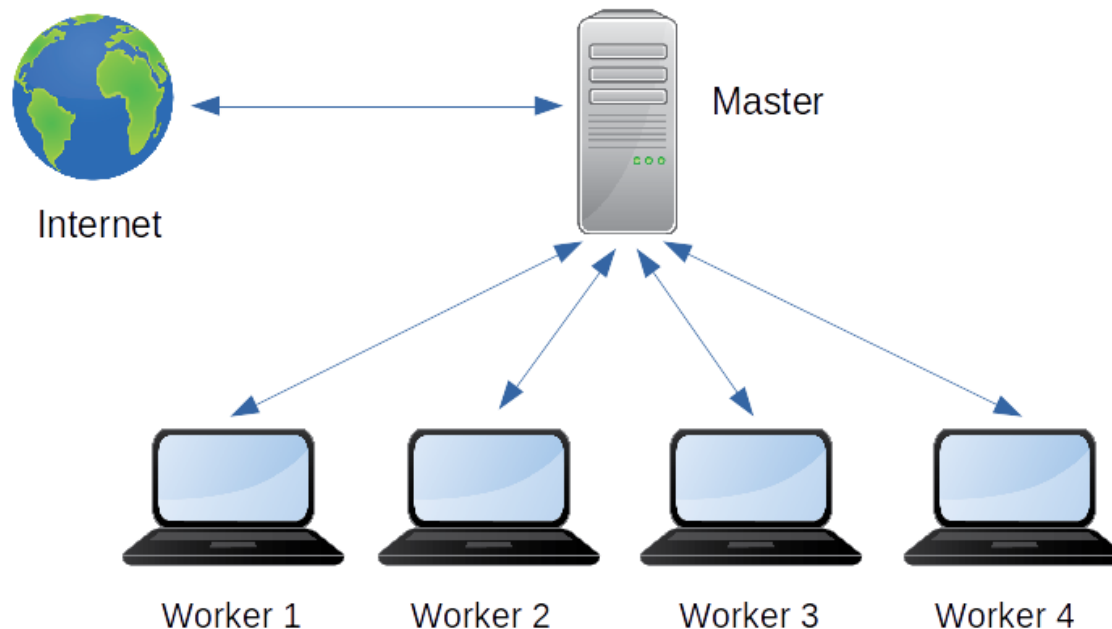
- Data parallel
- Task graph
  - ✓ parallel task can be described by task dependencies
  - ✓ useful for complex, interactive or recursive problems
- Work pool
- Master-worker architecture
- Pipelining



- Data parallel
- Task graph
- Work pool
  - ✓ dynamic task mapping to processes for load-balancing
  - ✓ work available up-front or dynamically generated
- Master-worker architecture
- Pipelining



- Data parallel
- Task graph
- Work pool
- Master-worker architecture
  - ✓ a node is selected to generate work for worker nodes
  - ✓ a common parallel computing architecture
- Pipelining



# Pipelining

- Pipelining
  - ✓ a stream of data is passed to a set of processes
  - ✓ can be linear or more complicated

Sequential processing

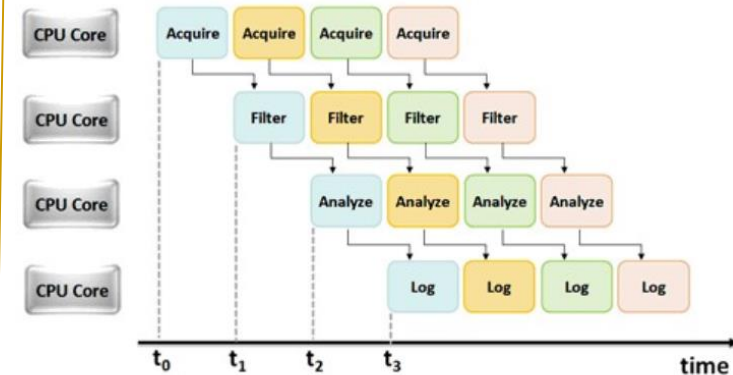
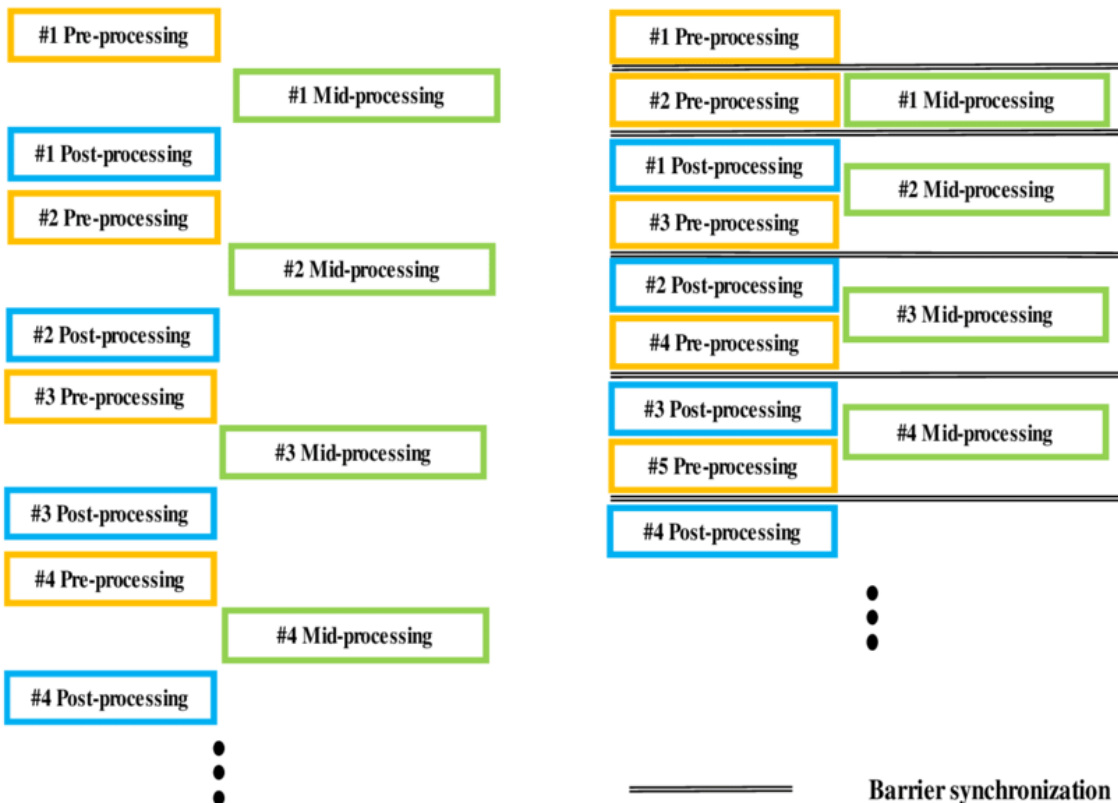
Pipeline processing

CPU

GPU

CPU

GPU

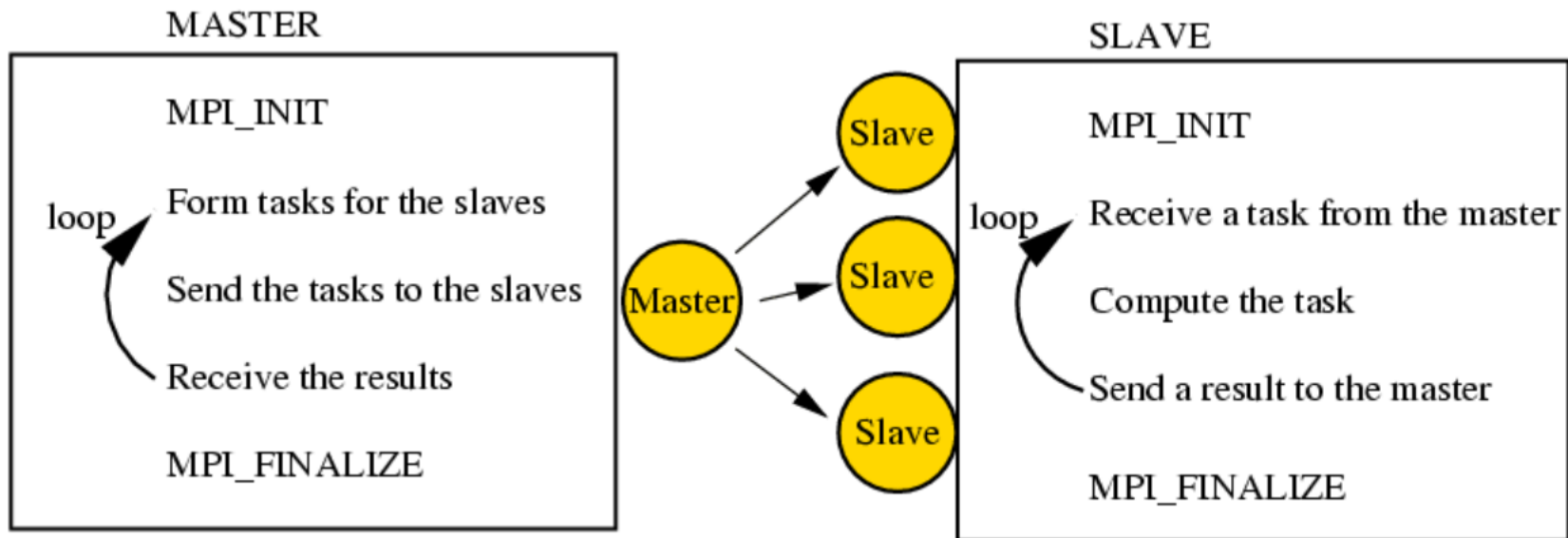


- Review of Parallel Programing
  - ✓ Why Parallel Programing
  - ✓ Why Parallel Code Slower
  - ✓ Improving Programs with Parallelism
  - ✓ The Structure of MPI and Common Error
- Performance Modelling
  - ✓ Why and What is Performance Modelling?
  - ✓ Performance Metrics in Parallel Systems



# The Structure of MPI

- Message passing is well-suited to handling computations where a task is divided up into subtasks, with most of the processes used to compute the subtasks and a few processes (often just one process) managing the tasks.



# The Structure of MPI

## MPI\_init

```
if( rank == root )
{
    // Master's work :
    // Form tasks, distribute tasks,
    // receive and form results
}
```

```
if (rank !=0 )
{
    // Worker's work:
    // receive tasks, compute tasks
    // send back to the master
}
```

```
// Tasks that all processes need to
do
```

## MPI\_Finalize

## Notes:

- **Rank** is used to distinguish processes from one to another.
- E.g. you have 8 parallel processes running; if you query for the current process **rank** via **MPI\_Comm\_rank** you'll get 0-7.
- In basic applications you'll probably have a "master" process on rank = 0 that sends out messages to "worker" processes on rank 1-7.
- For more advanced applications you can divide workloads even further using ranks (i.e. 0 rank master process, 1-4 perform function A, 5-7 perform function B).

# The Structure of MPI- Example

```
if (id == 0)
{
    for (int dest = 1; dest <= numworkers; dest++)
    {
        MPI_Send(&offset, 1, MPI_INT, dest, 1, MPI_COMM_WORLD);
        MPI_Send(&a[offset], 2, MPI_INT, dest, 1, MPI_COMM_WORLD);
        offset += 2;
    }
    for (int dest = 1; dest <= numworkers; dest++)
    {
        MPI_Recv(&offset, 1, MPI_INT, dest, 2, MPI_COMM_WORLD, &status);
        MPI_Recv(&a[offset], 2, MPI_INT, dest, 2, MPI_COMM_WORLD, &status);
    }
}
```

- The master (i.e. process id == 0) circularly sends/receives messages to other processes by using for loop.

# The Structure of MPI- Example

```
else if(id !=0 )
{
    MPI_Recv(&offset, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
    MPI_Recv(&a[offset], 2, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
    for (int i = 0; i < 2; i++)
    {
        int loc = i + offset;
        a[loc] += 1;
    }
    MPI_Send(&offset, 1, MPI_INT, 0, 2, MPI_COMM_WORLD);
    MPI_Send(&a[offset], 2, MPI_INT, 0, 2, MPI_COMM_WORLD);
}
```

- The other processes (i.e. process id != 0) only need to receive and send messages to the master once.

# Point to Point vs Collective

## MPI\_init

```
if( rank == root )
{
    // Master's work :
    // Form tasks, distribute tasks,
    // receive and form results
}
```

```
if (rank !=0 )
{
    // Worker's work:
    // receive tasks, compute tasks
    // send back to the master
}
```

```
// Tasks that all processes need to
do
```

## MPI\_Finalize

- As long as "send" and "receive" are matched, point to point functions can be placed everywhere.
- While, a **collective** function should be executed by all related process.

Therefore, collective functions should be written in here

# Common Errors Writing MPI Code

- Doing things before **MPI\_Init** or after **MPI\_Finalize**
- Matching **MPI\_Bcast** with **MPI\_Recv**
- Only one process executes a collective operation
- Assuming your MPI implementation is thread-safe

# The Structure of MPI- Incorrect Example

- A common **mistake** is using receive function to receive messages from collective function

```
#include "mpi.h"
int main( int argc, char* argv[] )
{
    int rank; int ibuf;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    if(rank == 0)
    {
        ibuf = 12345;
        MPI_Bcast(&ibuf, 1, MPI_INT, 0, MPI_COMM_WORLD);
    }
    else ibuf = 0;
    if (rank != 0 )
    {
        MPI_Recv(&ibuf, 1, MPI_INT, 0, 1, MPI_COMM_WORLD, &status);
        printf("my rank = %d ibuf = %d\n", rank, ibuf);
    }
    MPI_Finalize();
}
```



# The Structure of MPI- Correct Example

```
#include "mpi.h"
int main( int argc, char* argv[] )
{
    int rank;
    int ibuf;
    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    if(rank == 0) ibuf = 12345;
    else ibuf = 0;
    MPI_Bcast(&ibuf, 1, MPI_INT, 0, MPI_COMM_WORLD);
    if (rank !=0 )
    printf("my rank = %d ibuf = %d\n", rank,ibuf);
    MPI_Finalize();
}
```



In addition to usual, “serial” bugs, parallel programs can have “parallel-only” bugs, such as

- race conditions: when results depend on specific ordering of commands, which is not enforced
- deadlocks: when task(s) wait perpetually for a message/signal which never come
- Similarly to debugging, profiling of parallel codes deals both with issues common between serial and parallel codes (bad patterns for accessing memory, not cache friendly etc.), but also adds new, parallel-only issues:
  - ✓ workload balancing
  - ✓ costs of communications

- Review of Parallel Programing
  - ✓ Why Parallel Programing
  - ✓ Why Parallel Code Slower
  - ✓ Improving Programs with Parallelism
  - ✓ The Structure of MPI and Common Error
- Performance Modelling
  - ✓ Why and What is Performance Modelling?
  - ✓ Performance Metrics in Parallel Systems

- Parallelism generally can improve efficiency
  - ✓ How do you know if you are making good use of a system?
- The key questions are
  - ✓ Where is most of the time spent?
  - ✓ What is the achievable performance, and how to get there?
- This 2<sup>nd</sup> question is often overlooked, leading to erroneous conclusions based on the state of compiler/runtime/code implementations

- **Typical Approach**
  - ✓ **Profile code**: determine where most time is spent
  - ✓ **Study code**: measure absolute performance, look at performance counters, compare FLOP rates
  - ✓ **Improve code**: increase FLOP rates or memory accesses
- **Why this typical approach is not enough:**
  - ✓ How do you know when you are done?
  - ✓ What is the maximal improvement you can obtain?
- **Why is it hard to know?**
  - ✓ Many problems are too hard to solve without parallelism
  - ✓ It is getting harder and harder to provide performance without specialised hardware

# Why Performance Modelling?

- **What is the goal of performance modelling?**
  - ✓ It is **not precise** predictions
  - ✓ It is **insight** into whether a program is achieving the performance it could, and if not, how to fix it
- **Performance modeling can be used**
  - ✓ to estimate the **baseline** performance
  - ✓ to estimate the **potential benefit** of a nontrivial change to the code
  - ✓ to identify the **critical resource**

# What is Performance Modeling

- Two different models
  1. an analytic expression based on the **application code**
  2. an analytic expression based on the **application's algorithm and data structures**
- A series of measurements from benchmarks\* are not performance modeling
- Why this sort of modeling
  - ✓ Extrapolation to other systems
    - e.g. scalability in nodes or different interconnect
  - ✓ Comparison of the two models with observed performance can identify
    - **Inefficiencies in compilation/runtime**
    - **Mismatch in developer expectations**

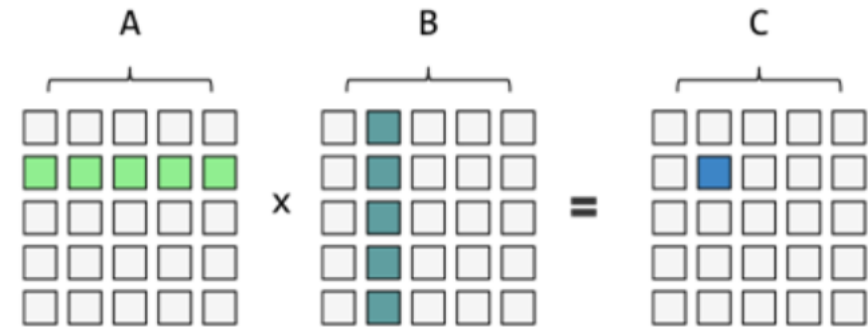
\*In computing, a benchmark is the act of running a computer program in order to assess the relative performance of an object, normally by running a number of standard tests and trials against it.

- Simulation:
  - ✓ Very accurate prediction, little insight beyond specifics of the simulation itself
- Traditional Performance Modelling (PM):
  - ✓ Focuses on accurate predictions
  - ✓ Tool for computer scientists, not application developers
- PM as part of the software engineering process
  - ✓ PM for design, tuning and optimisation
  - ✓ PMs are developed with algorithms and used in each step of the development cycle

# Example

- Lets look at a simple example
- Matrix-matrix multiply
  - ✓ Classic example in parallel computing
  - ✓ Core of the “HPLinpack” benchmark\*
  - ✓ Simple to express:

```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n; j++) {  
        for (int k = 0; k < n; k++) {  
            c[i][j] += a[i][k] * b[k][j];  
        }  
    }  
}
```



\*The [LINPACK Benchmarks](#) are a measure of 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  $Ax = b$ , which is a common task in engineering.



- **How fast should this run?**
  - ✓ Standard complexity analysis in numerical analysis counts floating point operations
  - ✓ The matrix-matrix multiply algorithm has  $2n^3$  floating point operations
    - 3 nested loops, each with  $n$  iterations
    - 1 multiply, 1 add in each inner iteration
  - ✓ For  $n=100$ , there are  $2 \times 10^6$  operations, or about **1 msec** on a 2GHz processor:  $2 \times 10^9$  operations per **sec**.
  - ✓ For  $n=1000$ ,  $2 \times 10^9$  operations, or about **1 sec**

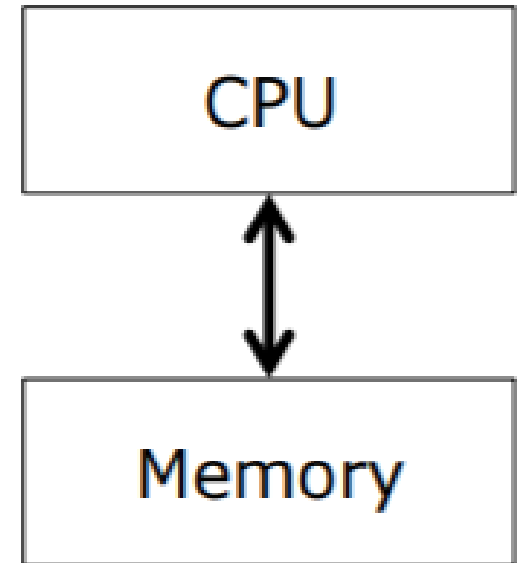
```
for (int i = 0; i < n; i++) {  
    for (int j = 0; j < n; j++) {  
        for (int k = 0; k < n; k++) {  
            c[i][j] += a[i][k] * b[k][j];  
        }  
    }  
}
```

# The Reality vs Theoretical Results

- $n=100$ : 1.1ms
- $n=1000$ : 6s
- What this tells us:
  - ✓ obvious expression of algorithms are not transformed into leading performance.
- **How fast should this run?**
  - ✓ ...
  - ✓ For  $n=100$ ,  $2 \times 10^6$  operations, or about 1 msec
  - ✓ For  $n=1000$ ,  $2 \times 10^9$  operations, or about 1 sec

# Thinking about Performance

- The performance model assumes the computer looks like the figure on the right
  - ✓ Memory is infinitely large
  - ✓ Memory is infinitely fast
- The performance models can be improved by adding features to model the computer hardware
- In the first enhancement, lets make memory **not infinitely fast**




# A Simple Performance Model

- Use the following:
  - ✓ Number of operations (e.g. floating point multiply)
  - ✓ Number of loads from memory
  - ✓ Number of stores to memory
- This model ignores many features of an architecture that are used to optimise performance (e.g. cache)
- Consider the following code:
  - ✓  $2n$  operations (i.e. floating add, floating multiply)
  - ✓  $2n$  **loads/reads** (i.e.  $x[i]$  and  $y[i]$  for  $i=1$  to  $n$ )
  - ✓  $n$  **stores/writes** (i.e.  $y[i]$ )

```
for (int i=1; i<n; i++ )
{
    int i=1;
    y[i] = a*x[i] + y[i];
}
```

- Assume that
  - $c$  = time for operation
  - $r$  = time to load/read an element
  - $w$  = time to store/write an element
- Then a very crude estimate of the time for this operation is
  - ✓  $T = n(2c + 2r + w)$
- Call this a model because it is too crude to be an estimate

Memory Specifications	
Max Memory Size (dependent on memory type)	8 GB
Memory Types	DDR3 1066/1333
Max # of Memory Channels	2
Max Memory Bandwidth	21.3 GB/s
ECC Memory Supported <sup>†</sup>	 No

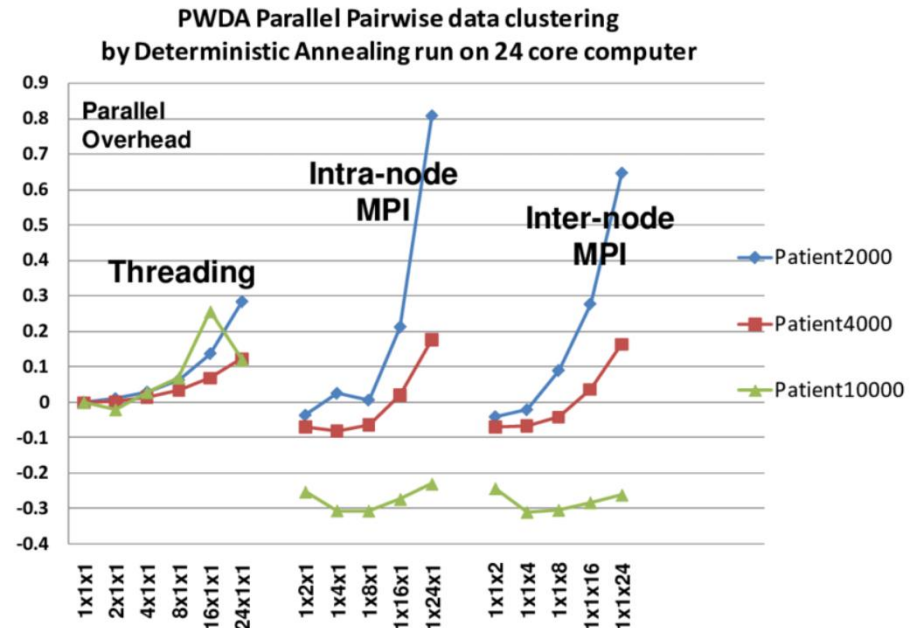
# Some Comments on This Model

- many analysis of algorithms set  $r$  and  $w$  to zero
- different ways to model communication time
  - ✓ load and store to memory
  - ✓ sharing of data between threads
  - ✓ communication between nodes in a parallel computer
  - ✓ load and store to a file system
- more general analytical modelling
  - ✓ consider sources of overhead: inter-process communication, idling, excess computation

- Review of Parallel Programing
  - ✓ Why Parallel Programing
  - ✓ Why Parallel Code Slower
  - ✓ Improving Programs with Parallelism
  - ✓ The Structure of MPI and Common Error
- Performance Modelling
  - ✓ Why and What is Performance Modelling?
  - ✓ Performance Metrics in Parallel Systems

# Performance Metrics for Parallel Systems

- Execution time
  - ✓ parallel run time
  - ✓ double `MPI_Wtime()`
- Total parallel overhead
  - ✓ overhead function
  - ✓  $T_0 = pT_p - T_s$



$T_p$  : parallel execution time

$T_s$  : serial execution time

$P$  : the number of processors



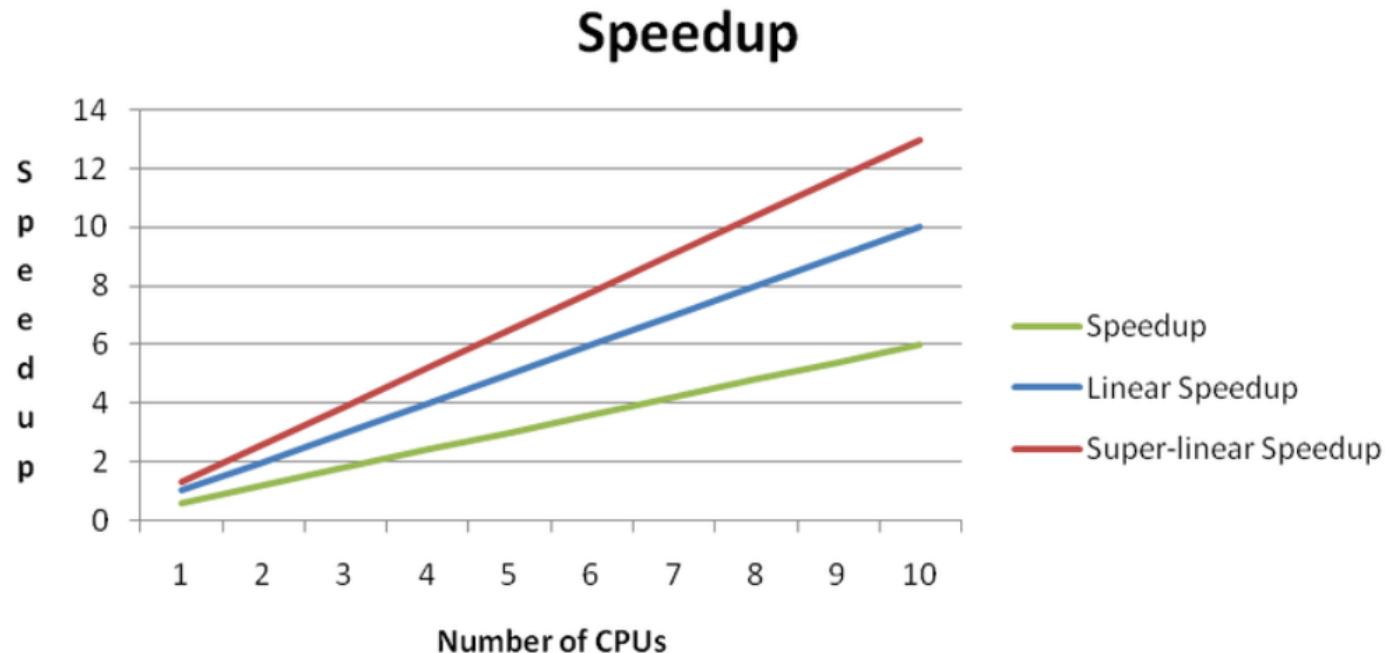
# Performance Metrics: Speedup

- **Speedup** (denoted by  $S$ )
  - ✓ ratio of serial execution time to parallel execution time
- Example: Summing  $n$  integers with  $n$  processes
  - ✓  $T_p = \Theta \log(n)$ ,  $T_s = \Theta(n)$ ,  $S = \Theta \frac{n}{\log(n)}$
- Can also be computed empirically with timing values
- Remember to compare to **best known sequential algorithm**
- Speedup can never be better than  $T_s/p$ , unless algorithmic optimisations have been done.



# Performance Metrics: Speedup

- Super-linear speedup
  - ✓ should be impossible, seems contradictory
- Occurs when hardware provides an extra advantage to parallel formulation
  - ✓ e.g. cache usage or vector units



- Efficiency: only a perfect system would be a speedup of  $p$ 
  - ✓ the fraction of time for which processing is usefully employed, speedup to the number of processing elements

$$E = \frac{s}{p} = \frac{T_s}{pT_p}$$

- **Cost** quantifies the amount of resource needed to achieve a particular performance
- A system is **cost-optimal** if the parallel formulation has the same asymptotic growth as a function of input size as the fastest known sequential algorithm
- **Increased** granularity results in **lower** cost if overheads are reduced

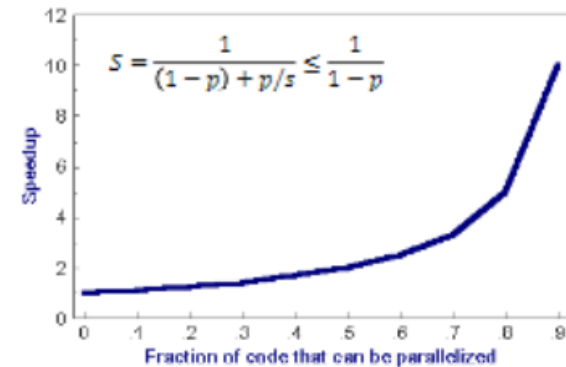
- Amdahl's Law (1967)
  - ✓ Gives the maximal theoretical speedup for a fixed workload
- Assumes total workload  $W$  with serial component  $W_s$
- $S(s) = \frac{w}{w_s} = \frac{1}{1-q+\frac{q}{s}}$ , where  $s$  is the speedup of the parallelisable portion  $q$  of the job
- Proof:

$$w_s = (1-q) w + \frac{qw}{s} \quad \Rightarrow \quad S(s) = \frac{w}{w_s} = \frac{1}{1-q+\frac{q}{s}}$$

# Performance Metrics: Amdahl's Law

Example: if we can do 30% of a task **four times** as fast

$$S(s) = \frac{1}{1 - 0.3 + \frac{0.3}{4}} = 1.29$$



N	speedup		
	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02

Shows that maximal speedup is always limited by what is **not parallelised**

## Gustafson's Law

- More optimistic and realistic than Amdahl's Law
- Estimates speedup with respect to execution time
- $S(s) = N + (1-N)(1-q)$ 
  - ✓ where N is the number of processors, q is the parallelisable portion of the job.
- Based on the observation that compute resources scale with problem size

- Readings
  - [CITS5507 Teaching Materials \(2019\)](#)
  - [Designing and Building Applications for Extreme Scale Systems](#)



## Copyright Notice

Material used in this recording may have been reproduced and communicated to you by or on behalf of **The University of Western Australia** in accordance with section 113P of the *Copyright Act 1968*.

Unless stated otherwise, all teaching and learning materials provided to you by the University are protected under the Copyright Act and is for your personal use only. This material must not be shared or distributed without the permission of the University and the copyright owner/s.