# Lecture 1: Introduction

We use C and C++ programming language.

Parallel computing examples:

* **Desktops**
* **Laptops**
* **Smartphones (since they come with multiple cores)**

Parallel processing examples:

* **Hardware**: Capable of doing parallel processing
* **Networks**: Allow multiple processors to communicate with each other
* **Operating systems**: can handle parallel processing
* **Parallel libraries**
* **Languages**
* **Compilers**
* **Algorithms**
* **Tools**

These all allow us to write parallel programs

Aspects of serial computing still apply to parallel computing such as:

* **Data structures**
* **Algorithms**
* **Accuracy**
* **Efficiency**

Also has extra things that serial computing does not care about such as:

* **Synchronization**
* **Communication**

Parallel computing characteristics:

If a problem has a subproblem that is dependent on it (one of the subproblems need the results of another problem) then these 2 subproblems cannot be processed in parallel as they are inherently sequential

**Performance-> technology used**

* **Wider data paths**- increased size of chips
* **More efficient pipelining**- execute more instructions per cycle (from 3.5 cycles-> 1 instruction to 1.1 cycles-> 1 instruction)
* **Exploiting instruction level parallelism**
* **Faster clock rates**

Greater clock frequency= greater clock power= greater heat

We combat this problem with more cores

If the growth rate of chip performance was compared to car performance then the speed of a car should be 1/10 of light speed (speed of light). Therefore cannot be related to physical world)

Memory speed not growing at same rate as CPU performance. Growing performance gap between performance + memory. Solving problem could be limited by memory performance instead of faster processor performance.

To address this performance gap is to add faster memory on the processor chip so that the data and processor are physically close i.e. faster data access.

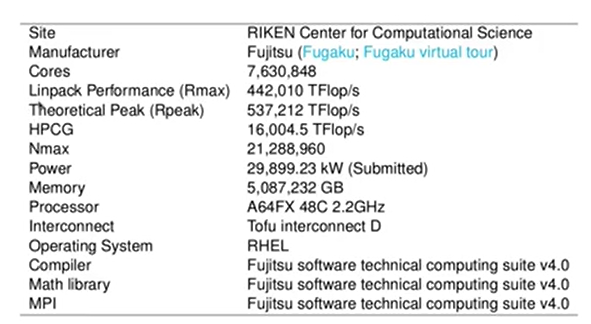
Chip area-> die area

With increased number of transistors in the chips, greater clock frequency means greater electrical power, which results in greater heat.

The solution to this is to add more cores to increase performance whilst keeping clock rate the same or reduced.

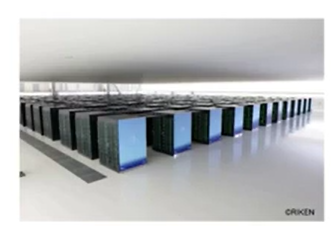
Supercomputers

Every year there are 2 supercomputer conferences which rank the world’s top 500 supercomputers, one in June, the other in November.



The list gives some of the specs of the computer that was voted the number 1 in 2020 November.

* **TFlop/s**= terraflops
* **Linpack benchmark**- measure of a computer’s rate of floating point execution
* **Theoretical peak> linpack performance** means that benchmark performance unable to utilise its full theoretical power, but few applications can meet the theoretical peak performance anyway
* **HPCG (High performance conjugate gradient)** is less parallelisable in terms of linear algebra problems which is why it is less than the linpack performance.
* **Nmax**- indicates problem size in order to achieve linpack performance. Around 21 million variables to be solved in this example
* **Power** is the power consumption of the supercomputer. Can estimate cost of running such a computer
* **Memory** in the example is 5PB (Petabytes) which is the 5x 10^15
* **Type of processors/ core** is ARM based core. Smartphones use ARM cores. Clock speed of core is the GHz number
* **Interconnect** needs to be fast as they are arranged in nodes etc. This one was specially designed for Fugaku
* **Operating system** is Red Hat Enterprise Linux
* The **Compiler, Math library and MPI** were all custom built for the supercomputer.



Consists of racks, in this case the supercomputer has 432 racks.

Each rack has a number of nodes, contains 368 nodes.

Each node has 48 cores

This makes up the millions of cores

For this type of computer to program, you will have to use parallel programming languages, tools, libraries, compilers, as well as interconnect networks.

What Fugaku (10 year project from conceptualisation to completion) is being used for and expected to run applications in areas such as:

* **Job discovery**
* **Personalised and preventive medicines**
* **Simulations of natural disasters**
* **Weather and climate forecasting**
* **Energy creation storage and use**
* **Development of clean energy**
* **New material development**
* **New design and production process**

Fugaku is already being deployed for use of research on Covid-19 including:

* **Diagnostics**
* **Therapeutics**
* **Simulations of the spread of the virus**

Our clusters at MSL:

* **Stampede**
* **Batch**
* **Biggpu**
* **Ranger**
* **Mia**

Some not accessible to students. Primary for research, for postgrad students for running big jobs.

Goal is to get excess scale processing speed.

A Quantitative Look at Parallel Computation

One of the main reasons for implementing a parallel program is to obtain better performance. We measure its performance against its serial counterpart as well as other parallel implementations to determine whether a parallel solution is giving us any performance gain.

A parallel computer’s performance can be both marbled and measured.

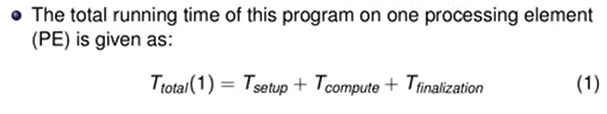
A simple performance modelling

Setup can be any initialisation including declaring variables, allocating memory spaces or initialising variables and so on.

Computation represents bulk of computation which is the main part of the program

Finalisation usually cleans up memory spaces used during the execution etc.

Total running time on a single processing element can be achieved by the summation of the times for the three parts respectively: setup, computation and finalisation.

The 1 represents that it is running on 1 processing element, which could be a single core.

With multiple Processing Elements (Pes), the full computation on P number of processing elements can be obtained by equation 2:



The setup and finalisation times are the same for both 1 and 2, which means that additional processing elements are actually useless for these sections in a program. These sections are usually not parallelisable. Only computation is usually parallelised and the time should be reduced by the factor/ number of processing elements being used. This is under the assumption that the computation problem can be parallelised in an ideal way.

Speedup

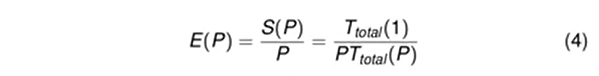
An important measure of how much multiple processing elements help with the performance of a parallel program is the relative speedup S which describes how much faster a parallel program runs regardless of the actual running time.

Speedup is simply the serial time over the parallel time. These are from equations 1 and 2.



A closely related measure is the **efficiency E** which is the ratio between speedup and the number of processing elements we used to achieve that speedup. Using efficiency we can measure how well we utilise the resources available in our parallel program. It is a value between 0 and 1.

0<E < 1



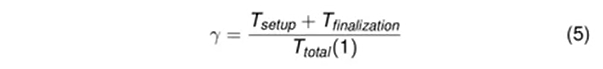
**Perfect Linear Speedup**: Ideally, we would want the speedup to be equal to P, the number of Processing Elements being used. This is rarely the case as the times for setups and finalisation using multiple processing elements won’t help with ? time.

Try to make parallel program as efficient as possible- as close to 1 as possible.

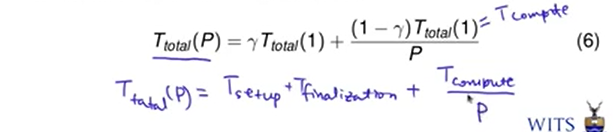
Amdahl’s Law

We use gamma to represent the term in a program that cannot be run in parallel and we call this serial fraction.

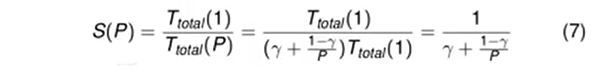
The two parts that cannot be parallelised (setup and finalisation) are combined, which become the serial fraction in our simple model.



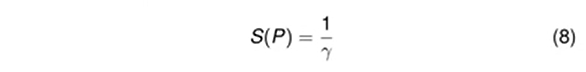
The parallelisable fraction in the problem is simply 1- gamma. We can then rewrite the total parallelisable time as equation 6. The following 2 equations below for total parallel time are equivalent



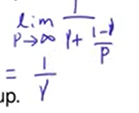
Therefore Amdahl’s law can be derived as follows for speedup:



Amdahl’s law says that the speedup we can achieve is limited by the serial fraction.

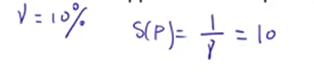


If we have an unlimited number of processing elements (approaches infinity) then the speedup will simply result in the following:



The upper bound of speedup you can achieve is actually limited by the serial fraction, not by the number of processing elements you can use. The speedup you can do will never exceed you S(P)

e.g.



The implicit assumption of Amdahl’s law: Even if we increase the number of processing elements, you will not be changing the problem size.

Unrealistic in some circumstances, as if we have more processing elements, we would also want to increase the problem size. Amdahl’s law has a fixed problem size with increase in number of processing elements.

Gustafson’s Law

We want to derive a similar expression for speedup but assuming that the problem size will change if we change the number of processing elements.

We will now derive the total serial time from the total parallel time. To do that, we compute the total parallel time first.



Scaled serial fraction: serial time over total parallel time



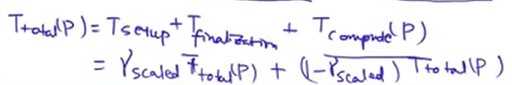
Parallelisable fraction is 1- gamma scaled

(1-gamma scaled) Ttotal(P) = Tcompute(P)

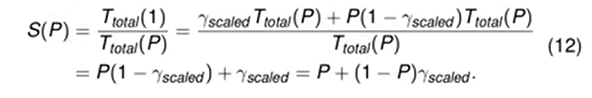


Equation 9 and 11 are equivalent

Total parallelisable time is calculated as follows:

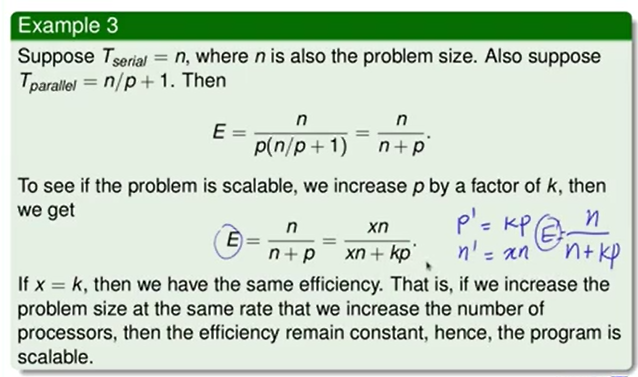


Gustafson’s Law (another speedup)



Types of scalability

* **Strong scalability-** corresponds to Amdahl’s Law
* **Weak scalability-** corresponds to Gustafson’s Law



This program is weakly scalable

My understanding of FLOPs:

Graphical user interface, text, application

Description automatically generated