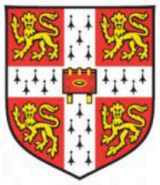


UNIVERSITY OF CAMBRIDGE

Advanced Research Computing

MPhil in Data Intensive Science - Trial Year

Dr James Fergusson and Dr Amelia Drew

Prerequisites

- **Research Computing Topics**
  - Linux and Bash
  - Python
  - Best Practices in Design and Development
  - Performance (profiling, optimisation)
  - Multi-language Programming
  - Public Release (including Docker)

Course Content

- **Advanced Programming**
  - Interpreters and Compilers
  - Review of C++
  - Overview of Fortran
  - Compilation (Makefiles, build automation tools)
  - Debugging
  - Profiling
  - Running on Supercomputers
  - Advanced Build and CI Management
  - Co-ordinating Large Software Projects

Course Content

- **Parallelisation**
  - Supercomputer Architectures and Overview of Parallelisation
  - Predictive Performance Models - Roofline, Threads and Processes
  - Vectorisation
  - OpenMP and MPI
  - Debugging
  - Parallel I/O
  - GPUs and Heterogeneous Programming (Introduction to DPC++)
- **Visualisation**
  - Advanced Visualisation Approaches
  - Paraview

Course Content

LUMI Supercomputer, Finland
Largest in Europe (TOP500)

Course Content

- If you would like to test out the material on your laptop, please do the following:
  - Download VSCode (https://code.visualstudio.com/)
  - Download and configure Docker
  - Pull Docker file from my GitHub page (https://github.com/amelialdrew/advanced_research_computing_docker)
  - Build Docker image in VSCode and run

Advanced Programming

Interpreters and Compilers

Advanced Programming

Interpreters and Compilers

- *Example:* we would like to write a computer program to add two numbers together
- We first pick a programming language that suits our needs best eg. C++, Python

python Copy code

```python
# Ask the user for the first number
num1 = input("Enter the first number: ")

# Ask the user for the second number
num2 = input("Enter the second number: ")

# Add the two numbers together
result = float(num1) + float(num2)

# Print the result
print("The sum of", num1, "and", num2, "is", result)
```

Advanced Programming

Interpreters and Compilers

- Computers cannot understand these languages directly - they can only execute *machine code*
- Each computer processor (different CPUs, GPUs etc.) has its own machine language
- The processor requires certain instructions which tell it eg. to find an address in memory and perform an operation on it
- Looks something like:

```
01001000 01100101 01101100 01101100 01101111 00100001
```

Advanced Programming

Interpreters and Compilers

- It would be very difficult to write this program in machine (eg. binary) code…
- To be able to program with any efficiency, we need a way to translate between the human-readable code, and the binary machine code
- This is primarily performed in two ways:
  - Use an **interpreter** - these go through code line by line and interpret each command for the processor so that it can be executed
  - Use a **compiler** - these translate code, check for errors and create an ‘executable,’ which can then be run by the processor. This is usually the fastest method

Advanced Programming

Interpreters and Compilers

- So far, you are familiar with Python (taught in the Research Computing module)
  - It is recommended that you use a *virtual environment* for Python projects
  - This is a *folder structure* that allows you to run a Python environment in a way that:
    - Any packages installed in your operating system's (OS's) global Python installation are not mixed up/overwritten
    - Updating your OS does not affect running your scripts
    - You can easily switch between Python versions/dependencies (eg. usually you cannot have two versions of the same library if you have one place to install packages)

Advanced Programming

Interpreters and Compilers

Continued..

- Recommended reading: https://realpython.com/python-virtual-environments-a-primer

- In our examples, we use a Docker image of Ubuntu with Python installed via pip - for more flexibility with versions, we would ideally use a virtual environment *within* the Docker container

```
# Get the base Ubuntu image from Docker Hub
FROM ubuntu:latest

# Update apps on the base image
RUN apt-get -y update && apt-get install -y

# Install Python
RUN apt-get -y install python3-pip

# Install iPython
RUN pip3 install ipython
```

Advanced Programming

Interpreters and Compilers

- Python is a high-level language that can be run directly using an **interpreter** (CPython)
- If using a virtual environment, the interpreter will be stored in `bin/`
- The interpreter performs certain steps:
  - Checks code for syntax and errors
  - Converts Python code into ‘byte code’
  - Initialises a *virtual machine* that converts byte code into binary code and executes
- For a Python script `test.py`, we run using the command `python3 test.py` (where `python3` is the command to call the interpreter)

Advanced Programming

Interpreters and Compilers

Practical Task 1:

- Run the test code from previous slide (in Docker container)

Practical Task 2:

- Start `iPython` on your laptop with the docker image - this invokes the interpreter so you can interact with it
- Try out some simple command eg. `1+2`, return
- The interactive interpreter executes the statement

Advanced Programming

Interpreters and Compilers

- C++ is another high-level language that must be **compiled** in order to run
- Like an interpreter, a **compiler** takes human-written code and converts it to be machine-readable
- The main difference to interpreters is that compilation is performed *in advance* of running the code
- We must install a **C++ compiler** on our machine (sort of the equivalent of installing Python)
- This can be done eg. on Mac, via Homebrew

Advanced Programming

Interpreters and Compilers

- Like an interpreter, the compiler performs certain steps:
  - Preprocesses source files, eg.
    - Inserts content from *header files* added via `#include`. These are replaced with entire content of included file.
    - Any `false` conditional compilation sections removed (`#ifdef, #endif`)
    - This creates a *translation unit*

Advanced Programming

Interpreters and Compilers

Continued..

- Compiles *object file* from the preprocessed translation unit
  - This is a machine code file
  - References to functions, symbols used are not yet defined - they have no memory address
- Object files are then *linked* into an executable
  - We must link any object file that specifies eg. a function, to the file in which the function is defined

- Unless you use specific compilation flags, all of these steps will be done automatically
- However, it is useful to know the steps to be able to diagnose compilation errors

Advanced Programming

Interpreters and Compilers

Example:

- We write some source code for a ‘Hello World’ program, `hello_world.cpp`

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

- There is no interpreter to run this directly, so we must compile first

Advanced Programming

Interpreters and Compilers

- We install a C++ compiler called `g++` and run the command `g++ hello_world.cpp -o hello_world`
- This will give us an ‘executable’ file called `hello_world` - without the `-o` option, the executable is given some default name `a.out`
- We can then run this executable using the command `./hello_world`

Advanced Programming

Interpreters and Compilers

Practical Task:

- Compile and run the `hello_world.cpp` in the Docker container
- Create and examine the *translation unit,* using `g++ -E hello_world.cpp -o whatever_preproc_name`
  - You can count the number of lines using `wc -l whatever_preproc_name`
- Create and examine the *object file,* using `g++ -c hello_world.cpp` (will create `hello_world.o`)
  - You can use the command `nm hello_world.o` to see which symbols are specified (they are not yet linked)

Advanced Programming

Interpreters and Compilers

- To recap - what has actually happened here?
    1. Compiler replaces `#include <iostream>` with entire content of included file
    2. Source file compiled into a translation unit, then an ‘object’ file
    3. Object file has been linked with the file that contains the `std::cout` and `std::endl` functions
    4. The code has been run
- We now have an idea of how interpreters and compilers work with different programming languages

Advanced Programming

Review of C++

Advanced Programming

Review of C++

Some recommended books:

- *Accelerated C++*, A. Koenig and B. E. Moo (slightly outdated but a good approach)

- Books by Bjarne Stroustrup, e.g. *Principles and Practice Using C++* and *The C++ Programming Language 4th Edition* (includes C++11 concepts)

- *Moving Planets Around - An Introduction to N-Body Simulations Applied to Exoplanetary Systems*, J. Roa et al. (how to write an N-body code)

Websites:

- http://en.cppreference.com

- ChatGPT or https://stackoverflow.com/ (general questions)

Advanced Programming

Review of C++

Aim:

- To refresh your knowledge of, or to learn, C++
- Primarily cover basic concepts, with some more advanced
- Led by examples and by what you are likely to need to do as a researcher/developer:
  - Analysing data
  - Using other people's code

Advanced Programming

Review of C++

- C++ is a language developed in 1979 by Bjarne Stroustrup
- The first C++ ‘standard’ was developed in 1998
- It is an *object-oriented* language (we will unpack this later)
- Changes/additions to the standard are made as new C++ versions appear (C++98, C++03, C++11, C++14, C++17, C++20)
- The current most up-to-date version is C++20, but in practice, actively developed codes are probably using up to C++14/17
- New versions usually provide new advanced features

Advanced Programming

Review of C++

Advantages:

- C++ can be optimised well by modern compilers - it is **fast**
- Object-oriented approach means code is (in theory) easy to read
- The basic features are relatively intuitive

Disadvantages:

- In large codes, object-orientation can mean you must go down ‘rabbit holes’ to work out how the code works
- Can be complex if you want to implement advanced features

Advanced Programming

Review of C++

- We have written some code to output the text, ‘`Hello, world!`’, to the terminal (using ChatGPT)
- What does this program demonstrate?

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

Advanced Programming

Review of C++

- C++ is an *object-oriented programming language* - it is organised around ‘objects’ (in contrast to eg. functional programming)
- Each object will have a defined *type* - these can be already built in (`int`, `double` etc.) or user-defined (`class`)
- The type defines permitted operations and a ‘semantic meaning’ to the object
- You will often see ‘an object is an *instance* of a *class’* - but also true for built-in types
- Objects enable code *abstraction* - unnecessary implementation code can be hidden
- Other codes that mainly use object orientation are Python, Java

Advanced Programming

Review of C++

- In `hello_world.cpp` example, `cout` is an *object* of the *class* `ostream` - it is an object of *type* `ostream`
- Type `ostream` is defined by the external library, `<iostream>`
  - https://cplusplus.com/reference/iostream/
- The `0` returned by the `main()` function is also an object of type `int`

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

Advanced Programming

Review of C++

- The most used built-in C++ data types are:
  - `int` - integer (from -2147483648 to 2147483647, 2-4 bytes memory)
  - `float` - floating point (decimals and exponentials, 4 bytes memory)
  - `double` - floating point with double precision (8 bytes)
  - `bool` - boolean (`true` or `false`, used for conditional statements, 1 byte)
  - `void` - valueless - only used for functions that do not return any data
  - `char` - character (1 byte)
  - [`wchar_t` - wide character (character that takes up more than one byte)]

Advanced Programming

Review of C++

In example program:

- `int` (integer) is a built-in type
- `'Hello, world!'` is made up of several `char`

```
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

Advanced Programming

Review of C++

- Every C++ program must have a `main()` function that returns `0` when successful
- This is what is called (by the C runtime library initialisation code) when you run the code
- We note the `#include <iostream>` - this is an external library

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

Advanced Programming

Review of C++

- In the example, `std` refers to the ‘standard *namespace*’
- Namespaces are collections of related names
- `std` is used by standard libraries to contain all the names that it defines, eg. `std::cout`
- If we decided to use a different namespace, we could have another function eg. `otherlib::cout`

Advanced Programming

Review of C++

- Now that we know how to output text to the terminal, we might like to write a program for a scientific purpose
- Eg. We would like to analyse some data in a text file, `data.csv`

Practical Task: Use ChatGPT to create a C++ program that calculates the average value of the second column of a csv file (comma separated values)

Advanced Programming

Review of C++

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <sstream>
#include <string>

double calculateAverage(const std::vector<double>& values) {
    double sum = 0.0;
    for (size_t i = 0; i < values.size(); i++) {
        sum += values[i];
    }
    return sum / values.size();
}

int main() {
    std::ifstream file("data.csv"); // Replace "data.csv" with the path to

    if (!file) {
        std::cerr << "Failed to open the file." << std::endl;
        return 1;
    }
```

```
    std::vector<double> columnData;
    std::string line;

    while (std::getline(file, line)) {
        std::istringstream ss(line);
        std::string cell;

        // Split the line into cells using comma as the delimiter
        std::getline(ss, cell, ','); // Skip the first column
        std::getline(ss, cell, ','); // Read the second column

        // Convert the cell value to double and store it in the vector
        double value;
        std::istringstream(cell) >> value;
        columnData.push_back(value);
    }

    file.close();

    if (columnData.empty()) {
        std::cerr << "No data found in the second column." << std::endl;
        return 1;
    }

    double average = calculateAverage(columnData);
    std::cout << "Average value of the second column: " << average << std::e

    return 0;
}
```

Advanced Programming

Review of C++

- We have defined a new function, `calculateAverage()`
  - What type of output will this function give?
  - This function is called in `main()` and requires a `vector` input

```
double calculateAverage(const std::vector<double>& values) {
    double sum = 0.0;
    for (size_t i = 0; i < values.size(); i++) {
        sum += values[i];
    }
    return sum / values.size();
}
```

Advanced Programming

Review of C++

- This introduces the concept of a *derived data type* - a type created by combining built-in data types
  - `function` - a code segment for a specific purpose, saves us from having to duplicate code

  eg. `sometypea calculateAverage(sometypeb parameters) {`

  `//Content of file`

  `}` (types can be the same, but don't have to be)

  called by

  `calculateAverage(myparams)`

Advanced Programming

Review of C++

- `array`
  - Set of elements of the same type kept in memory in a continuous way
  - Allows us to store data with a single variable name, in sequence
  - Need to know the number of elements before you define

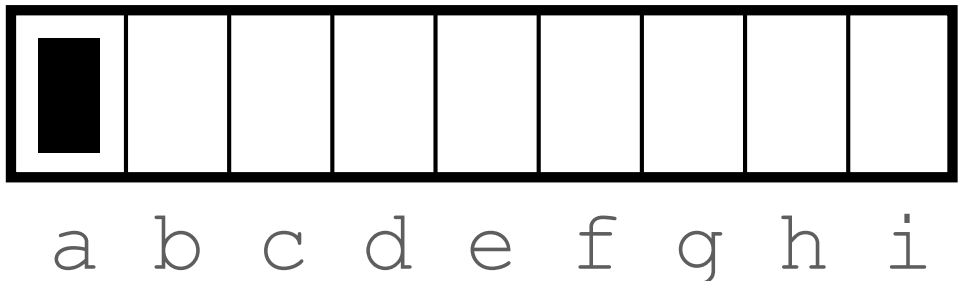
eg. `int time_in_seconds[5] = {0,1,2,3,4};`

Advanced Programming

Review of C++

- `pointer *`
  - a variable that holds the address in memory that another variable is stored
- `reference &` (address-of)

eg.
```
int  black_box;
int* pointer_to_box = &black_box;
```
(the value is `a`)



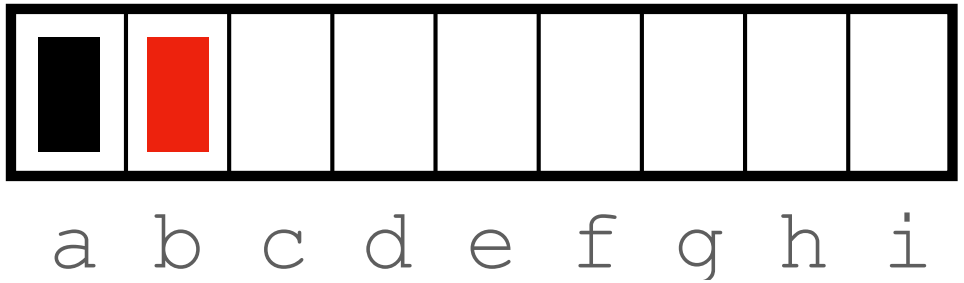
Advanced Programming

Review of C++

Note, `*` can also be used to *dereference* pointers

eg. `int red_box = *pointer_to_box;`

- Here, `red_box` is an *integer* set equal to ‘the variable pointed to by `pointer_to_box`’
- The new variable will be stored in a new memory address, `b`
- When passed as a function parameter, this is known as ‘passing by value’

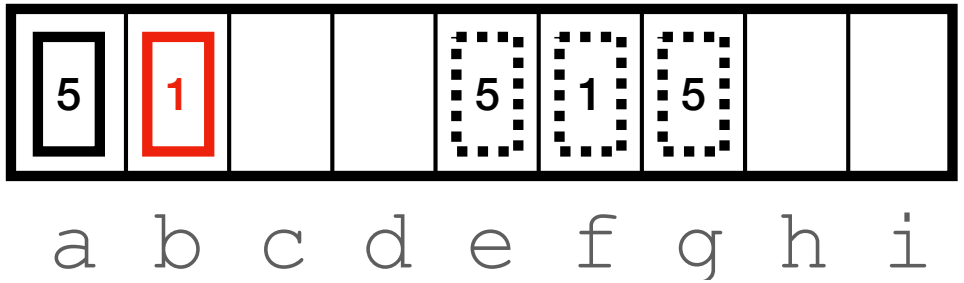


Advanced Programming

Review of C++

Example: what is returned here?

```
void swap(int x, int y) {
    int temp = x; //eg. stored in location e
    x = y;        //eg. stored in location f
    y = temp;     //eg. stored in location g
}
int main() {
    int black_box = 5;
    int red_box = 1;
    swap(black_box, red_box);
    return 0;
}
```



Advanced Programming

Review of C++

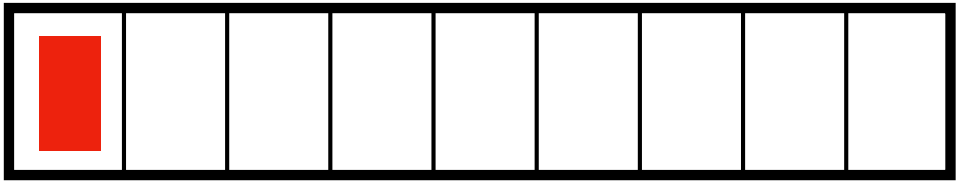
Additionally, `&` can be used to pass references to variables

eg. `int& red_box = black_box;`

- We read this as, ‘the address in memory that points to `red_box` is equal to the address in memory that points to `black_box`’ (which we saw earlier was `a`)

- The variable stored in the location `a` will be effectively aliased by a new variable, pointed to by same memory location

- When passed as a function parameter, this is known as ‘passing by reference’

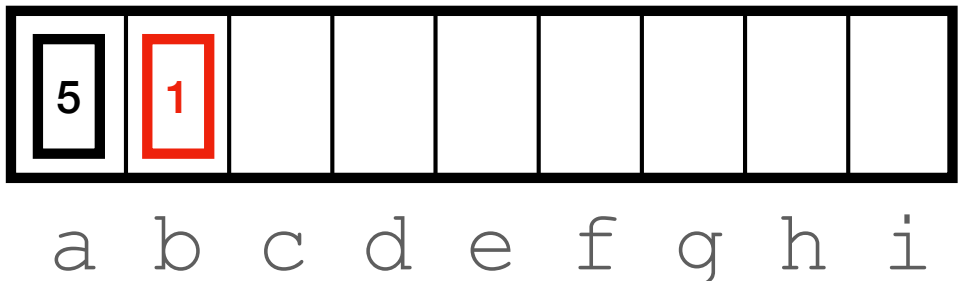


Advanced Programming

Review of C++

Example: what is returned here?

```
void swap(int& x, int& y) {
    int temp = x;
    x = y;
    y = temp;
}
int main() {
    int black_box = 5;
    int red_box = 1;
    swap(black_box, red_box);
    return 0;
}
```

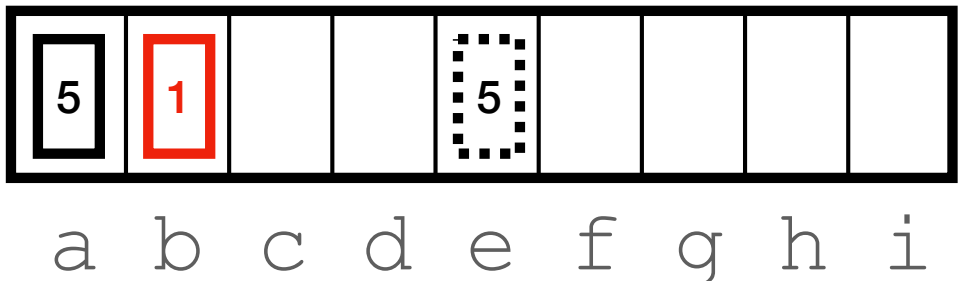


Advanced Programming

Review of C++

Example: what is returned here?

```
void swap(int& x, int& y) {
    int temp = x;
    x = y;              //Read x, y as 'the variable
    y = temp;             pointed to by this address'
}
int main() {
    int black_box = 5;
    int red_box = 1;
    swap(black_box, red_box);
    return 0;
}
```

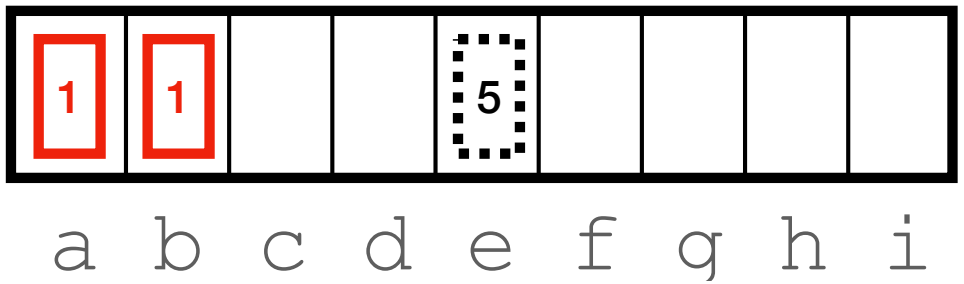


Advanced Programming

Review of C++

Example: what is returned here?

```
void swap(int& x, int& y) {
    int temp = x;
    x = y;              //Read x, y as 'the variable
    y = temp;             pointed to by this address'
}
int main() {
    int black_box = 5;
    int red_box = 1;
    swap(black_box, red_box);
    return 0;
}
```

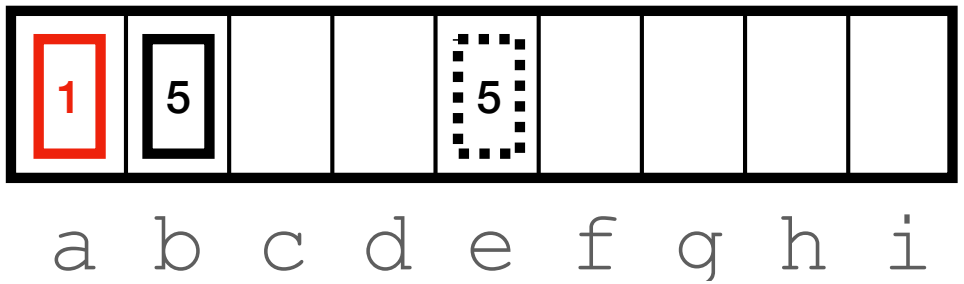


Advanced Programming

Review of C++

Example: what is returned here?

```
void swap(int& x, int& y) {
    int temp = x;
    x = y;              //Read x, y as 'the variable
    y = temp;             pointed to by this address'
}
int main() {
    int black_box = 5;
    int red_box = 1;
    swap(black_box, red_box);
    return 0;
}
```



Advanced Programming

Review of C++

- Pointers and references in particular can take a lot of repetition to get your head around
- Advise keeping good notes that you can refer to while you are getting familiar with the concept (and for when you forget)
- Passing by reference vs by value becomes *very important* with large datasets
- If you have gigabytes or terabytes of data eg. on a time-evolving 3D simulation grid, you do NOT want to be copying unnecessarily
- Passing by reference is the most memory- and time-efficient, especially for large simulations

Advanced Programming

Review of C++

- In our example, we see that the function `calculateAverage()` is passed a `vector` *by reference*
- This means that no copies of `values` will be made in memory
- We have also used `const`, which additionally ensures that the values will not be changed

```
double calculateAverage(const std::vector<double>& values) {
    double sum = 0.0;
    for (size_t i = 0; i < values.size(); i++) {
        sum += values[i];
    }
    return sum / values.size();
}
```

Advanced Programming

Review of C++

- `<vector>` is included as an external library (https://en.cppreference.com/w/cpp/container/vector)
- This is a *class template* (see later) for sequence containers that allow for *dynamic size arrays*
- Vectors are useful if you want to store variables, but you do not know how many there will be before you run the code
- Eg. output from a simulation that depends on runtime parameters
- Some particularly useful functions are `size` (in example, this measures the size of the vector), `push_back` (add an element to the end of the vector), `insert`

Advanced Programming

Review of C++

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <sstream>
#include <string>

double calculateAverage(const std::vector<double>& values) {
    double sum = 0.0;
    for (size_t i = 0; i < values.size(); i++) {
        sum += values[i];
    }
    return sum / values.size();
}

int main() {
    std::ifstream file("data.csv"); // Replace "data.csv" with the path to

    if (!file) {
        std::cerr << "Failed to open the file." << std::endl;
        return 1;
    }
```

```
    std::vector<double> columnData;
    std::string line;

    while (std::getline(file, line)) {
        std::istringstream ss(line);
        std::string cell;

        // Split the line into cells using comma as the delimiter
        std::getline(ss, cell, ','); // Skip the first column
        std::getline(ss, cell, ','); // Read the second column

        // Convert the cell value to double and store it in the vector
        double value;
        std::istringstream(cell) >> value;
        columnData.push_back(value);
    }

    file.close();

    if (columnData.empty()) {
        std::cerr << "No data found in the second column." << std::endl;
        return 1;
    }

    double average = calculateAverage(columnData);
    std::cout << "Average value of the second column: " << average << std::e

    return 0;
}
```

Advanced Programming

Review of C++

- We also see a `for` loop
- This is a type of *control flow statement*
- This allows us to iterate a particular operation using an index

```cpp
double calculateAverage(const std::vector<double>& values) {
    double sum = 0.0;
    for (size_t i = 0; i < values.size(); i++) {
        sum += values[i];
    }
    return sum / values.size();
}
```

Advanced Programming

Review of C++

- We define an index `i` (for the purposes of this example, read `size_t` as `int`)
- The index begins at `0`, and the second argument provides the total number of points - here, the loop will iterate from `0` to `values.size() - 1`
- `i++` indicates at the end of each time the statement in `{ }` is executed, `i` is increased by 1, i.e. `i = i+1`
- The statement in `{ }` will be executed until the max value of `i` is reached, then the loop will be exited

```
for (size_t i = 0; i < values.size(); i++) {
    sum += values[i];
}
```

Advanced Programming

Review of C++

- A more concise method is to use a range-based for loop
- This functionality has been available since C++11
- Here, `double value : values` indicates that the `for` loop should be executed for every `value` within the `values` vector

```
for (double value : values) {
    sum += value;
}
```

Advanced Programming

Review of C++

- We can now determine exactly what this function does
  - It is passed the reference to a vector of values
  - It defines a `double sum = 0.0`
  - It iterates through vector, adding the values together
  - The average value is returned

```
double calculateAverage(const std::vector<double>& values) {
    double sum = 0.0;
    for (size_t i = 0; i < values.size(); i++) {
        sum += values[i];
    }
    return sum / values.size();
}
```

Advanced Programming

Review of C++

```cpp
#include <iostream>
#include <fstream>
#include <vector>
#include <sstream>
#include <string>

double calculateAverage(const std::vector<double>& values) {
    double sum = 0.0;
    for (size_t i = 0; i < values.size(); i++) {
        sum += values[i];
    }
    return sum / values.size();
}

int main() {
    std::ifstream file("data.csv"); // Replace "data.csv" with the path to

    if (!file) {
        std::cerr << "Failed to open the file." << std::endl;
        return 1;
    }
```

Advanced Programming

Review of C++

- The file is read in the `main()` function
- First, `std::ifstream file("data.csv")` opens file (input file stream)
- Second part is an error message - these are very important for smooth debugging

```
if (!file) {
    std::cerr << "Failed to open the file." << std::endl;
    return 1;
}
```

Advanced Programming

Review of C++

- `if` is a conditional statement
  - Checks whether a given statement is true
  - If yes, executes the commands in `{ }`
- Often combined with an `else` statement with what to do if the statement is *not* true
- `if (!file)` checks if the file was *not* found, `{ }` outputs an error message

```
    std::vector<double> columnData;
    std::string line;

    while (std::getline(file, line)) {
        std::istringstream ss(line);
        std::string cell;

        // Split the line into cells using comma as the delimiter
        std::getline(ss, cell, ','); // Skip the first column
        std::getline(ss, cell, ','); // Read the second column

        // Convert the cell value to double and store it in the vector
        double value;
        std::istringstream(cell) >> value;
        columnData.push_back(value);
    }

    file.close();

    if (columnData.empty()) {
        std::cerr << "No data found in the second column." << std::endl;
        return 1;
    }

    double average = calculateAverage(columnData);
    std::cout << "Average value of the second column: " << average << std::e

    return 0;
}
```

Advanced Programming

Review of C++

- We have another type of control flow statement, a `while` loop
- These loop through a block of code contained in `{ }` as long as the condition specified is `true`

```
while (std::getline(file, line)) {
```

- Here, we use the `std::getline` function (https://cplusplus.com/reference/string/string/getline/) to extract characters from `file` and store them as a string in `line`
- `while` loop is executed while there is still a row of data in the file

Advanced Programming

Review of C++

- `std::istringstream` reads a string to a stream
- `std::getline` is also used with a comma delimiter to read the value of the second column in each row
- Values are appended onto the vector `columnData` using `push_back`
- File is closed with `file.close()`

Practical Task: Compile and run the `data_reader.cpp` example

Advanced Programming

Review of C++

- We have now been introduced to some key C++ concepts
  - Defining additional functions
  - Passing by value/reference
  - Control statements: `if, else, for, while`
  - Opening and reading data files
  - Manipulating strings and streams
  - Using vectors
  - Writing error messages

Advanced Programming

Review of C++

- What would we do to `data_reader.cpp` if we wanted to calculate the same average, but for a different data file?
- Using what we currently know, we would either have to:
  - Write all the same code out again with a different file name
  - Recompile the code with a different file name and run again
  - Change the name of our data file to fit with the name in the code
- Might work for a few files, but not straightforward with many files
- A more efficient option is to use a more advanced, central C++ concept: a *class* (a user-defined data type)
- We can define a class that takes a file name as a *parameter*

Advanced Programming

Review of C++

- Looking at the new `main()` function in `data_reader_class.cpp`, we see an unfamiliar data type - `CSVReader`

- This is a user-defined *class*

```
int main() {
    std::string filename = "data.csv"; // Replace "data.csv" with the path t

    CSVReader reader(filename);
    if (!reader.readData()) {
        return 1;
    }

    double average = reader.calculateAverage();
    std::cout << "Average value of the second column: " << average << std::e

    return 0;
}
```

```cpp
class CSVReader {
private:
    std::string filename;
    std::vector<double> columnData;

public:
    CSVReader(const std::string& filename) : filename(filename) {}

    bool readData() {
        std::ifstream file(filename);
        if (!file) {
            std::cerr << "Failed to open the file." << std::endl;
            return false;
        }

        std::string line;
        while (std::getline(file, line)) {
            std::istringstream ss(line);
            std::string cell;

            // Skip the first column
            if (std::getline(ss, cell, ',') && std::getline(ss, cell, ',')) {
                double value;
                std::istringstream(cell) >> value;
                columnData.push_back(value);
            }
        }

        file.close();

        if (columnData.empty()) {
            std::cerr << "No data found in the second column." << std::endl;
            return false;
        }

        return true;
```

Advanced Programming

Review of C++

- First, focus on the section underneath keyword `public`
- This defines functions that we can access from outside the class - here, we have `readData()` and `calculateAverage()`
- These are called in `main()` by:

```
CSVReader reader(filename);
if (!reader.readData()) {
    return 1;
}

double average = reader.calculateAverage();
```

- Any function defined within a class is called a *member function*

Advanced Programming

Review of C++

- At the top of the section `public`, we also have

```
public:
    CSVReader(const std::string& filename) : filename(filename) {}
```

- This is called the *constructor*
- The name of the constructor, `CSVReader`, must match the name of the class itself
- `(const std::string& filename)` specifies the parameters that the constructor takes

Advanced Programming

Review of C++

- At the top of the section `public`, we also have

```
public:
    CSVReader(const std::string& filename) : filename(filename) {}
```

- This is called the *constructor*
- The name of the constructor, `CSVReader`, must match the name of the class itself
- `(const std::string& filename)` specifies the parameters that the constructor takes
- The member variables must be initialised - this is done here by `: filename(filename)`
- Eg. Here, if we did not initialise, `reader.readData()` would return an error

Advanced Programming

Review of C++

- In the section underneath keyword `private`, we see the declaration of two variables

```
class CSVReader {
private:
    std::string filename;
    std::vector<double> columnData;
```

- These are used by the functions in `public`, but *cannot* be accessed outside the class - eg. `reader.columnData` is not valid

- Any variable defined within a class is called a *member variable*

Advanced Programming

Review of C++

- Another useful property of classes is *inheritance*
- This allows a class to *inherit* (access) public members from a base class

```
class CSVReader {
public:
    // Public members and functions of the CSVReader
};

class HigherLevelReader : public CSVReader {
    // Additional members and functions specific to t
};

int main() {
    HigherLevelReader obj;

    // Accessing public members from the base class
    obj.somePublicMemberOfCSVReader;
    obj.somePublicFunctionOfCSVReader();

    return 0;
}
```

Advanced Programming

Review of C++

- One specific kind of class is a `friend class`
- These can access *private and protected* members of other classes in which it is declared as a friend
- This is not possible via inheritance

Advanced Programming

Review of C++

```cpp
class CSVReader {
private:
    int privateData;

public:
    CSVReader(int data) : privateData(data) {}

    friend class CSVReaderFriend;
};

class CSVReaderFriend {
public:
    void accessPrivateData(const CSVReader& obj) {
        int data = obj.privateData;  // Friend class
        // Perform operations using private data
    }
};

int main() {
    CSVReader obj(42);
    CSVReaderFriend friendObj;

    friendObj.accessPrivateData(obj);  // Friend clas

    return 0;
}
```

Advanced Programming

Review of C++

- Note: there is another user-defined data type - `struct`
- All members are `public` by default, vs `private` by default with classes
- They are defined similarly to classes, eg.

```
#include <iostream>
using namespace std;

// Define a struct for representing a student
struct Student {
    int id;
    string name;
    int age;
};
```

- In practise, classes are used much more often

Advanced Programming

Review of C++

- Another key concept in C++ is *templating*
- This allows us to define a function *without having to specify the type of the variables it acts on*
- Some example syntax:

```
// Templated function that swaps two values of any type
template <typename T>
void swapValues(T& a, T& b) {
    T temp = a;
    a = b;
    b = temp;
}
```

- Note that this is the same syntax used for `vector` - these are class templates

Advanced Programming

Review of C++

Some miscellaneous advanced features:

- When defining the `main()` function, we will often see the definition

  ```
  int main(int argc, char *argv[]) {}
  ```

- This is the method used to pass *command line arguments* to `main()` - whatever you put on the command line will become an input
- `argc` - argument count (number of strings pointed to by `argv`)
- `argv` - argument vector (these names are convention, can be given other names)
- Run programs like this using eg. `./Hello_World param1 param2`
- `::` denotes the *scope* (as we have seen before, in i.e. the namespace)
- This also applies to *classes*, eg. `classname::functionname`

Advanced Programming

Review of C++

Now that we understand a lot of C++ key features, how do we navigate a large, complex research code?

- Will use the example of adaptive mesh refinement numerical relativity code, GRChombo (https://github.com/GRChombo/GRChombo)
- Other codes large codes include eg. CosmoLattice, SOFTSUSY, N-Body code GADGET
- In short:
  - Read any documentation
  - Make use of example code
  - Find `main()` and follow functions from there

Advanced Programming

Debugging

Advanced Programming

Debugging: Types of Bugs

- Debugging is the process of identifying and fixing errors in code
- Almost impossible to avoid, so must know how to identify and fix them
- Common types of error in C++ code are:
  - Compile-time errors
    - Syntax error (eg. missing `;` or `)`) or type error (eg. `int x = "hello";`)
    - These are usually flagged during compilation
    - The program will not compile until the error is fixed
    - Usually easy to identify

Advanced Programming

Debugging: Types of Bugs

- Linker error
  - Occurs during the ‘linking’ stage of compilation, program will not compile
  - Can occur if eg.
    - Included header file is not found
    - `main()` is incorrectly written as eg. `Main()`
    - A function or class is declared, but never defined, eg.

```
int func();

int main() { int x = func(); }
```

Advanced Programming

Debugging: Types of Bugs

- Runtime error
  - The code will compile and run, but will crash or give an unexpected result, such as:
    - Segmentation fault - the program tries to access a memory location that is not allowed, eg. accessing `array[10]` when we have only defined `array[5]`
    - `NaN` - this stands for ‘not a number’ and occurs most commonly when the code has attempted to divide a number by zero
  - Can be tricky to find, experience often helps

Advanced Programming

Debugging: Types of Bugs

- Logic or other semantic error (type of runtime error)
  - Again, the code will usually compile and run, but will give a result that we know is wrong, eg.:

```
int add(int x, int y){ return x-y; }
int main() {
      cout << "5+3=" << add(5,3) << '\n';
      return 0;
 }
```

  - These can be very difficult to identify, and can cause subtly incorrect results
  - Can mitigate against these by comparing with output from another code, or using own physics/maths knowledge

Advanced Programming

Debugging: Types of Bugs

Example: Compile `data_reader_class_buggy.cpp`

First, some setup (some of this will be used later) - this is to enable us to easily edit files within the container:

- Pull the latest commit from the course repository
- Open Docker on your machine (laptop)
- Install ‘Remote Development’ extension pack in VSCode

Advanced Programming

Debugging: Types of Bugs

Running the Docker container:

- Rebuild the Docker image in VSCode (View -> Command Palette -> Docker Images: Build Image)
- ‘Run interactive’ Docker container

Next, we want to be able to use VSCode tools from within our container:

- View -> Command Palette… -> Attach to running container
- Open in new window `/usr/src/dockertest1/data_reader_class_buggy.cpp`

Advanced Programming

Debugging: Types of Bugs

- From within the container, we can now compile the example
- We first see a *syntax error* at compile time - how do we trace it?

```
root@5b69181e1e32:/usr/src/dockertest1# g++ data_reader_class_buggy.cpp data_reader_class_buggy
data_reader_class_buggy.cpp: In function 'int main()':
data_reader_class_buggy.cpp:51:5: error: expected ',' or ';' before 'reader'
   51 |     reader.readData();
      |     ^~~~~~
```

Example: Fix the syntax error, recompile and run

- The code now complies, but running gives a runtime error

```
root@5b69181e1e32:/usr/src/dockertest1# ./data_reader_class_buggy
Average value of the second column: -nan
```

- How do we determine the cause of this bug?

Advanced Programming

Debugging: Methods

- There are two primary `code-based' methods of debugging:
  - Print debugging - coder adds print statements at strategic points to check whether the code runs to those points
  - Using a debugger program - coder steps through the source code to find sources of error
- Even better than debugging is to add in your own error messages in advance to anticipate potential problems
  - Note that we included these in the `data_reader_class.cpp` example - would this have helped us catch the runtime error above?

Advanced Programming

Debugging: Methods

- Another very useful method is ‘rubber duck debugging’
- Essentially, this means using your own brain rather than outsourcing the solution to the computer
- Go through the section of code that you suspect contains the bug, line by line, explaining what each line does
- ‘Rubber duck’ refers to any object that you can pretend you are explaining to
- Unless you are lucky, real humans tend not to want to listen to the minutiae of your code implementation 😇 (unless you are really stuck, which is where RSEs come in)

Advanced Programming

Debugging: Methods

- Finally, we can debug programs by checking that the output makes sense in and of itself, as well as in comparison to previous output
- This can be done by:
  - Visualising the output (we will come to this later, especially useful with 3D grid output)
  - Plotting global variables, e.g. some physical quantity, such as total energy density
  - Comparing the above with output before the latest change (can use the Linux command `diff`)
  - If running in parallel, compare parallel to serial output

Advanced Programming

Debugging: Print Debugging

Advantages of print debugging:

- Easy to add print statements into code
- No need to learn to use other tools
- Especially useful in serial codes - it is clear where the program has run to
- If you have an idea of what might be going wrong, you can print out specific variables to double check
- Can use without removing optimisation

Advanced Programming

Debugging: Print Debugging

Disadvantages of print debugging:

- In compiled codes, must recompile and run each time you want to move the print statements
- Can take time if you have to queue your simulation on a large machine
- Doesn’t always catch errors
- Need to go back through at the end and remove the statements that you have added
  - This can be avoided by using a specific debug flag in your Makefile

Advanced Programming

Debugging: Print Debugging

Example: Compile `data_reader_class_buggy.cpp` *with additional print statement(s) to help locate the bug*

- We notice that `calculateAverage()` is the function that returns the error
- Might want to print out the variables that it uses in its calculation
- See that `columnData.size()` returns zero

If we are still stuck, we can use a debugger program…

Advanced Programming

Debugging: Debuggers

Advantages of debuggers:

- Can obtain detailed information about variables and memory locations while it executes
- Can step through code line by line
- Can stop execution at any point using a *breakpoint*
- Some integrated with code editors for easy use
- Some debuggers designed for parallel code

Advanced Programming

Debugging: Debuggers

Disadvantages of debuggers:

- Have to learn to use (especially if using from command line)
- Setup and use can be complex
- Can lead to focussing too much on minute details, rather than the bigger picture
- Doesn’t always catch errors
- Can be slow - usually need to compile code without optimisation

Advanced Programming

Debugging: Debuggers

Some examples of debuggers:

- GDB (`GNU Project Debugger' - https://www.sourceware.org/gdb/)
  - Main debugger used by VSCode (C/C++ Extension Pack)
  - Debugs serial applications
- Valgrind (https://valgrind.org/)
  - For debugging and profiling Linux programs
  - Can use for parallel programs
- Linaro DDT (formerly Allinea DDT) (DDT - distributed debugging tool)
  - Graphical debugger for parallel programs

Advanced Programming

Debugging: Debuggers

Example - Debugging in Docker Container in VSCode:

- Click ‘Run and Debug’ in tab on left
- Choose preferred compiler from drop down menu and ‘debug active file’
- This allows us to use GDB to examine the code
- At first, the code runs and no obvious bug is identified - we must add *breakpoints* to examine variables which we suspect
- In VSCode, click on the left of a line number where you would like the execution to stop

Advanced Programming

Debugging: Debuggers

Example - Debugging in Docker Container in VSCode:

- Click ‘Run and Debug’ again - the code will stop at the breakpoint, we can inspect the variables in the panel on the left
- Can use the icons at the top to step over/into/out
- If you add a breakpoint, two new windows will appear on the left:
  - Call Stack - shows which functions have been called to reach the line with the breakpoint
  - Watch - allows you to type in variable names that you want to watch throughout the execution

Advanced Programming

Debugging: GDB

- We can also run GDB from the command line (installed via Dockerfile):
  - `cd /usr/src/dockertest1/`
  - `gdb data_class_reader_buggy`
- We then add breakpoints/watchpoints and control the program flow using command line
  - The sample session (https://sourceware.org/gdb/current/onlinedocs/gdb.html/Sample-Session.html#Sample-Session) is a useful starting point

Advanced Programming

Debugging: GDB

- Add breakpoint with eg. `break data_reader_class_buggy.cpp:47`
- Run with `run` (or `r`)
- Use `n` for next, `s` to step into a subroutine, `c` to continue, `p` to print a value
- `bt` gives a backtrace to see where we are in the stack

```
(gdb) break data_reader_class_buggy.cpp:47
Breakpoint 1 at 0x26b7: file /usr/src/dockertest1/data_reader_class_buggy.cpp, line 48.
(gdb) run
Starting program: /usr/src/dockertest1/data_reader_class_buggy
warning: Error disabling address space randomization: Operation not permitted
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at /usr/src/dockertest1/data_reader_class_buggy.cpp:48
48          CSVReader reader(filename);
(gdb)
```

Advanced Programming

Debugging: GDB

Other useful features are:

- Calling functions from command line eg. `call calculateAverage()`
- Adding watch points eg. watch `average`
  - Program must be running and stopped at a breakpoint
- Inspecting the code without running eg. look at specific memory address, `l *0x8000000000000`
- Inspect core dumps eg. to find out which line caused a seg fault
- Define your own commands via `~/.gdbinit`

Advanced Programming

Debugging: Valgrind

- Valgrind is open source `instrumentation framework' for building dynamic analysis tools
- This includes profiling tools (see later) and debugging
- For debugging, Valgrind is most useful for detecting memory-related errors and threading errors for parallel applications
- Most popular (and default) Valgrind tool is `memcheck`
- To use Valgrind, first compile your program with `-g` flag to include debugging information
  - This will ensure that error messages include exact line numbers
- Do not compile with any optimisation flags higher than `-O0`

Advanced Programming

Debugging: Valgrind

- *Example: run memory check using*

  ```
  valgrind --leak-check=yes ./data_reader_class_buggy
  ```

  (this enables the detailed memory leak detector)

- To change from default tool `memcheck`, add `--tool=`
- Program is run on a `synthetic' CPU provided by Valgrind core
- `Memcheck` adds code to check every memory access and computed value
- Note: program will run 20-30x slower and use a lot more memory

Advanced Programming

Debugging: Valgrind

- Example output for `data_reader_class_buggy`:

```
root@526151d9bd1f:/usr/src/dockertest1# valgrind --leak-check=yes ./data_reader_class_buggy
==4655== Memcheck, a memory error detector
==4655== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==4655== Using Valgrind-3.18.1 and LibVEX; rerun with -h for copyright info
==4655== Command: ./data_reader_class_buggy
==4655==
Average value of the second column: -nan
==4655==
==4655== HEAP SUMMARY:
==4655==     in use at exit: 0 bytes in 0 blocks
==4655==   total heap usage: 3 allocs, 3 frees, 74,200 bytes allocated
==4655==
==4655== All heap blocks were freed -- no leaks are possible
==4655==
==4655== For lists of detected and suppressed errors, rerun with: -s
==4655== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
root@526151d9bd1f:/usr/src/dockertest1#
```

Advanced Programming

Debugging: Valgrind

- For more detailed debugging, Valgrind can interface with gdb

Example: run the command

```
valgrind --vgdb-error=0 /.data_reader_class_buggy
```

then follow instructions provided in the terminal

- Start gdb in another shell
- Tunnel to vgdb (output from first command is input to second command)
- Can debug with gdb commands and combine with eg. the command `monitor` to use Valgrind tools

Advanced Programming

Profiling

Advanced Programming

Profiling

- Profiling is the measuring of the performance of a program
- Can identify where the program spends its time/memory
- Usually used to identify bottlenecks to enable optimisation
- It can also help to identify bugs, if certain functions are being called more or less than expected
- Profiling can be carried out by:
  - Putting timers manually into the program
  - Using software eg. gprof, Valgrind, VTune
    - Note: there is no straightforward profiling tool built into VSCode, as there is for debugging (other than for JavaScript)

Advanced Programming

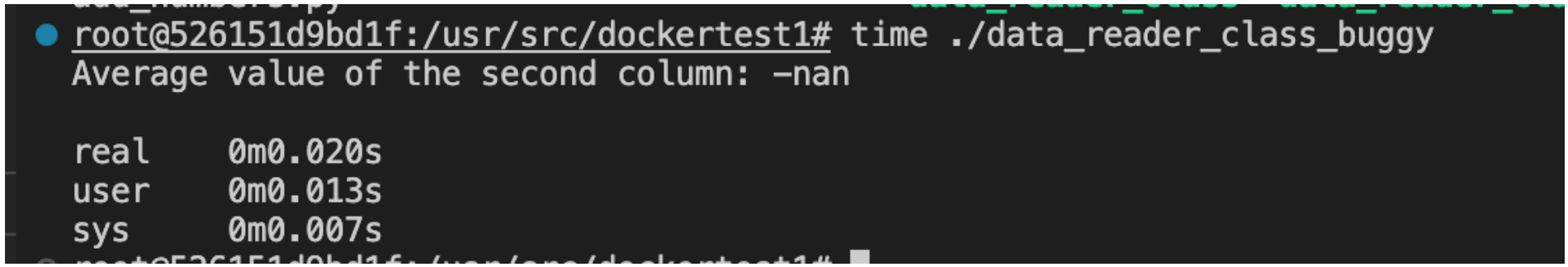
Profiling

- Performance is primarily measured in two ways:
  - Wall time - total time elapsed while a certain section of the code is running
  - Processor time
    - You will often see the phrase `CPU time' used as a measurement
    - Nowadays, GPUs are becoming increasingly popular, so `GPU time' will likely also be widely used
    - The concept behind both is the same - the total time taken by the processor to run part of the code
    - This excludes communication time, I/O and other costs

Advanced Programming

Profiling

- Simple profiling can be done using the Unix `time` command
  - `real` - total time for program to run from start to finish i.e. wall time
  - `user` - CPU time spent in user mode
  - `sys` - CPU time spent in kernel mode (access hardware components etc.)
  - *Example: Try this on one of the examples from the class repo*



Advanced Programming

Profiling

- For more detailed profiling, you will need to use additional tools:
  - Timers - most coding languages will have a timer module that can be used to write timers into the code
  - Profilers - these are programs that can tell the user where the time running code is being spent, eg. functions, wait time for parallel programs
- (There is a parallel here with print debugging vs software debuggers)
- In both cases, timing data can fluctuate - usually a good idea to take an average/median

Advanced Programming

Profiling: Instrumenting with Timers

Advantages:

- Straightforward to implement
- Straightforward to run and interpret
- You can specify exactly which parts you want to time

Disadvantages:

- Takes time to implement in a large code
- Can easily miss sections if implemented sporadically
- Can provide incomplete information eg. Wall time vs CPU time

Advanced Programming

Profiling: Instrumenting with Timers

- Most up to date timer for C++ is `<chrono>` - measures wall time
- Another commonly used library in C++ and C is `time.h` (`<ctime>`)
  - Can measure CPU (`<time.h>`) or wall time (`<sys/time.h>`)
  - Use of this library is sometimes discouraged (out of date, not thread safe)
- Equivalent timers exist in other languages eg. Python `time` and Fortran `CPU_TIME`

Advanced Programming

Profiling: Instrumenting with Timers

- `<chrono>` - main functions used are `high_resolution_clock` and `duration_cast` (converts time into desired measurement, eg. milliseconds)

Example 1: Compile `chrono_timer.cpp` *and run a few times*

- `time.h` - main function is `clock()` and macro `CLOCKS_PER_SEC`

Example 2: Compile `timeh_time.cpp` *and run a few times*

Advanced Programming

Profiling: Profilers

Advantages:

- Can quickly identify bottlenecks, especially useful for parallel code
- Some have user-friendly graphical interfaces

Disadvantages:

- Need to learn how to use
- Sometimes unavailable on large machines, need to arrange installation
- Can add overhead which skews performance data

Advanced Programming

Profiling: gprof

- `gprof` (GNU profiler) is an open source profiler for Unix applications
- It provides the user with (`https://ftp.gnu.org/old-gnu/Manuals/gprof-2.9.1/html_mono/gprof.html`)
  - *Flat profile* - how much time the program spends in each function, and how many times that function is called
  - *Call graph* - which functions called each function, and which other functions they called. This also estimates how much time is spent in the subroutines of each function
  - *Annotated source listing* - copy of program's source code, labelled with number of times each line is executed, `-A` flag

Advanced Programming

Profiling: gprof

- Before profiling with `gprof`, the program must be compiled with the flag `-pg`
  - The `-g` flag can also be useful for line-by-line profiling and basic-block counting
- The program must be run before profiling to generate the information for `gprof`
  - It will run slower due to the time taken to collect and write the profile
- Running a program compiled with the `-pg` flag will generate a file called `gmon.out` - this is the profiling information
- We can then run `gprof` with e.g. `gprof chrono_timer > output`

Example: Run on eg. chrono_timer.cpp and examine output

Advanced Programming

Profiling: gprof

```
^L
                     Call graph (explanation follows)


granularity: each sample hit covers 4 byte(s) for 0.33% of 3.02 seconds

index % time    self  children    called     name
                                               <spontaneous>
[1]    100.0    0.00    3.02                 main [1]
                1.76    0.00       1/1           someFunction() [2]
                0.88    0.00       1/1           anotherFunction() [3]
                0.38    0.00       1/1           yetAnotherFunction() [4]
                0.00    0.00       3/3           std::common_type<std::chrono::duration<long, std::ratio<1l, 1000000000l> >, std::chrono::dur\
ation<long, std::ratio<1l, 1000000000l> > >::type std::chrono::operator-<std::chrono::_V2::system_clock, std::chrono::duration<long, std::rat\
io<1l, 1000000000l> >, std::chrono::duration<long, std::ratio<1l, 1000000000l> > >(std::chrono::time_point<std::chrono::_V2::system_clock, st\
d::chrono::duration<long, std::ratio<1l, 1000000000l> > > const&, std::chrono::time_point<std::chrono::_V2::system_clock, std::chrono::durati\
on<long, std::ratio<1l, 1000000000l> > > const&) [18]
                0.00    0.00       3/3           std::enable_if<std::chrono::__is_duration<std::chrono::duration<long, std::ratio<1l, 1000l> \
> >::value, std::chrono::duration<long, std::ratio<1l, 1000l> > >::type std::chrono::duration_cast<std::chrono::duration<long, std::ratio<1l,\
 1000l> >, long, std::ratio<1l, 1000000000l> >(std::chrono::duration<long, std::ratio<1l, 1000000000l> > const&) [14]
                0.00    0.00       3/3           std::chrono::duration<long, std::ratio<1l, 1000l> >::count() const [13]
-----------------------------------------------
                1.76    0.00       1/1           main [1]
[2]     58.3    1.76    0.00       1         someFunction() [2]
-----------------------------------------------
                0.88    0.00       1/1           main [1]
[3]     29.1    0.88    0.00       1         anotherFunction() [3]
-----------------------------------------------
                0.38    0.00       1/1           main [1]
[4]     12.6    0.38    0.00       1         yetAnotherFunction() [4]
-----------------------------------------------
                0.00    0.00       3/9           std::chrono::duration<long, std::ratio<1l, 1000l> > std::chrono::__duration_cast_impl<std::c\
hrono::duration<long, std::ratio<1l, 1000l> >, std::ratio<1l, 1000000l>, long, true, false>::__cast<long, std::ratio<1l, 1000000000l> >(std::\
chrono::duration<long, std::ratio<1l, 1000000000l> > const&) [15]
                0.00    0.00       6/9           std::common_type<std::chrono::duration<long, std::ratio<1l, 1000000000l> >, std::chrono::dur\
ation<long, std::ratio<1l, 1000000000l> > >::type std::chrono::operator-<long, std::ratio<1l, 1000000000l>, long, std::ratio<1l, 1000000000l>\
 >(std::chrono::duration<long, std::ratio<1l, 1000000000l> > const&, std::chrono::duration<long, std::ratio<1l, 1000000000l> > const&) [19]
[11]     0.0    0.00    0.00       9         std::chrono::duration<long, std::ratio<1l, 1000000000l> >::count() const [11]
-----------------------------------------------
                0.00    0.00       6/6           std::common_type<std::chrono::duration<long, std::ratio<1l, 1000000000l> >, std::chrono::dur\
ation<long, std::ratio<1l, 1000000000l> > >::type std::chrono::operator-<std::chrono::_V2::system_clock, std::chrono::duration<long, std::rat\
```

Advanced Programming

Profiling: Valgrind for Profiling

- We can also use Valgrind for profiling
- Run command such as eg.

  ```
  valgrind --tool=callgrind ./chrono_timer
  ```

  (Remember we need to have compiled using `-g` to use valgrind)
- Note that with no optimisation, this is significantly slower than gprof
- The output depends significantly on the level of optimisation - we use `-O0` here
- Outputs `callgrind.out.xxxxx` which can be interpreted using `callgrind_annotate callgrind.out.xxxxx`

Example: try out the above commands

Advanced Programming

Profiling: Valgrind for Profiling

```
--------------------------------------------------------------------------------
-- Auto-annotated source: chrono_timer.cpp
--------------------------------------------------------------------------------
Ir

            .           #include <iostream>
            .           #include <chrono>
            .
            .           // Function to be timed
            .           void someFunction()
            3 ( 0.00%)  {
            .               // Simulating some work
3,000,000,004 (58.80%)      for (int i = 0; i < 1000000000; ++i)
            .               {
            .                   // Do some computation
            .               }
            4 ( 0.00%)  }
            .
            .           // Another function to be timed
            .           void anotherFunction()
            3 ( 0.00%)  {
            .               // Simulating some work
1,500,000,004 (29.40%)      for (int i = 0; i < 500000000; ++i)
            .               {
            .                   // Do some computation
            .               }
            4 ( 0.00%)  }
            .
            .           // Yet another function to be timed
            .           void yetAnotherFunction()
            3 ( 0.00%)  {
            .               // Simulating some work
  600,000,004 (11.76%)      for (int i = 0; i < 200000000; ++i)
            .               {
            .                   // Do some computation
            .               }
            4 ( 0.00%)  }
            .
            .           int main()
```

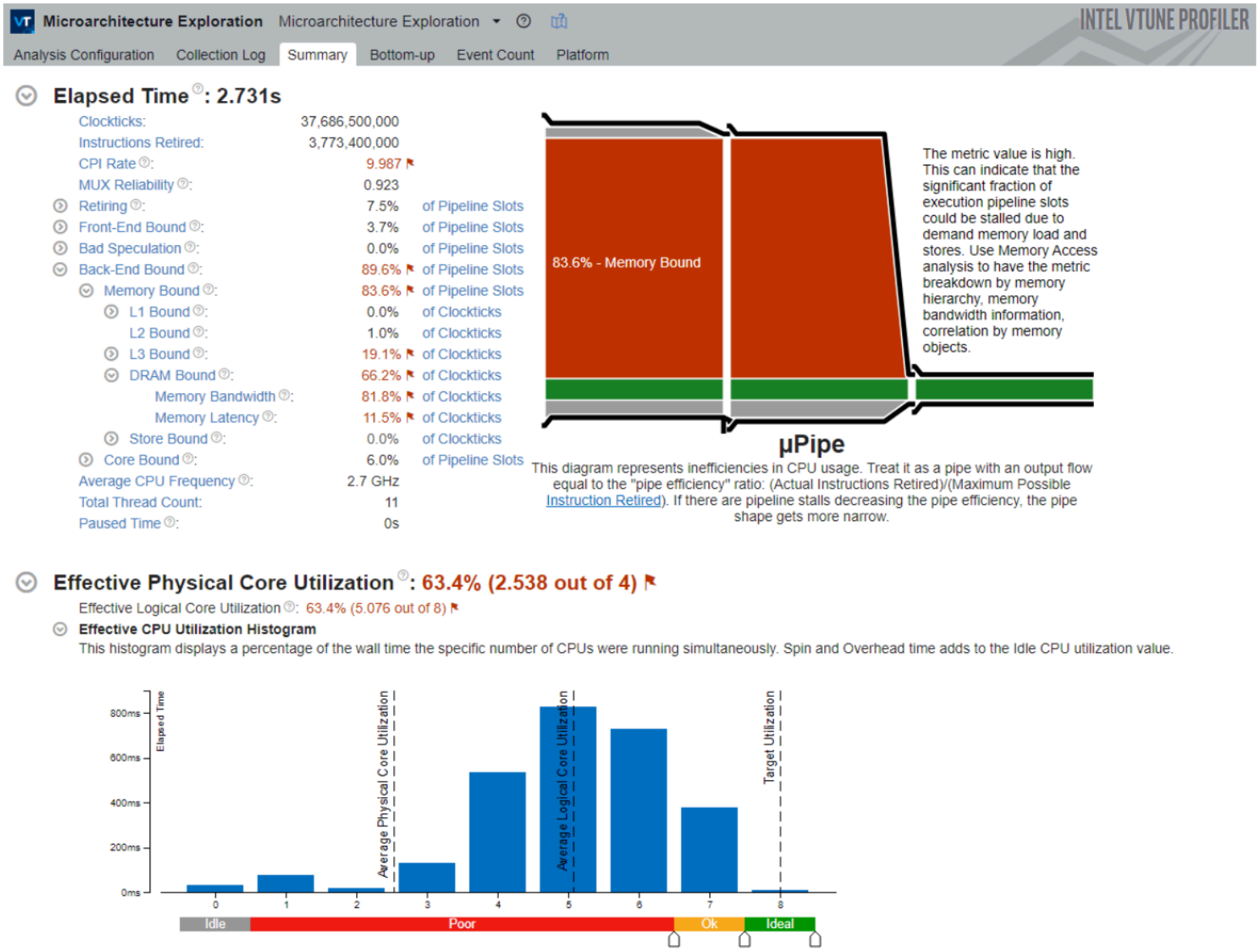
Advanced Programming

Profiling: Other Advanced Tools

- Advanced profiling tools usually require licenses, eg. Intel VTune
- Provides much more detailed information than open source options
- Particularly useful for parallel code eg. effectiveness of threading and vectorisation, scalability
- Often will have a GUI - these can be tricky to set up remotely, but your system admin should be able to help

Advanced Programming

Profiling: Other Advanced Tools



Parallelism

Supercomputer Architectures and Overview of Parallelisation

Parallelism

Supercomputer Architectures

- CPUs (central processing units) have been the workhorse of computing since the 1960s, taking over from mechanical punch cards

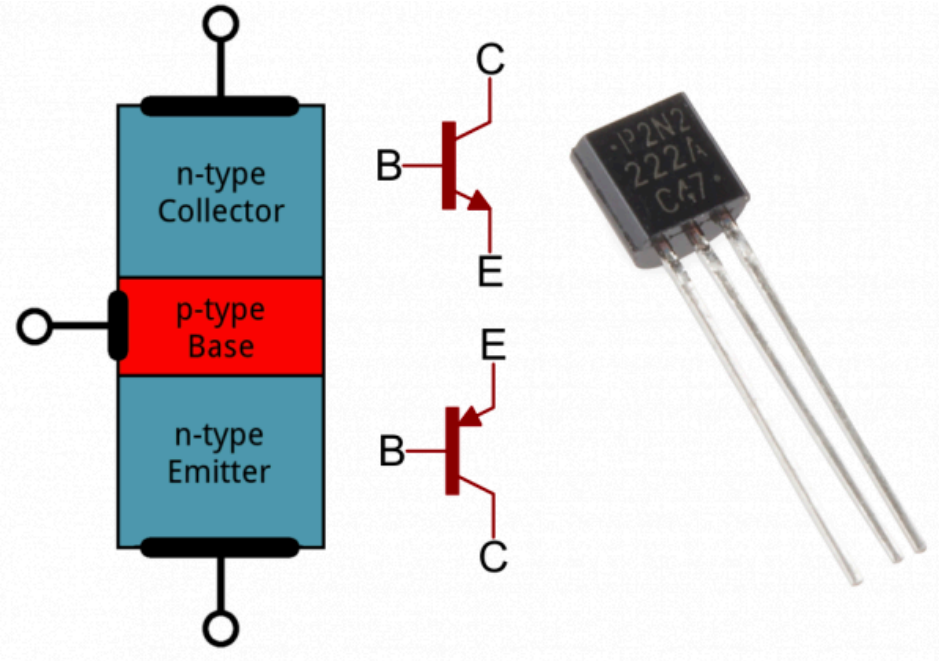
- From the 1960s, it was possible to speed up a CPU by adding more *transistors* to a chip, due to a reduction in the transistor size over time

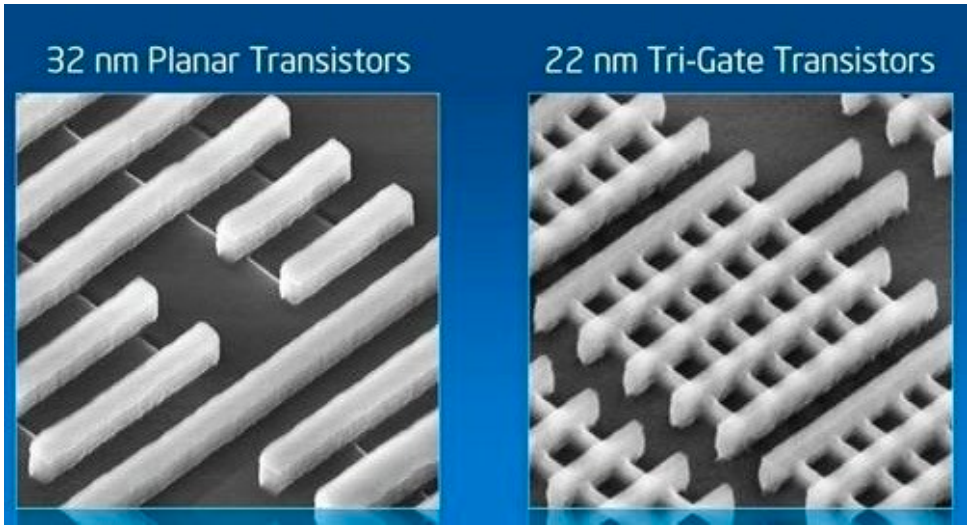
  - A transistor is a semiconductor device that is used heavily in electronics

  - They can amplify or switch electric signals via *logic gates* - used to switch between binary 0s and 1s

Parallelism

Supercomputer Architectures





Parallelism

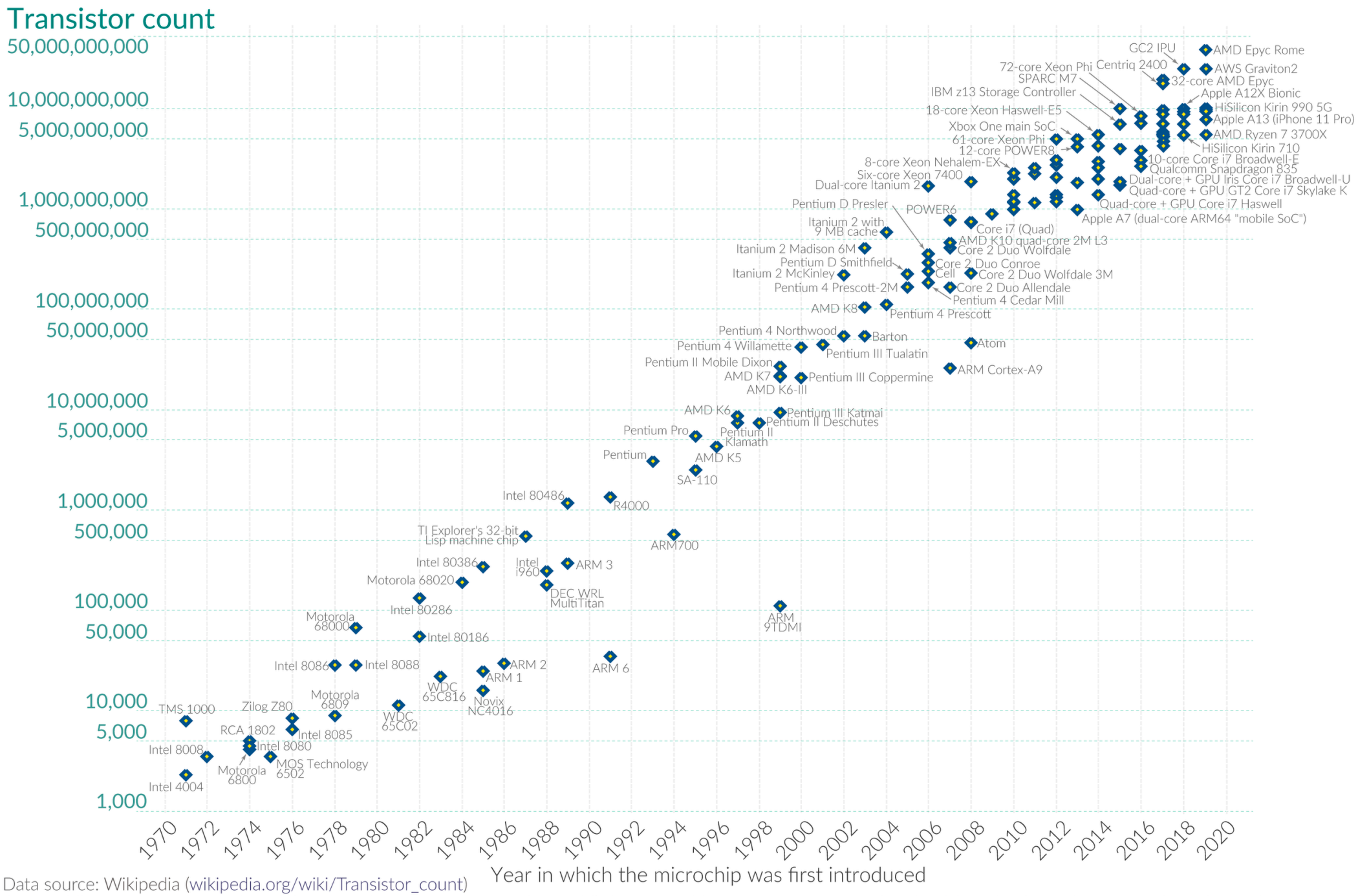
Supercomputer Architectures

- This is encapsulated in *Moore's Law* - 'the number of transistors in an integrated circuit doubles about every two years' i.e. exponential growth
  - The key reason for this growth in number was due to a *reduction in size* of the transistor
  - Currently, the most modern chips can fit tens of billions of transistors
- In general, this doubling would come at *no increase in cost*

Moore's Law: The number of transistors on microchips doubles every two years

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years. This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.

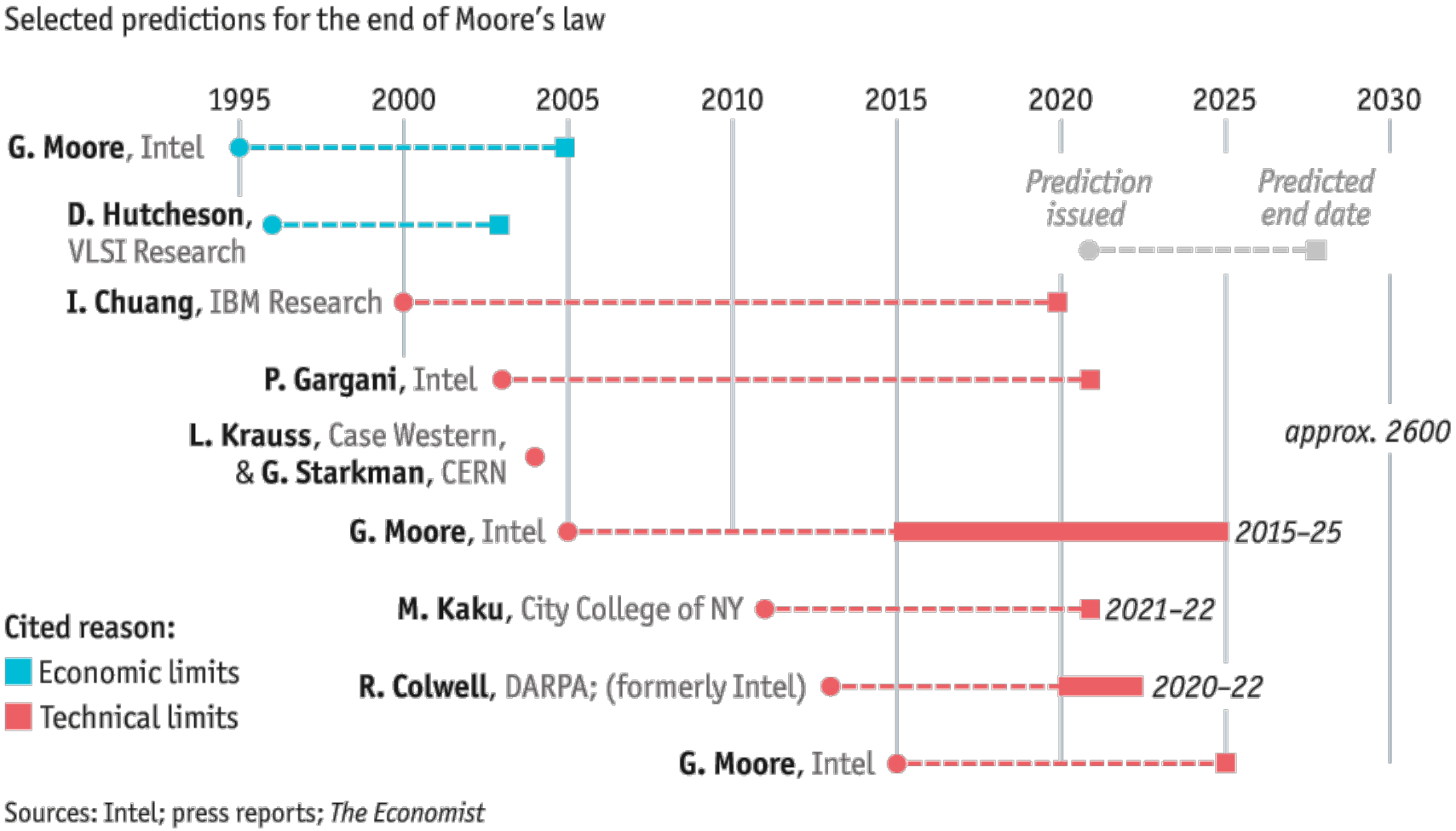


Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)

Parallelism

Supercomputer Architectures

- There have been many predictions over when Moore's law will end…



Parallelism

Supercomputer Architectures

- Moore's Law can be combined with *Dennard's (MOSFET) scaling:*
  - *F*or a transistor, $P = CV^2f$,
    - C - capacitance, scales with linear size of transistor
    - V - voltage, scales with linear size (electric field constant)
    - Current and transition time also scaled down with linear size
    - f - frequency scales with 1/delay
    - P - power consumption decreases, scaling with area
  - As the transistors get smaller, the *power density* stays constant - if power density doubles, power consumption (twice the transistor number) stays the same

Parallelism

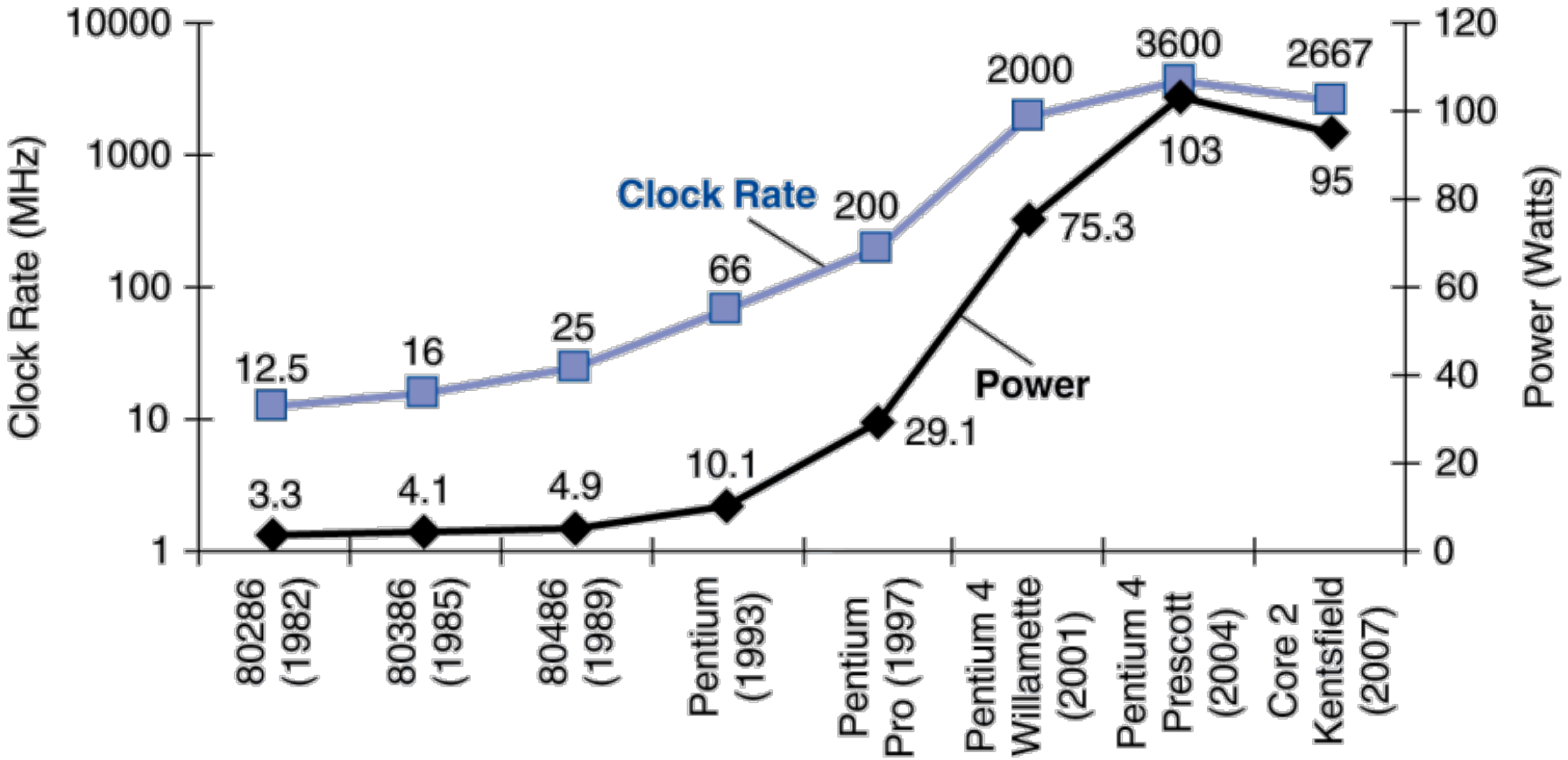
Supercomputer Architectures

- Together, this meant that performance per joule of energy grew even faster than the transistor number, doubling every ~18 months
- Smaller transistors enables a higher frequency of operations for a fixed power
- Over this time, *clock speed* of CPUs increased dramatically
  - Clock speed is the number cycles per unit time
- This meant that, in practice, you could speed up your program simply by using more up-to-date CPUs
- This is no longer the case…

Parallelism

Supercomputer Architectures

- Although Moore's law seems to be holding for now, for the last 20 years or so, the performance of CPUs has stalled



Parallelism

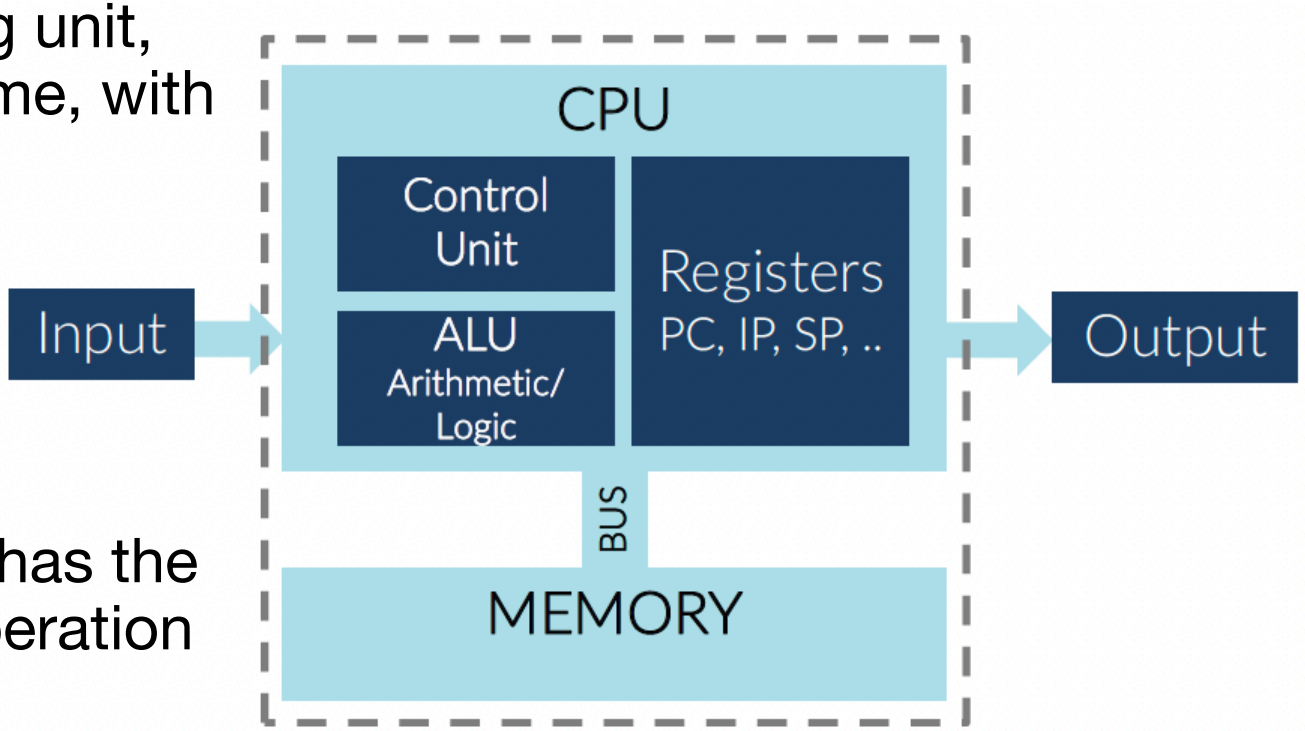
Supercomputer Architectures

- This is because physical limits have come into play - Dennard's scaling is broken
- Since the mid 2000s, as transistors have become smaller, the power density has *increased*, limiting the clock rate
- This increase is caused by *quantum effects*, such as:
  - Current leakage - also causes the chip to heat up
  - Threshold voltage
  - Physical limits and atomic scales
- One intrinsic limit on CPUs is the clock rate
- However, CPUs contain other elements that can also limit performance…

Parallelism

Supercomputer Architectures

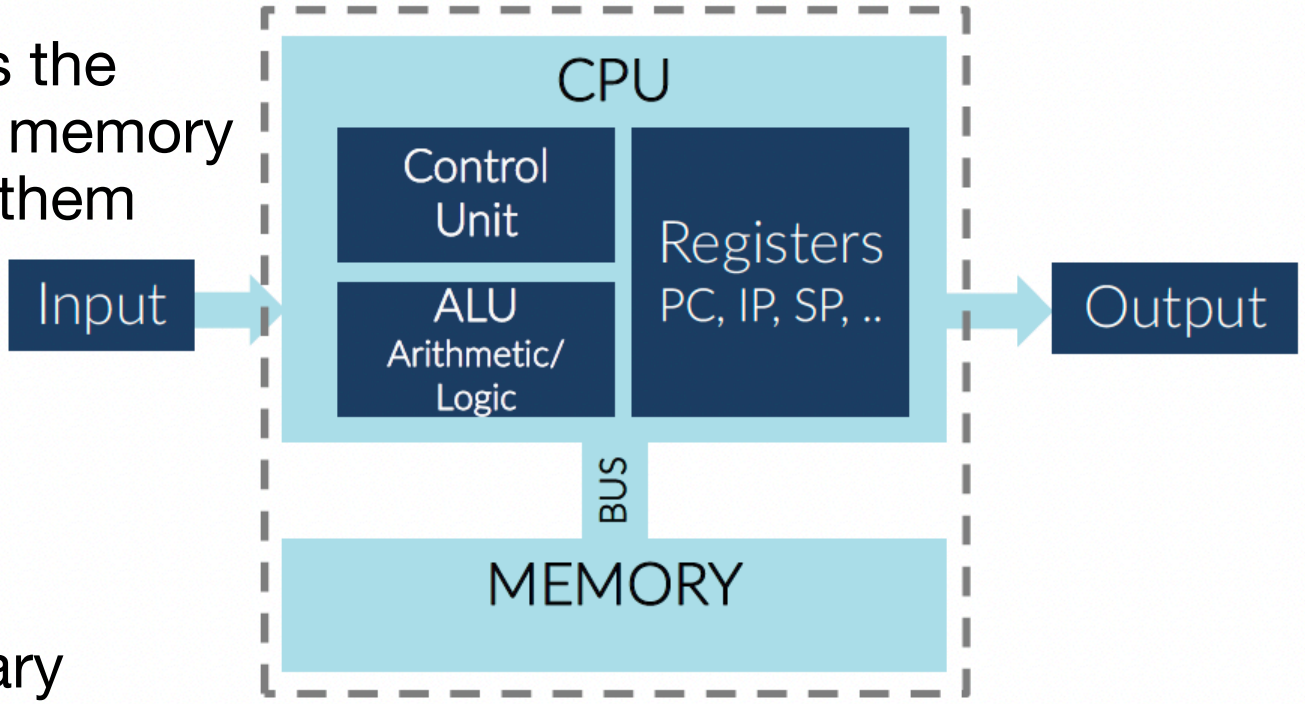
- To understand how a CPU works, it is useful to understand the *architecture*
- We first look at a *single core* (von Neumann architecture)
- This is one processing unit, one instruction at a time, with `flat' memory, i.e.
  - Access to any memory location has the same cost
  - Access to memory has the same cost as an operation execution



Parallelism

Supercomputer Architectures

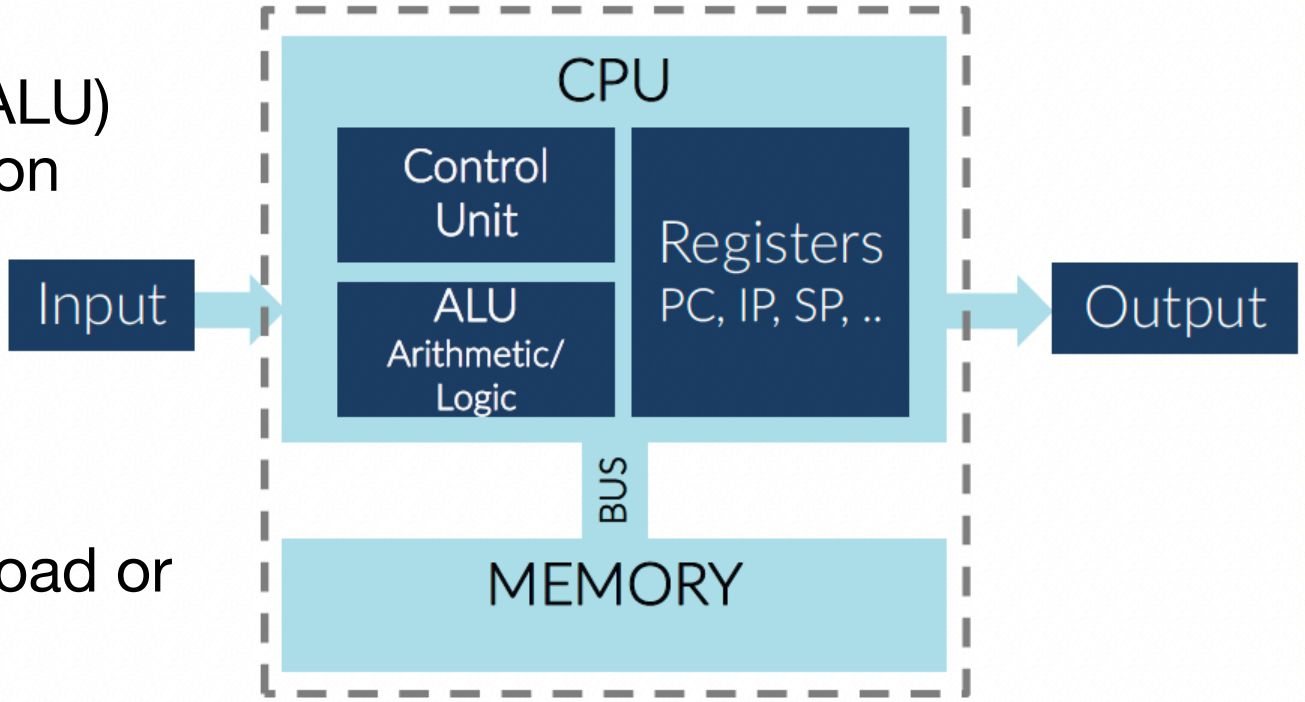
- Registers - high speed storage areas in the CPU where all data must be stored before it can be processed
- Arithmetic and Logic Unit - enables arithmetic and logic operations
- Control Unit - controls the operation of the ALU, memory and I/O devices, tells them how to respond to program instructions from memory unit
- Bus - transmits data
- Memory - RAM, primary



Parallelism

Supercomputer Architectures

- First, control unit fetches the next instruction from memory to the instruction register (IR) (instructions are stored as data, with memory addresses)
- The control unit decodes the instruction on the IR
- The processing unit (ALU) executes the instruction
- The control unit stores the result to memory
- Input and output eg. load or display results



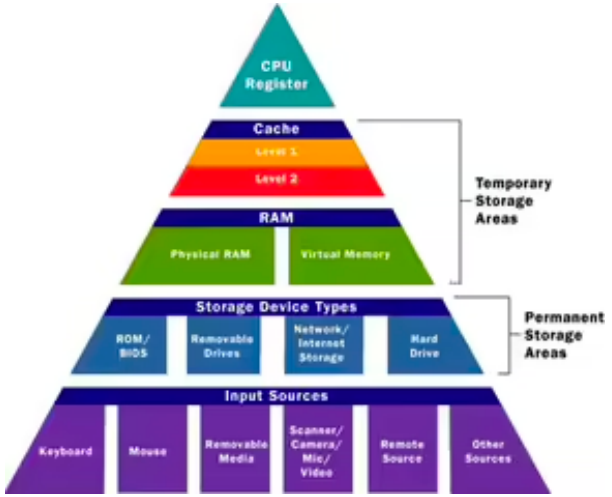
Parallelism

Supercomputer Architectures

- In the 1990s, the CPU speed began to overtake the speed of the memory transfer from RAM
  - The time between initiating a request for something stored in memory and it being received by a processor is called the *memory latency* - higher latency = longer wait
- This is the so-called *memory wall,* creating another limit on the overall performance
- To get around this, it was necessary to speed up the memory
- This was achieved via a mechanism called the *cache*
  - This term comes from wineries - we will see why!

Parallelism

Supercomputer Architectures



- For memory to be faster, it has to be *closer* to the CPU - it is (mostly) part of the CPU itself, rather than being accessed via bus
- The cache is therefore significantly smaller than the RAM
- Overall, cache memory can operate 10-100x faster than RAM
- There are different levels of cache memory:
  - L1 (Level 1) - fastest memory (lowest latency), contains the data the CPU is most likely to need. Usually < 1MB (although no standard size), ~100x faster than RAM. Assigned for each core.
  - L2 - slower, but larger in size, such as few MB, ~25x faster than RAM. Assigned for each core.
  - L3 - slower again, larger again ~ a few tens of MB. Can be used by entire CPU, i.e. shared by many cores (see later)

Parallelism

Supercomputer Architectures

- The CPU decides what to store in the cache using *prefetching*
- For example, if we define an array `a[100]` and access the first element, it is very likely we will also access the other elements - the CPU will prefetch the array to the cache
- More generally, `local' data are likely to be in the cache, by which we mean either:
  - Temporal locality - if an address is referenced, it is likely to be referenced again soon
  - Spatial locality - if an address is referenced, nearby addresses are likely to be referenced again soon
- If we are successful in fetching data from the cache, this is a *cache hit*
- If we are unsuccessful, this is a *cache miss* - these can give significant performance penalties

Parallelism

Supercomputer Architectures

Recent example of optimising using cache blocking (from Wu Hyun Sohn, one of James' ex-PhD students!) - this is an optimisation of a code for CMB bispectrum estimation of primordial non-Gaussianity:

for each set of modes (p_1, p_2, p_3) **do**
  for each pixel n **do**
    $\beta^{\rm cub}(i, p_1, p_2, p_3)$ += $m(p_1, n) \cdot m(p_2, n) \cdot m(p_3, n)$
    $\beta^{\rm lin}(i, p_1, p_2, p_3)$ += $C(p_1, p_2, n) \cdot m(p_3, n)$
  end for
end for

- The original algorithm fetches 4 large arrays from RAM, as each of them are too large to be stored in the cache

Parallelism

Supercomputer Architectures

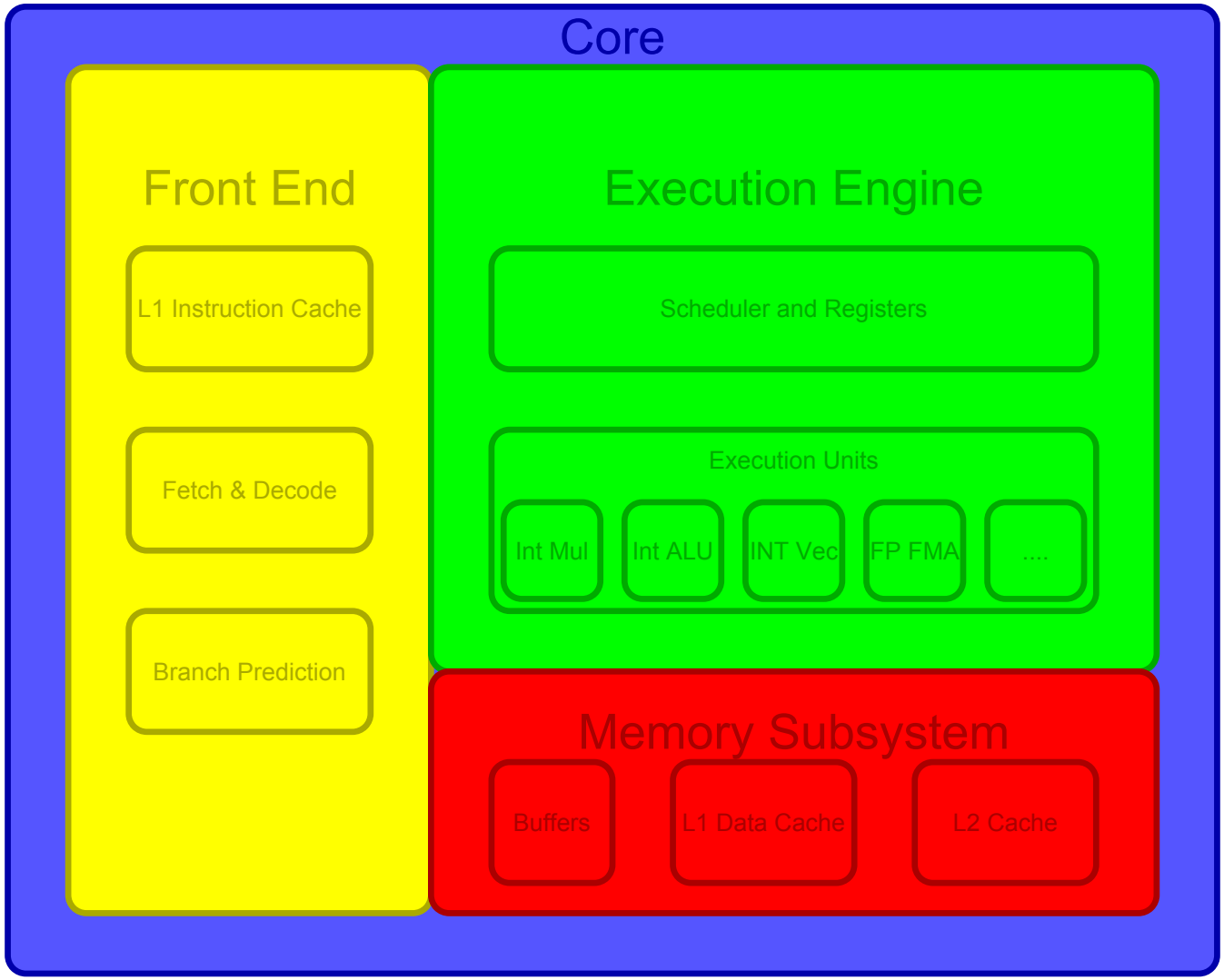
- We can divide these large arrays into blocks that fit inside the cache memory:

for each block b **do**
  for each set of modes (p_1, p_2, p_3) **do**
    for each pixel n' in block **do**
      $\beta^{\text{cub}}(i, p_1, p_2, p_3) \mathrel{+}= m(p_1, n') \cdot m(p_2, n') \cdot m(p_3, n')$
      $\beta^{\text{lin}}(i, p_1, p_2, p_3) \mathrel{+}= C(p_1, p_2, n') \cdot m(p_3, n')$
    end for
  end for
end for

- Here, the new pixel number is calculated as $n' = B \cdot b + n$, where B is the size of each block and $o \leq n < B$
- Data locality is greatly improved, as well as temporal locality

Parallelism

Supercomputer Architectures



Parallelism

Supercomputer Architectures

- Since the introduction of caches in the 1970s, there have been continuous improvements made to CPUs to enhance performance
- These include:
  - *Superscalar capacity* - the ability to execute more than one instruction per cycle
  - Hardware that facilitates *(multiple) pipelines* - detaching the fetching, decoding, execution and writeback stages of an instruction
  - *Vector registers* - large special registers in the CPU that can be subdivided into independent chunks, over which the same operation can be performed (see James' next lecture)
- Development of CPUs has been very effective, however…

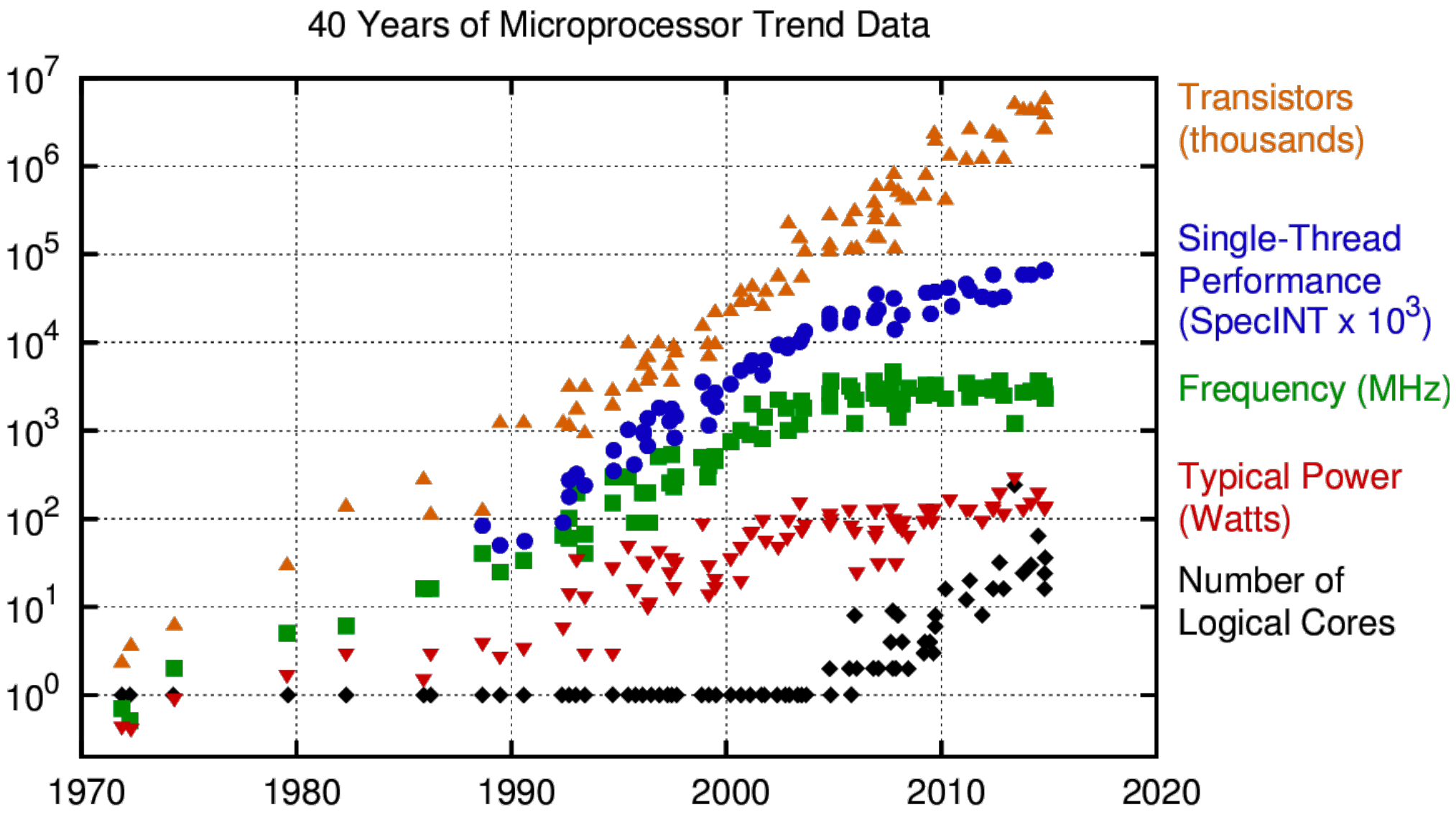
Parallelism

Overview of Parallelisation

- As we have seen, speeding up the CPU ad infinitum is not possible, due to breaking of Dennard's scaling
- This means that we must use *parallelism*, usually over *multiple cores* in order to increase performance
- In short, this means that we split our program (or, more commonly, parts of a program) into smaller chunks, each of which can be run independently
  - Note that this is most effective for *large problems* - for small problems that fit on one core, splitting across multiple processors may not speed up your program (it could even slow it down, due to communication costs)
  - If your problem is inherently not parallelisable, e.g. each operation must be performed in sequence, throwing more cores at it will not give any performance increase - you will be limited by the clock speed
- Why can't we just use one very big core? Both physical (limits on eg. heat dissipation) and business (cost, size) reasons

Parallelism

Overview of Parallelisation



Parallelism

Overview of Parallelisation

Example: parallelisable vs non-parallelisable programs

- Non-parallelisable - calculation of Fibonacci series

```
for (k=0, k < n, k++) {
   F(k+2) = F(k+1) + F(k);}
```

- Parallelisable - calculation of a sum

```
my_sum = 0;
my_first_i = ..;
my_last_i  = ..;
for (my_i = my_first_i; my_i < my_last_i, my_i++) {
   my_x = compute_next_value(..);
   my_sum += my_x; }
```

Parallelism

Overview of Parallelisation

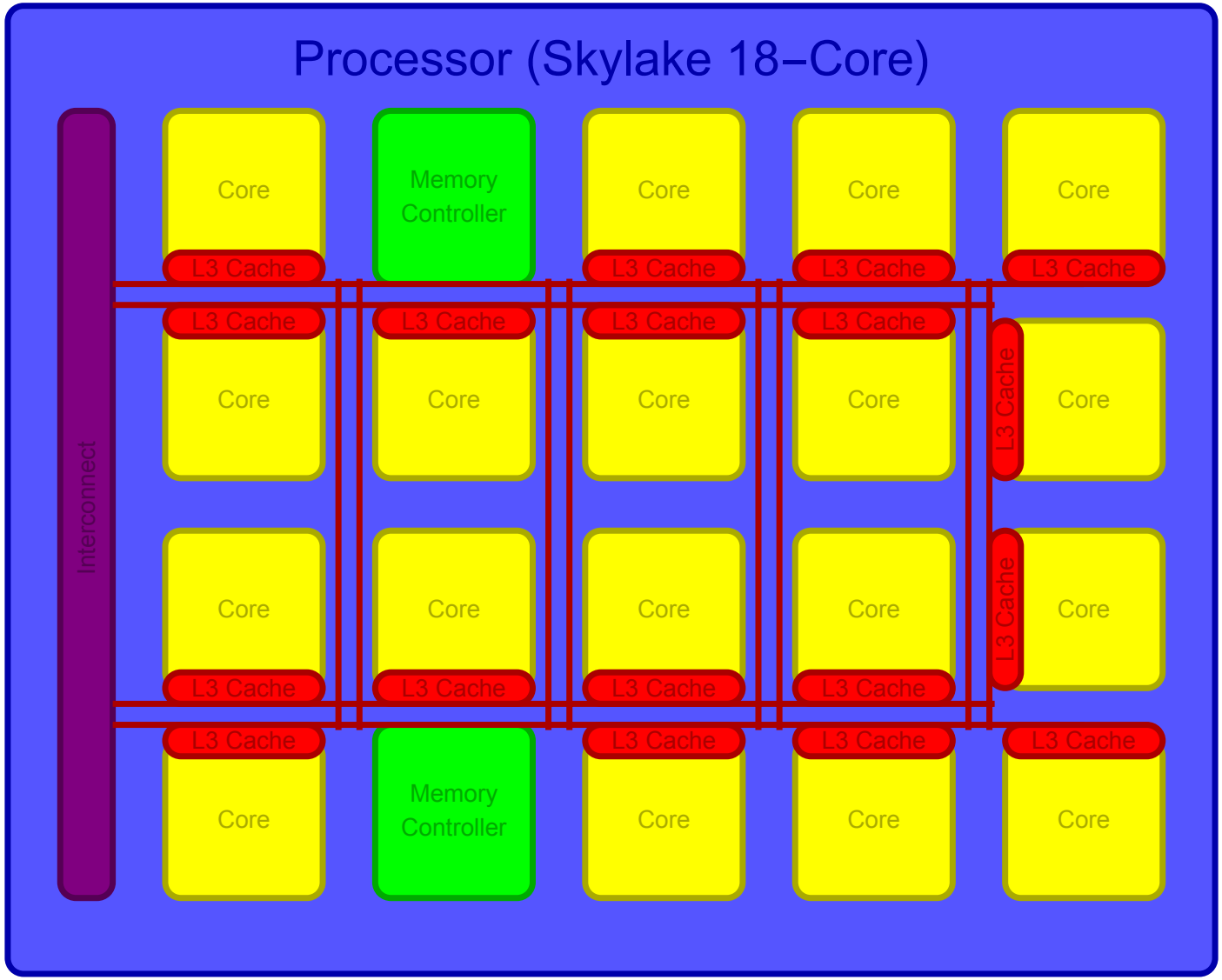
- The potential performance gains from parallelism are given by *Amdahl's law:*

$$S_{\text{latency}}(s) = \frac{1}{(1-p) + \frac{p}{s}}$$

  - S_{latency} - theoretical speedup of a whole program
  - s - speedup from parallelism (or number of processors)
  - p - fraction of the program that can be parallelised

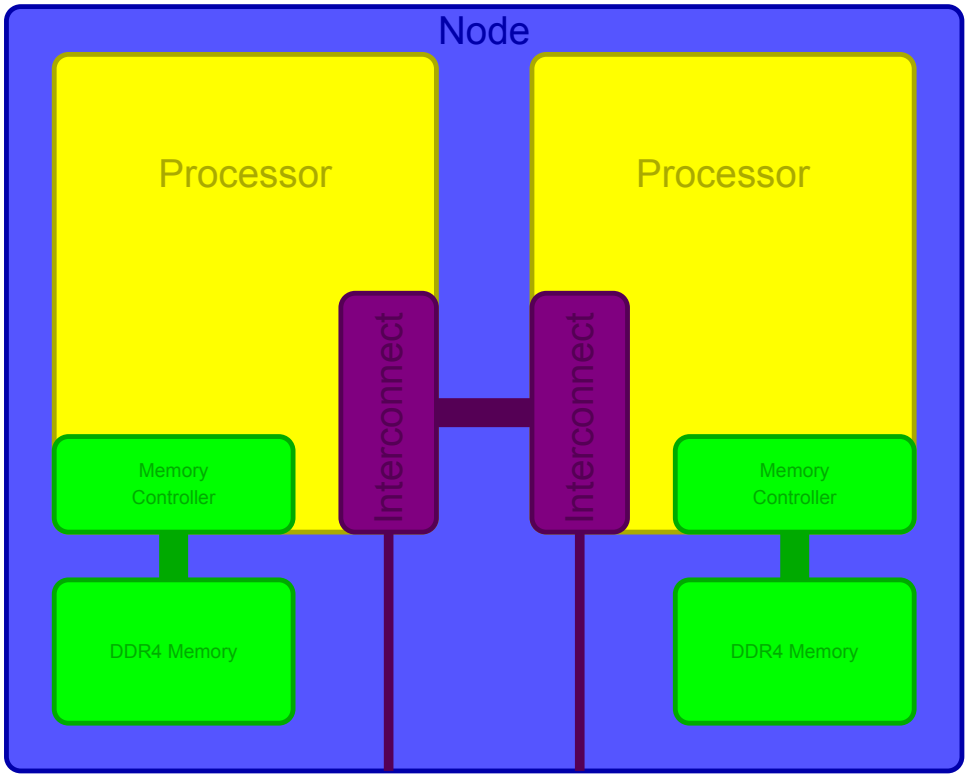
Parallelism

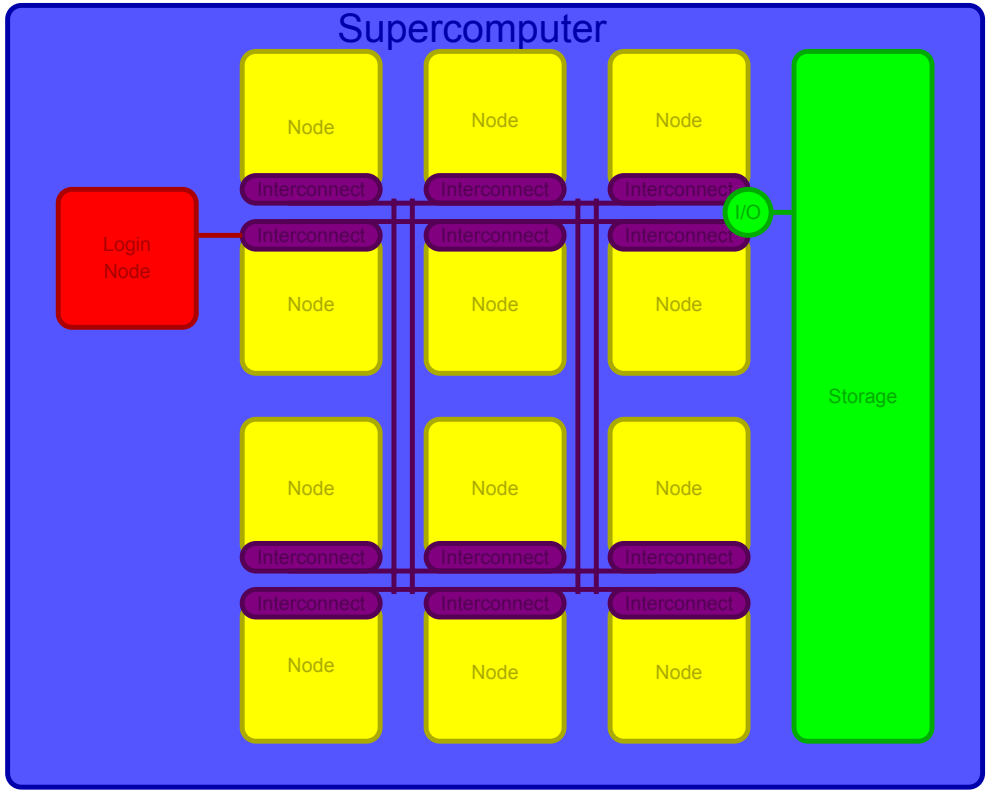
Overview of Parallelisation



Parallelism

Overview of Parallelisation

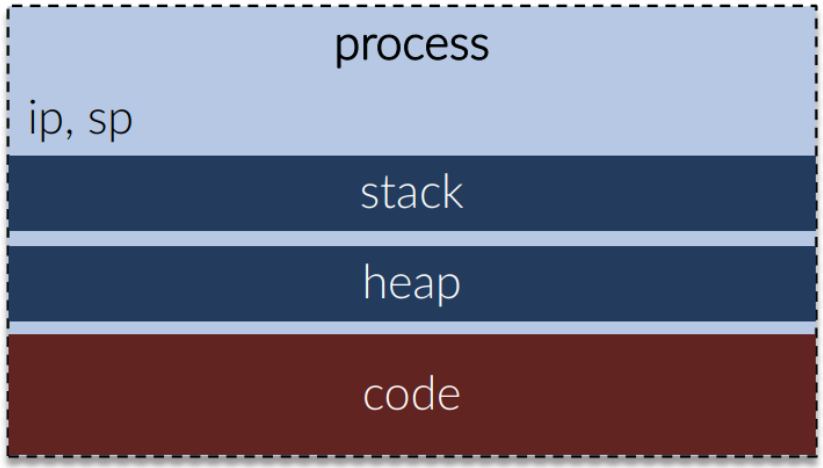


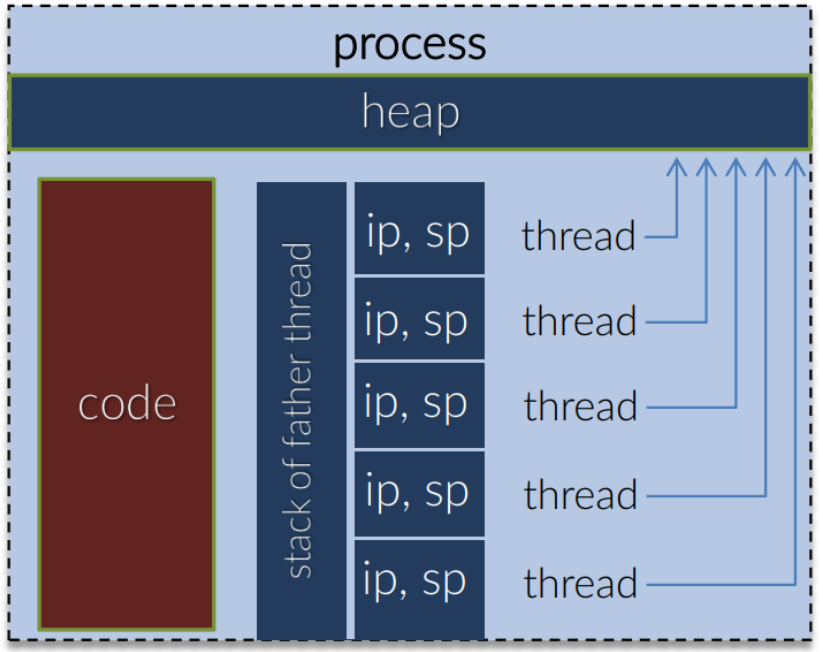


Parallelism

Overview of Parallelisation

- How do we actually implement parallelism?
- Parallelism makes use of two important concepts:
  - *Process* - this is an independent sequence of instructions and all of the resources needed for their execution
  - *Thread* - an independent instance of code execution *within* a process
    - Each thread shares the same code, memory address space and resources to its process
    - Each thread has its own *stack*, but the *heap* is shared between threads which operate in *shared memory*

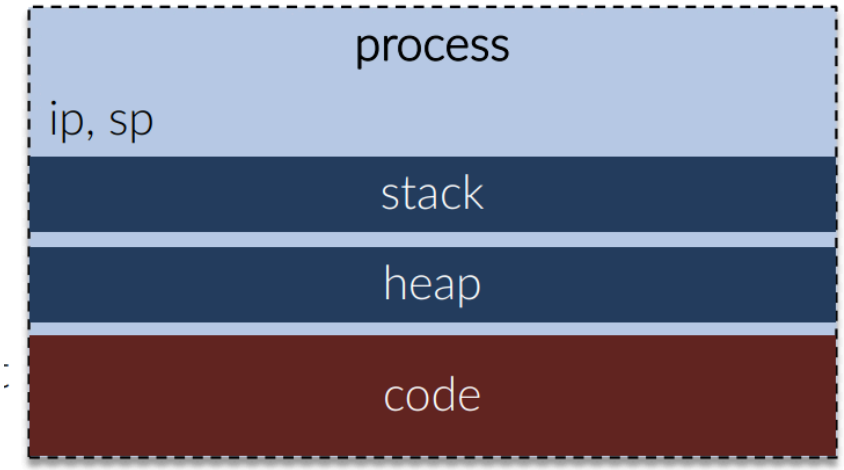


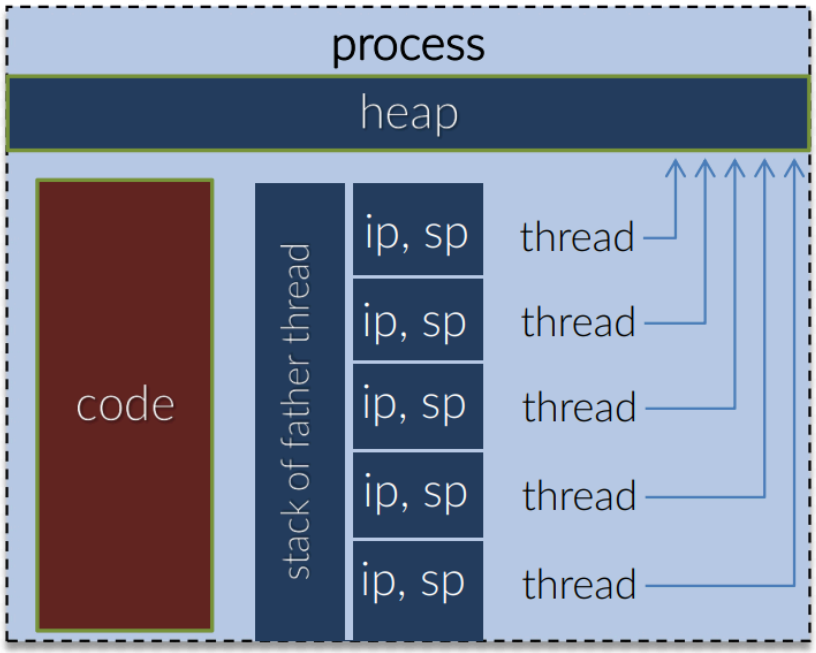


Parallelism

Overview of Parallelisation

- `Stack' and `heap' are both types of memory that are stored in RAM
- What is the stack?
  - `Local' memory that contains local variables of each function
  - Very limited scope - can only be accessed by the current function or its callees
  - Allocated to you by the operating system
  - Used primarily for *static allocation* (ie. at compile time)
  - A lot faster and more optimised
  - (Cf stackoverflow)

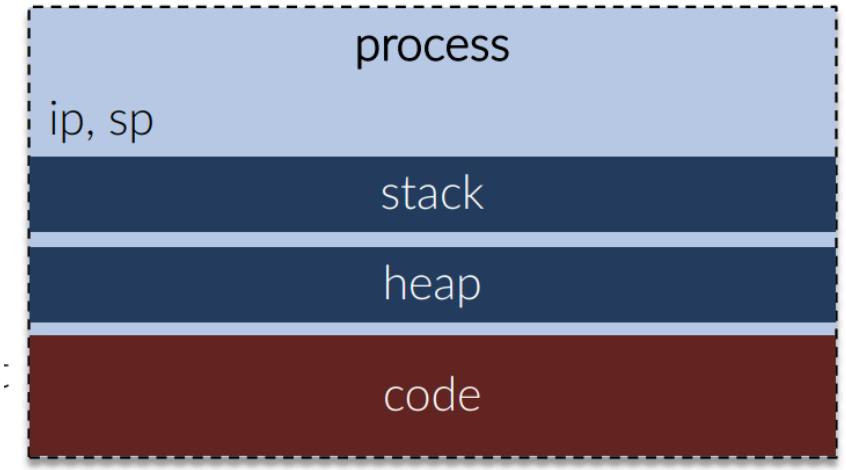


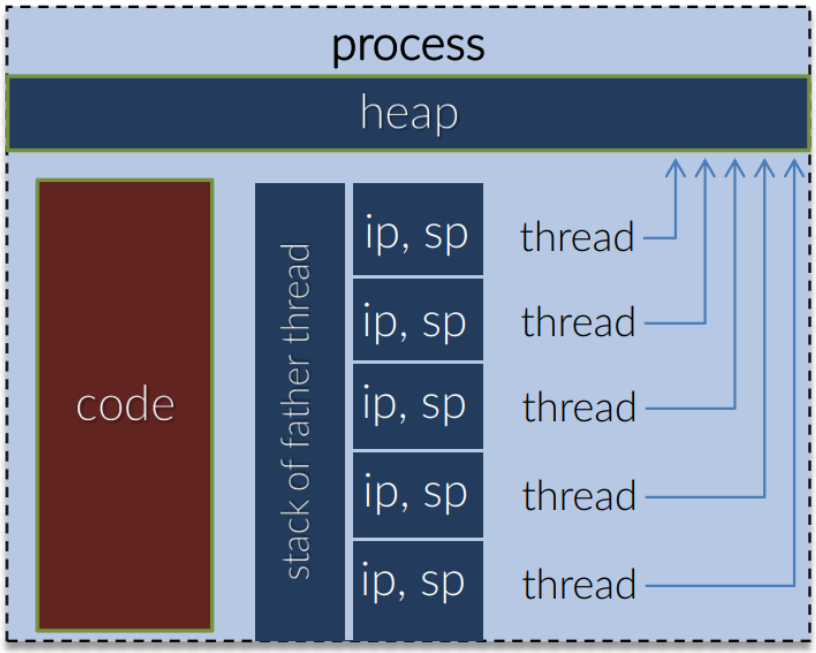


Parallelism

Overview of Parallelisation

- What is the heap?
  - Memory that stores all of your data and global variables
  - Accessible from all functions in all code units
  - Used primarily for *dynamic allocation* (ie. at runtime)
- So, to recap - *processes* have access to both stack and heap memory, *threads* have access to the heap, each with their own stack
- In general, creating threads inside a process is much less costly than creating more processes





Parallelism

Overview of Parallelisation

- In order to parallelise a program, we need to assign different chunks to different threads, different processes, or a combination
- The main API (application programming interface) for using threads for parallelism is *OpenMP* - OPEN specifications for MultiProcessing
  - Allows you to (quite) easily reformat your code using `pragma` statements
- For using processes, the most common method is MPI - Message Passing Interface
  - Usually requires more involved code restructuring with explicit communication between processors
- We usually assign one thread or one process per core (unless we have hyperthreading - multiple threads per core)
- We will cover how to implement OpenMP and MPI in future lectures…

Parallelism

Overview of Parallelisation

- It is not always clear which of the two methods is the most appropriate for a given problem
- One useful `rule' - OpenMP is most useful for running over multiple cores within *one CPU*
- This is because OpenMP does not include a mechanism for sharing data between multiple CPUs
- MPI facilitates the sharing of data between CPUs, so this is usually more appropriate in this case (although MPI is also often used to run over multiple cores within one CPU)
- It is also necessary to keep in mind how much memory your program will require
- Usually, a hybrid approach is the most effective

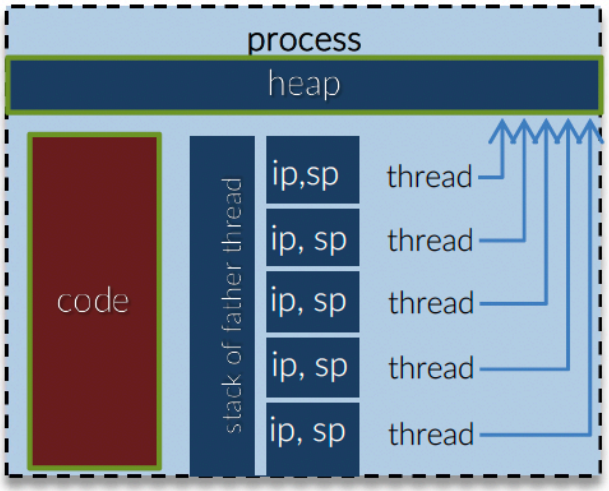
Parallelism

Overview of Parallelisation

Shared-Memory

(e.g. OpenMP)

A unique process that spawns a number of threads. There is a unique memory space that is accessible by all the threads



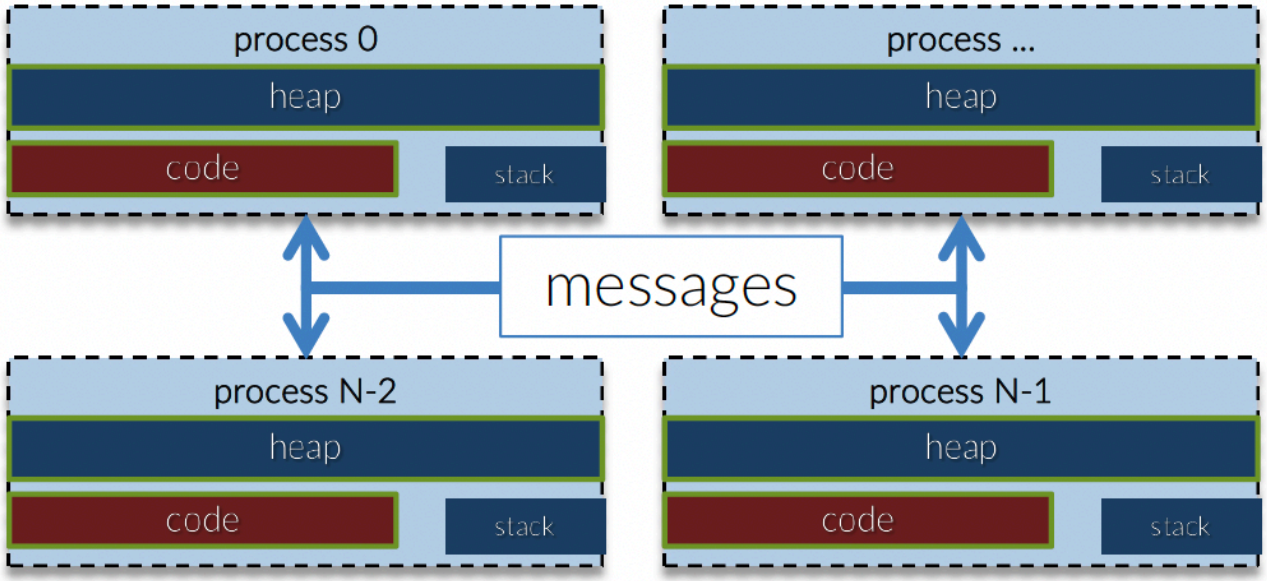
Distributed-Memory

(e.g. MPI)

N processes are created, each with its own copy of the code and its own memory space.

A process ***can not*** access the memory space of another process.

The processes communicate through *messages*.



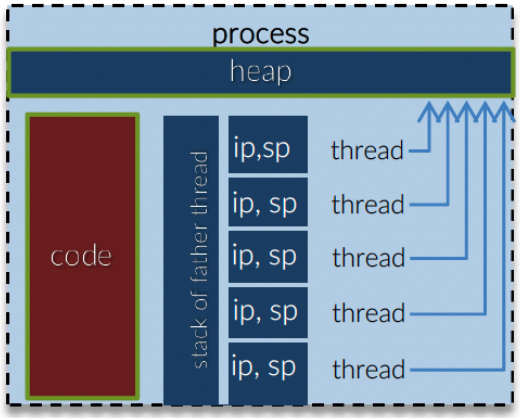
Parallelism

Overview of Parallelisation

Shared-Memory

(e.g. OpenMP)

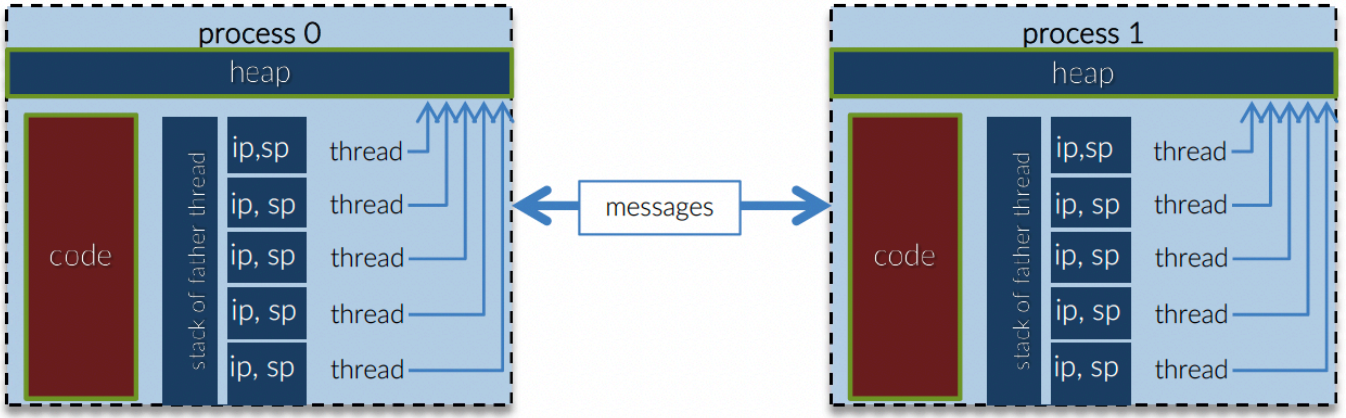
A unique process that spawns a number of threads. There is a unique memory space that is accessible by all the threads



Distributed-Memory

(e.g. MPI) + Shared-Memory

N processes are created, each with its own copy of the code and its own memory space. Each process may spawn a number of threads as in shared-memory.
A process *can not* access the memory space of another process (nor any of its threads can). The processes communicate through *messages*.



Parallelism

Overview of Parallelisation

- The next generation of supercomputing is *zettascale computing*
- Zettascale projects aim for 10^{21} operations per second (flops)
- See e.g. Cambridge Open Zettascale Lab https://www.zettascale.hpc.cam.ac.uk/
- Current state of the art supercomputing happens at *exascale* (10^{18} flops)
- Another important consideration is *energy usage* - one exascale goal is to reach exaflops at 20MW of electric power
  - Moving memory is the most expensive operation
- Some HPC systems eg. LUMI in Finland heat local towns using the extra heat from the system

Parallelism

Threads and OpenMP

Parallelism

Threads and OpenMP

- Some useful resources:
  - Tim Mattson OpenMP notes (https://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf) and videos (https://www.youtube.com/playlist?list=PLLX-Q6B8xqZ8n8bwjGdzBJ25X2utwnoEG)
  - ARCHER2 Youtube courses (https://www.archer2.ac.uk/training/courses/201006-openmp/)
  - Many other online lecture courses

Parallelism

Threads and OpenMP

- Around 20 years ago, the clock rate for CPUs stalled - we can no longer speed up code simply by running on newer chips
- We must now make use of multiple chips to *parallelise* our code
- This means to provide access for one program to run on many cores
- We first concentrate on *threads* - independent instances of code within a process which *share the same memory*

Parallelism

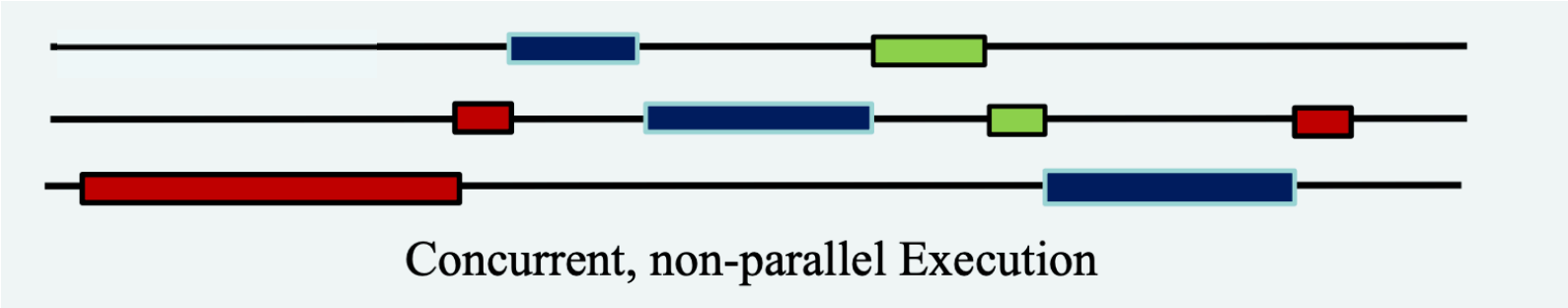
Threads and OpenMP

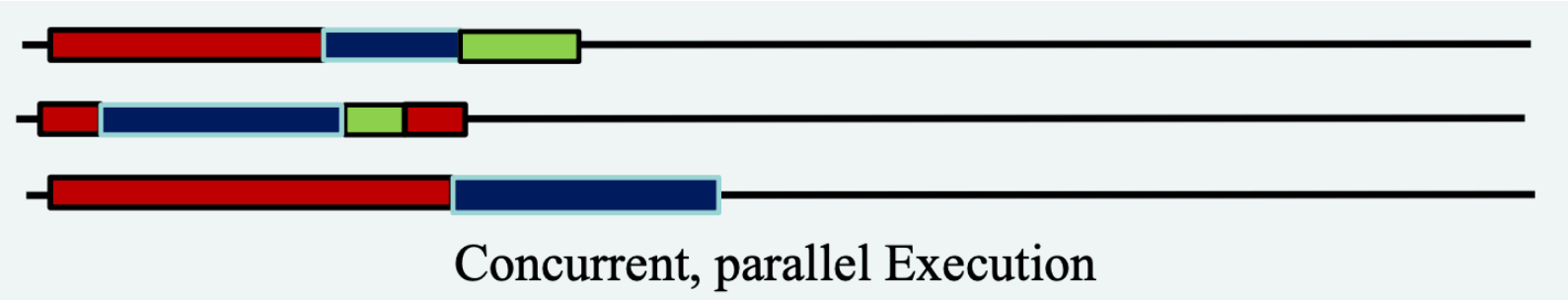
- An important distinction:
  - *Concurrency* - multiple tasks are *logically* active at the same time but *not running* at the same time
    - E.g. checking emails while watching Netflix
    - E.g. C++ `<thread>` library
    - Concurrency *can* reduce wasted clock cycles (reduce latency) - for example, while the program, is fetching something from memory, a program can implement some other part of the program
    - However it is not always associated with speedup

Parallelism

Threads and OpenMP

- *Parallelism* - multiple tasks are *actually* active at the same time
  - Implemented to speed up execution





https://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf

Parallelism

Threads and OpenMP

- Another important concept in parallelism is *scaling* - this is how efficiently the program scales to using more cores
- There are two types of scaling:
  - *Strong scaling* - how does the solution time vary with core number for a *fixed problem*
    - Eg. When we want to *speed up* an existing serial program
  - *Weak scaling* - how does the solution time vary with core number for a fixed problem size *per core*
    - Eg. If we want to run a larger problem in the *same time*
- You will use these terms frequently if you do any kind of HPC development

Parallelism

Threads and OpenMP

- OpenMP is an API that supports shared-memory parallelisation via *multithreading*, developed in ~1990s
- The computational workload is divided up between threads that can run at the same time
- Practically:
  - Directives start with `#pragma omp`
  - We usually need to add `#include <omp.h>`

Parallelism

Threads and OpenMP

Example to demonstrate speedup:

- Set up the Docker container and `connect to running container'
- Compile and run `openmp.cpp` as normal, then time the program with `time`
- Uncomment the line containing `#pragma`
- Compile again, but this time with the flag `-fopenmp`
- Run and time again - also try with different numbers of threads, set using `export OMP_NUM_THREADS=number`
- (But note: are we getting the right answer..?)

Parallelism

Threads and OpenMP

Exercise: Write a multithreaded program that prints “hello world”

```
#include <stdio.h>

int main() {

  int ID = 0;

  printf(“hello(%d)”, ID);

  printf(“world(%d)\n”, ID);

  }
```

Parallelism

Threads and OpenMP

Exercise: Edit to run on multiple threads (remember the compiler flag `-fopenmp`)

```
#include <stdio.h>

int main() {

#pragma omp parallel

   {

   int ID = 0;

   printf("hello(%d)", ID);

   printf("world(%d)\n", ID);

   }

}
```

Parallelism

Threads and OpenMP

Exercise: Edit so we know which output comes from which thread

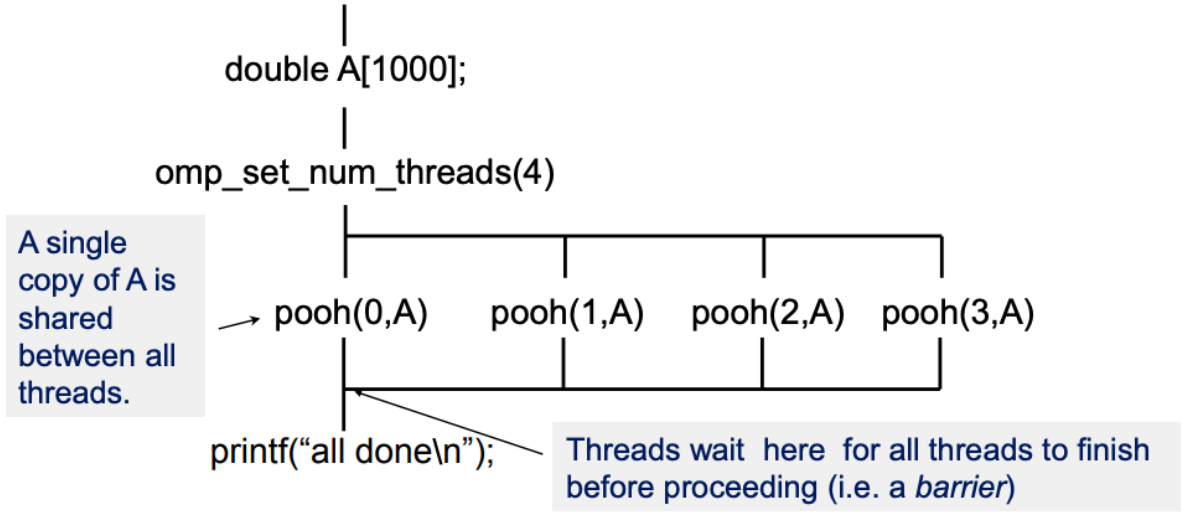
```
#include <stdio.h>
#include <omp.h>


int main() {
#pragma omp parallel
   {
   int ID = omp_get_thread_num();
   printf("hello(%d)", ID);
   printf("world(%d)\n", ID);
   }
}
```

Parallelism

Threads and OpenMP

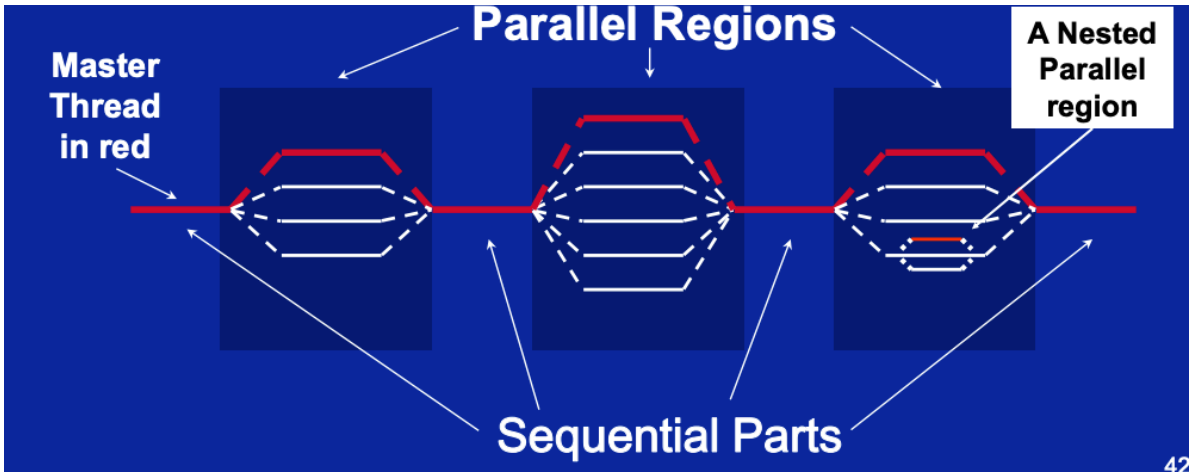
- Here we have the desired output, but it is very muddled
- As we know, OpenMP threads share an address space
- This can lead to *race conditions* - this is where the output of the program changes due to the threads running/finishing at different times



Parallelism

Threads and OpenMP

- We have seen how OpenMP creates a team of threads from the main thread using `#pragma omp parallel`



https://www.openmp.org/wp-content/uploads/Intro_To_OpenMP_Mattson.pdf

- If we want to set the number of threads in the code instead of using environment variables, we use `omp_set_num_threads(num)` before the pragma

- OR we can edit the pragma to `#pragma omp parallel num_threads(num)`

Parallelism

Threads and OpenMP

Exercise: Write a multithreaded program to perform a numerical integration of:

$$\int_0^1 \frac{4}{1+x^2} \mathrm{d}x$$

- We start with the serial version `numerical_integration.cpp`
- How do we want to split this up?
- Solution in `numerical_integration_solution.cpp`
  - Important: pay attention to the local and global variables
- Try timing both of these with more steps

Parallelism

Threads and OpenMP

- `numerical_integration_solution.cpp` is an example of a Single Program Multiple Data (SPMD) algorithm - uses the thread ID to control which tasks to run

- If we time this example, we currently don't get good scaling

- This could be due to *false sharing* - this occurs if independent data elements happen to sit on the *same cache line*

  - This is common in OpenMP when we promote a scalar to an array to index by the thread number

  - The array elements are contiguous in memory, so share cache lines - poor scalability

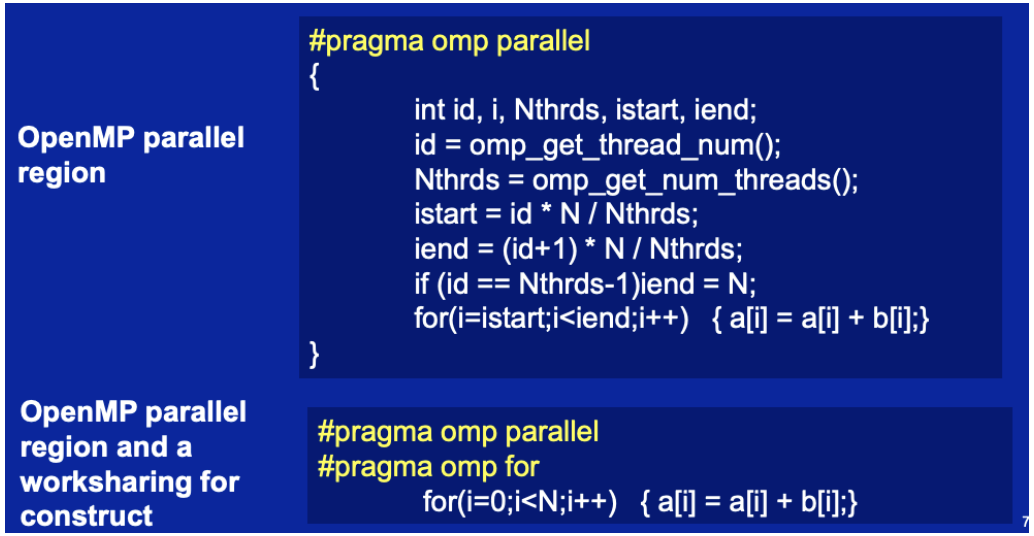
- One way around this is to *pad* the arrays so that we are on distinct cache lines - note though that this is architecture dependent

- Another way is to make use of `private` variables and restructure so there is less sharing of data (see later…)

Parallelism

Threads and OpenMP

- As we saw briefly at the beginning, we can parallelise `for` loops using `#pragma omp parallel for` (or in Fortran, `#pragma omp parallel do`)
- This splits up the loop across threads in a team
- Significantly improves the readability of the code over indexing by thread number



Parallelism

Threads and OpenMP

- Must make sure that the loop iterations are *independent*
- We can also use this for *nested loops*

  ```
  #pragma omp parallel for collapse(2)
  ```

  where here, the number of loops is 2
- If we do have a loop where we are gathering values into one variable, such as eg. `sum += value[i]`, we can use a *reduction*
- In this case, a local copy of each list variable is made and updates, then local copies are reduced to a single value and combined

Parallelism

Threads and OpenMP

Example:

```
double sum=0., average, values[max];
int i;
for(i=0; i<max; i++)

{

  sum += values[max];

}

average = sum/max;
```

Parallelism

Threads and OpenMP

Example:

```
double sum=0., average, values[max];
int i;
#pragma omp parallel for reduction (+:sum)
for(i=0; i<max; i++)

{

  sum += values[max];

}

average = sum/max;
```

Exercise for outside class - parallelise the integration program with a loop construct

Parallelism

Threads and OpenMP

- In a reduction, a local copy of each list (target) variable is made and initialised for each thread
- Updates occur on the local copy
- Local copies are reduced to a single value and combined with the original global value
- The same effect as a reduction can be achieved using other directives eg. `#pragma omp critical` (see next)
- However, reduction is *more scalable*

Parallelism

Threads and OpenMP

- Other ways of controlling thread access are using:
  - Barriers - force each thread to wait until all of the threads have finished executing

    `#pragma omp barrier`

  - Mutual exclusion - define a block of the code that only one thread at a time can execute (sort of defeats the point of adding the parallelism…)

    `#pragma omp critical` or

    `#pragma omp atomic` (for memory updates only)

    - We can also use `#pragma omp critical` to stop false sharing
    - All of the above can be used to prevent race conditions

Parallelism

Threads and OpenMP

```
#pragma omp parallel
{
        int id=omp_get_thread_num();
        A[id] = big_calc1(id);
#pragma omp barrier

        B[id] = big_calc2(id, A);
}
```

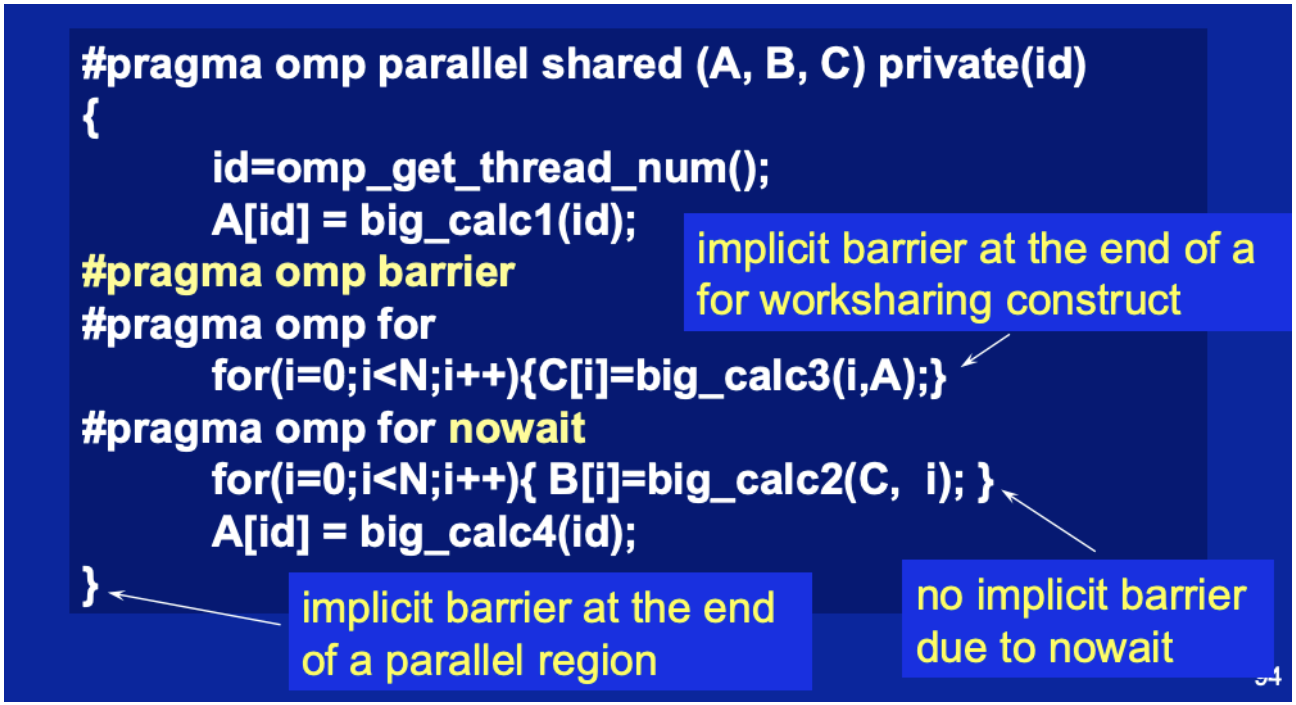
```
float  res;
#pragma omp parallel
{    float B;   int i, id, nthrds;
     id = omp_get_thread_num();
     nthrds = omp_get_num_threads();
      for(i=id;i<niters;i+=nthrds){
           B =  big_job(i);
#pragma omp critical
           res += consume (B);
     }
}
```

Parallelism

Threads and OpenMP

- Barriers are *implicit* at the end of work sharing constructs such as `#pragma omp for`, and at the end of parallel regions

- To get rid of this implicit barrier, we add `nowait`, eg.



Parallelism

Threads and OpenMP

- Variables within a worksharing construct (eg. `#pragma omp for`) are `private` by default
- This can be achieved explicitly using eg. `#pragma omp for private(x)`, where `x` are the variables inside the construct - this creates a new local (uninitialised) copy of `x` for each thread
- Outside working constructs, variables are `shared`
- Default attributes can be overridden using `default(private|shared|none)`
- We can also use:
  - `firstprivate` - initialises from shared variable
  - `lastprivate` - passes last value out to shared variable

Parallelism

Threads and OpenMP

- Consider this example of PRIVATE and FIRSTPRIVATE

```
variables:  A = 1,B = 1, C = 1
#pragma omp parallel private(B)  firstprivate(C)
```

- Are A,B,C local to each thread or shared inside the parallel region?
- What are their initial values inside and values after the parallel region?

Inside this parallel region ...

- "A" is shared by all threads; equals 1
- "B" and "C" are local to each thread.
  - B's initial value is undefined
  - C's initial value equals 1

Following the parallel region ...

- B and C revert to their original values of 1
- A is either 1 or the value it was set to inside the parallel region

Parallelism

Threads and OpenMP

- Other useful features of OpenMP that we will not cover in detail include
  - `#pragma omp master` denotes a structured block that is only executed by the master thread
  - `#pragma omp single` - only executed by one thread (not necessarily the master thread)
  - `#pragma omp sections` - gives a different structured block to each thread
  - Simple and nested locks - these produce a memory fence, flushing all thread visible variables

Parallelism

Threads and OpenMP

Cheat Sheet

Directives:

```
#pragma omp parallel
#pragma omp parallel for (or parallel do for FORTRAN)
#pragma omp parallel for collapse(number of loops)
#pragma omp parallel for reduction (operation:variable name)
#pragma omp parallel shared(A,B,C) private(x)
#pragma omp sections
#pragma omp nowait
```

To avoid if possible (but sometimes necessary):

```
#pragma omp barrier
#pragma omp critical
#pragma omp atomic
#pragma omp master
#pragma omp single
```

Runtime Library functions:

```
omp_get_num_threads()
omp_get_thread_num()
omp_set_num_threads()
omp_get_max_threads()
omp_num_procs()
```

Parallelism

Threads and OpenMP

- In summary:
  - OpenMP can be a relatively simple way to parallelise your program (see last slide), but…
  - THINK before you implement
  - Race conditions and global/private variables can create strange bugs that can be difficult to find
  - False sharing can degrade your performance if not mitigated against

Parallelism

Threads and OpenMP

```
#include <omp.h>
static long num_steps = 100000;        double step;

void main ()
{          int i;    double x, pi, sum = 0.0;
           step = 1.0/(double) num_steps;
#pragma omp parallel for private(x) reduction(+:sum)
           for (i=0;i< num_steps; i++){
                      x = (i+0.5)*step;
                      sum = sum + 4.0/(1.0+x*x);
           }
           pi = step * sum;
}
```

For good OpenMP implementations, reduction is more scalable than critical.

i private by default

Note: we created a parallel program without changing any executable code and by adding 2 simple lines of text!

Parallelism

Message Passing Interface (MPI)

Parallelism

Message Passing Interface (MPI)

- Some useful resources:
  - ARCHER2 Youtube courses (https://www.archer2.ac.uk/training/courses/200514-mpi/)
  - James' git repository - MPI in Python (https://github.com/JamesFergusson/Research-Computing/blob/master/14_Parallelisation.ipynb)
  - https://mpitutorial.com/tutorials/ (some examples from here)
  - Several other online lecture courses

Parallelism

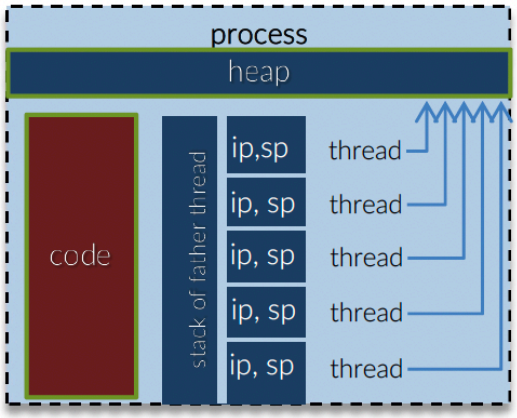
Message Passing Interface (MPI)

- If we want to parallelise our code over multiple cores *without* shared memory, we need to use MPI - recall:

Shared-Memory

(e.g. OpenMP)

A unique process that spawns a number of threads. There is a unique memory space that is accessible by all the threads



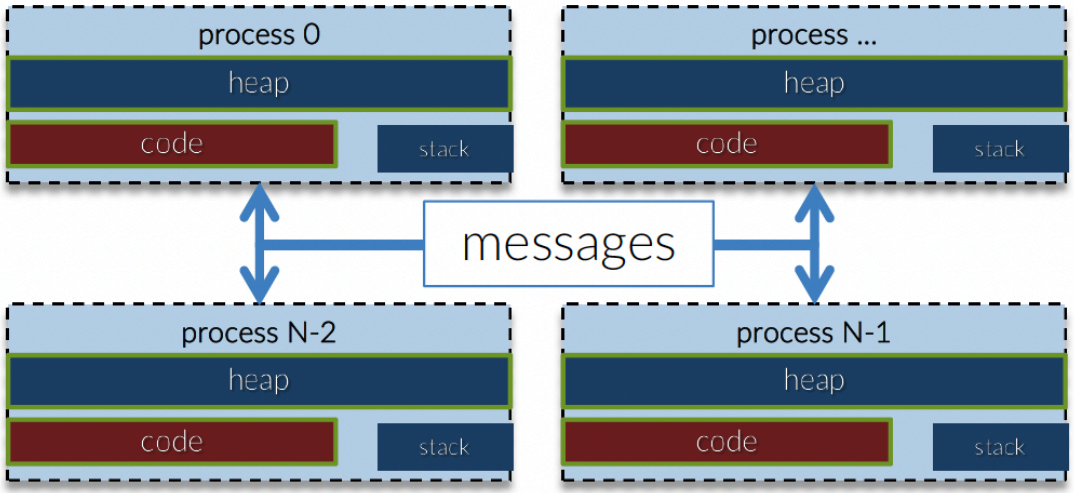
Distributed-Memory

(e.g. MPI)

N processes are created, each with its own copy of the code and its own memory space.

A process ***can not*** access the memory space of another process.

The processes communicate through *messages*.



Parallelism

Message Passing Interface (MPI)

- This means that once we have split up the problem, we need to explicitly *communicate* between cores
- This is done using the routines:
  - `MPI_Init` initialise MPI
  - `MPI_Comm_size` get number of processes
  - `MPI_Comm_rank` get the process ID (rank)
  - `MPI_Send` send data from core
  - `MPI_Recv` receive data to core
  - `MPI_Finalize` close MPI
- This is almost all you need to know to use MPI

Parallelism

Message Passing Interface (MPI)

Example: Compile and run `mpi.cpp`

- Compile with `mpicc mpi.cpp -o whatever_name`
  - `mpicc` is a wrapper around a certain compiler(s)
  - You can see what commands are run using `mpicc -show` (for me, it uses `gcc`)
- To run, use the command `mpirun -np 4 ./whatever_name`
  - The flag `-np` sets the number of *processes*
  - You must ensure that the number of ranks does not exceed the number of available cores
  - If you do assign more ranks than cores, the program will probably run, but the performance will be poor

Parallelism

Message Passing Interface (MPI)

- `MPI_Init` initialises execution environment, takes command line arguments (always keep these arguments)
- `MPI_COMM_WORLD` is defined by `mpi.h` and designates processes in the MPI job
- Each statement executes independently in each process
- As with OpenMP, there is no defined output order
- Note we have also included the library `<mpi.h>` and installed `mpich` (see Dockerfile)

Parallelism

Message Passing Interface (MPI)

- The previous example performed the same command for each rank - usually, we want to split a given problem between several cores
- We therefore require the processes running on different cores to communicate using:
  - `MPI_Send(void* data, int count, MPI_Datatype data_type, int destination, int tag, MPI_Comm communicator)`
  - `MPI_Recv(void* data, int count, MPI_Datatype data_type, int source, int tag, MPI_Comm communicator, MPI_Status* status)`
- Although this looks like a lot to remember, most MPI calls use the same syntax (see example)

Parallelism

Message Passing Interface (MPI)

- The MPI data types are mostly just the normal data types you use in the language you are coding, with `MPI_` in front
  - Eg. `MPI_INT`, `MPI_FLOAT`
- We can even choose which ranks will send/receive certain chunks of work

Example 1: Compile and run `mpi_send_recv.cpp`

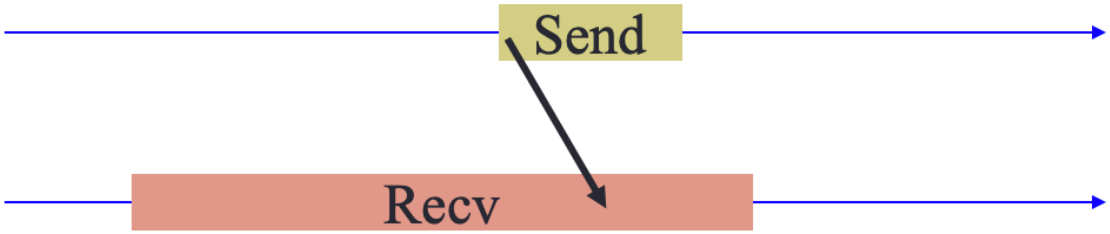
- Change the code so that process 0 sends a message to all other processes (not just 1)

Example 2: Compile and run `mpi_ping_pong.cpp`

Parallelism

Message Passing Interface (MPI)

- In the ping pong example, we notice that some signals are `received' before they are sent - how is this possible?



- If the (blocking) receive is posted before its matching send, the receiving task must wait until the data is sent

  - We will come onto blocking…

Parallelism

Message Passing Interface (MPI)

- With `MPI_Send` and `MPI_Recv`, the instruction is complete when it is *safe to change/access the data we sent/received*
- We can choose either to
  - Start a communication and wait for it to complete - *blocking*
  - Start a communication and return control to the main problem - *non-blocking*
    - This requires us to *check for completion* before we can change/ access the data we sent/received
- `MPI_Send` and `MPI_Recv` are both blocking operations
- These can be unsafe and lead to deadlocks if the message passing cannot be completed
  - Eg. `MPI_Send` is called before `MPI_Recv` for a previous exchange

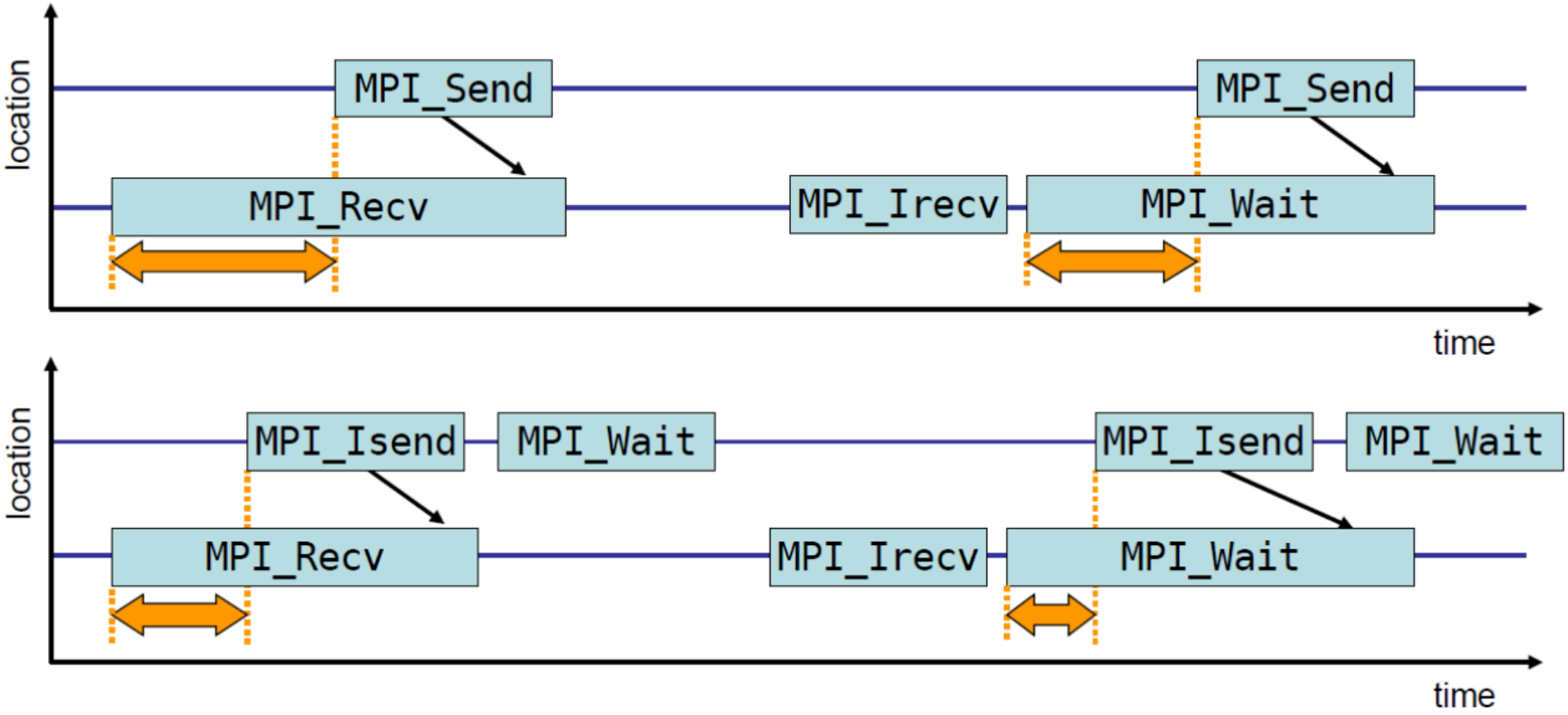
Parallelism

Message Passing Interface (MPI)

- Non-blocking allows separation between the initiation of a communication and the completion
- This can be more efficient but the programmer must also make sure to check for completion
- For these, we instead use `MPI_Isend` and `MPI_Irecv`
  - `MPI_Isend(void* data, int count, MPI_Datatype data_type, int destination, int tag, MPI_Comm communicator, MPI_Request *req)`
  - `MPI_Irecv(void* data, int count, MPI_Datatype data_type, int source, int tag, MPI_Comm communicator, MPI_Status* status, MPI_Request *req)`
- The I stands for `immediate’

Parallelism

Message Passing Interface (MPI)



http://www.morrisriedel.de/wp-content/uploads/2019/10/High-Performance-Computing-Lecture-5-1-Understanding-MPI-Communicators-and-Data-Structures.pdf

Parallelism

Message Passing Interface (MPI)

- If using `MPI_Isend` and `MPI_Irecv`, we can use

  ```
  MPI_Wait(MPI_Request *req, MPI_Status *status)
  ```

  to wait until the communication pointed to by `req` is complete

- We can also use

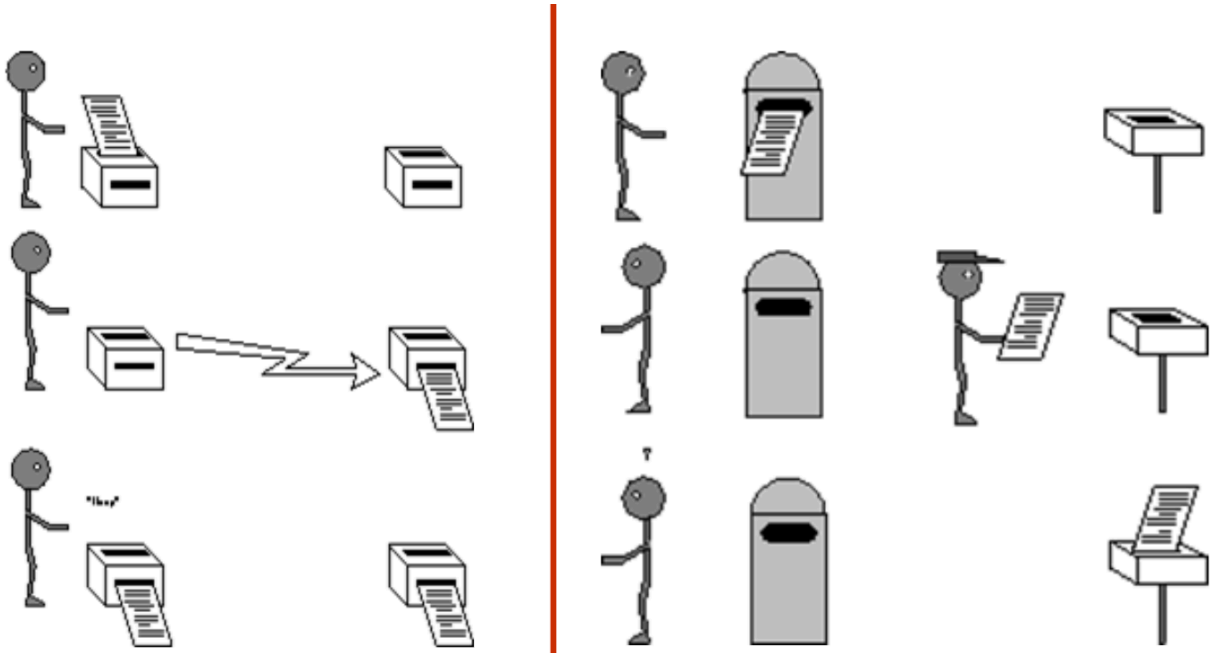
  ```
  MPI_Test(MPI_Request *req, int *flag, MPI_Status
  *status)
  ```

  to test whether the message has been completed

Parallelism

Message Passing Interface (MPI)

- `MPI_Isend` and `MPI_Irecv` can use either:
  - *Synchronous* communication - the sender blocks further operations until an acknowledgement or response is received
  - *Asynchronous* communication - the sender continues execution without waiting for an acknowledgement



Parallelism

Message Passing Interface (MPI)

- Some other options are:
  - `MPI_Bsend` (buffer send) - always completes, irrespective of receiver
  - `MPI_Ssend` (synchronous send) - only completes when the receive has completed, guarantees the buffer passed can be safely reused
  - `MPI_Rsend` (ready send) - always completes, irrespective of whether the receive has completed
- For comparison, standard send and receives
- All of the above can be used as blocking or non-blocking

Parallelism

Message Passing Interface (MPI)

- Sending and receiving large data can be done using the datatype `MPI_BYTE`
- For example (from https://community.intel.com/t5/Intel-oneAPI-HPC-Toolkit/MPI-Code-hangs-when-send-recv-large-data/m-p/1081072):

```
int main(){
   int length = MSG_LENGTH;  // In bytes
   char* buf = malloc(length); // Memory for 'length' number of characters
   int size, rank;

   MPI_Init(0, NULL);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   if(rank == 0){
      MPI_Send(buf, length, MPI_BYTE, 1, 0, MPI_COMM_WORLD); //passing
      printf("Sent");                        // 'length' number of bytes
   }else{
      MPI_Recv(buf, length, MPI_BYTE, 0, 0, MPI_COMM_WORLD,
MPI_STATUS_IGNORE);
      printf("Received");
   }
   free(buf);
   return 0;
}
```

Parallelism

Message Passing Interface (MPI)

- In practice, sending *large* datasets using blocking `MPI_Send` and `MPI_Recv` is *more likely* to cause deadlocks

  Example deadlock (remember send does not complete until the corresponding receive is posted and vice versa):

```
if (rank == 0) {
     MPI_Send(..., 1, tag, MPI_COMM_WORLD);
     MPI_Recv(..., 1, tag, MPI_COMM_WORLD, &status);
} else if (rank == 1) {
     MPI_Send(..., 0, tag, MPI_COMM_WORLD);
     MPI_Recv(..., 0, tag, MPI_COMM_WORLD, &status);
}
```

- This is because of the MPI implementation, which sometimes allows the receive to be posted before the send is completed for smaller data (eager send)

  - For larger data, it is not safe to keep buffered copies of the message in memory, so the send must wait for the matching receive

- *(*This quite common, see eg. https://stackoverflow.com/questions/72827398/mpi-send-and-receive-large-data-to-self, https://stackoverflow.com/questions/15833947/mpi-hangs-on-mpi-send-for-large-messages)

Parallelism

Message Passing Interface (MPI)

- The other most important operations for MPI are *collective operations*
- All collective operations are *blocking*
- Some of the most commonly used examples are:
  - `MPI_Bcast`
  - `MPI_Scatter`
  - `MPI_Gather`
  - `MPI_Allgather`
  - `MPI_Reduce`
  - `MPI_Allreduce`

Parallelism

Message Passing Interface (MPI)

- `MPI_Bcast` - one-to-all communication where the same data is sent from the root process to all others in the communicator `MPI_COMM_WORLD`

Example: Compile and run `mpi_bcastcompare.cpp`

- Demonstrates that `MPI_Bcast` is more efficient than the same operation using `MPI_Send` and `MPI_Recv`
- This example also uses `MPI_Barrier(MPI_Comm communicator)`
  - This stops processes until all processes within a communicator reach the barrier
  - Mostly used for measuring performance and load balancing (kills parallel performance)
- We also see the function `MPI_Wtime()` which can be used to time parallel programs

Parallelism

Message Passing Interface (MPI)

- `MPI_Scatter` - one-to-all communication where *different* data, e.g. parts of an array, is sent from the root process to all others in the communicator `MPI_COMM_WORLD`

- This time we need to define send and receive data and properties

  - ```
    MPI_Scatter(void* send_data, int send_count,
    MPI_Datatype send_datatype, void* recv_data,
    int recv_count, MPI_Datatype recv_datatype, int
    root, MPI_Comm communicator)
    ```
    ```

Parallelism

Message Passing Interface (MPI)

- `MPI_Gather` - one-to-all communication where *different* data is collected by the root process from all others in the communicator `MPI_COMM_WORLD`
- This is the opposite of `MPI_Scatter`, but has a similar structure
  - ```
    MPI_Gather(void* send_data, int send_count,
    MPI_Datatype send_datatype, void* recv_data,
    int recv_count, MPI_Datatype recv_datatype,
    int root, MPI_Comm communicator)
    ```

Example: Compile and run `mpi_scatter_gather.cpp`
    ```

Parallelism

Message Passing Interface (MPI)

- `MPI_Allgather` is the same as `MPI_Gather` but has *no root*
  - `MPI_Allgather(void* send_data, int send_count, MPI_Datatype send_datatype, void* recv_data, int recv_count, MPI_Datatype recv_datatype, MPI_Comm communicator)`
- Usually slightly smoother to implement

Example: Compile and run `mpi_scatter_allgather.cpp`

Parallelism

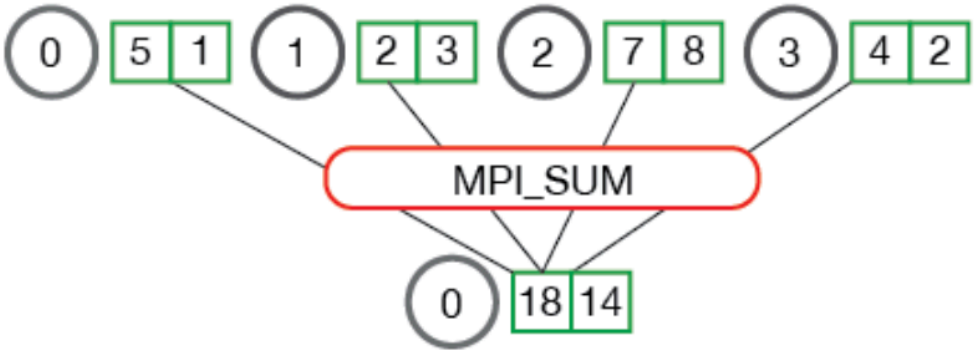
Message Passing Interface (MPI)

- `MPI_Reduce` and `MPI_Allreduce` are reduction operations, similar to those used in OpenMP
  - `MPI_Reduce(void* send_data, void* recv_data, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm communicator)`
  - `MPI_Allreduce(void* send_data, void* recv_data, int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm communicator)`
- These operations collect data from each process and reduce it to a single value
- The result is sorted on the root process (`MPI_Reduce`) or all processes (`MPI_Allreduce`)

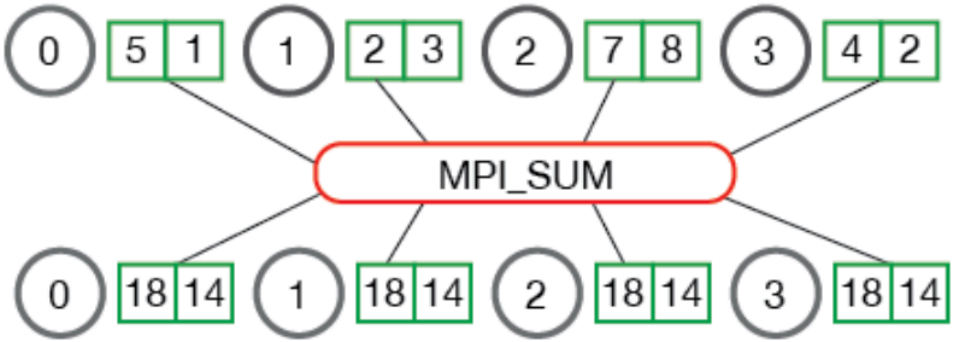
Parallelism

Message Passing Interface (MPI)

MPI_Reduce



MPI_Allreduce



https://mpitutorial.com/tutorials/mpi-reduce-and-allreduce/

Note: If there is more than one element per process (i.e. an array), the reduction happens on a per-element basis

Parallelism

Message Passing Interface (MPI)

- Predefined collective operations include:
  - `MPI_MAX` - returns maximum element
  - `MPI_MIN` - returns minimum element
  - `MPI_SUM` - sums the elements
  - `MPI_PROD` - multiplies all elements
  - `MPI_LAND` - performs logical *and* across the elements
  - `MPI_LOR` - performs logical *or* across the elements
  - `MPI_BAND` - performs bitwise *and* across the bits of the elements
  - `MPI_BOR` - performs bitwise *or* across the bits of the elements
  - `MPI_MAXLOC` - returns maximum value and rank of the process
  - `MPI_MINLOC` - returns minimum value and rank of the process

Example: Compile and run `mpi_reduce_allreduce.cpp`

Parallelism

Message Passing Interface (MPI)

- Now we have covered how to *implement* MPI, we can address the best *methods* to use it
- The best way to parallelise a given code will be quite specific to the problem
- There are a few basic ‘patterns’ that are commonly used:
  - Task parallelism (independent tasks)
  - Domain decomposition
  - Divide and conquer
  - Recursive data
  - Pipelines
- Whichever pattern we use, we need to keep in mind *scaling* (as previously covered) and *load balancing* (ensuring the CPU does roughly the same amount of work)

Parallelism

Message Passing Interface (MPI)

Task Parallelism:

- Divide up independent tasks over the available ranks
- Eg. Splitting a loop, similar to the OpenMP integration example

See Task Parallelism in Python (https://github.com/JamesFergusson/Research-Computing/blob/master/14_Parallelisation.ipynb)

  - Pattern 1 - splits up loop into chunks, calculates partial sum for each rank, combines result using reduce
  - Pattern 2 (multiple loops) - as above, flatten the loops to ensure load balancing
  - Pattern 3 (very large jobs with uneven workload) - use parent/child setup to even out the workloads

Parallelism

Message Passing Interface (MPI)

Domain Decomposition:

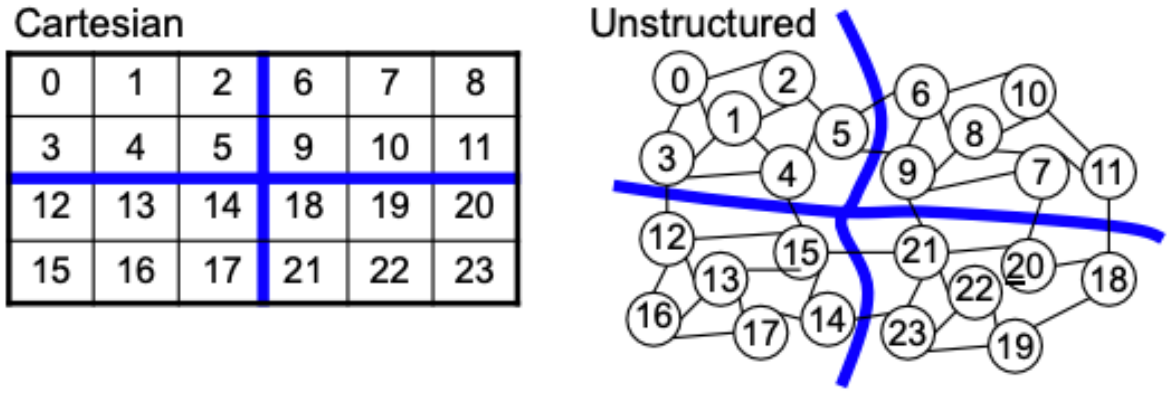
- This is a *fundamental* technique in scientific programming
- We perform a numerical simulation on a domain of a given shape, eg. a 3D grid
- The domain is discretised, usually using a *mesh* or *particles*
- The work and memory can be split among N processors
- Domains can have complex shapes

Parallelism

Message Passing Interface (MPI)

Domain Decomposition:

- It is important that:
  - All processors have equal amounts of work - *load balanced*
  - The number of communication steps between the processors is reduced - *communication scheduling*



Examples with 4 sub-domains

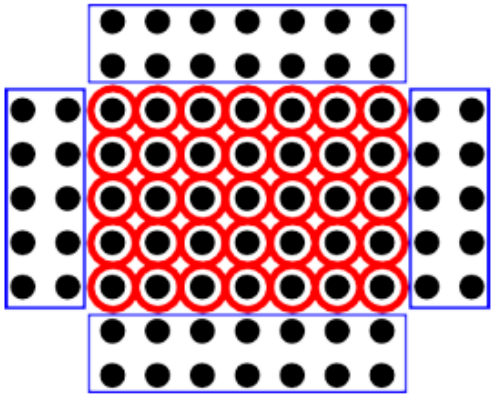
https://github.com/Foundations-of-HPC/Foundations_of_HPC_2021/tree/main/MPI

Parallelism

Message Passing Interface (MPI)

Domain Decomposition:

- For Cartesian coordinates, we usually have a regular mesh
- Values at the mesh nodes are updated to numerically solve given equations, eg. finite difference methods
- It is necessary in this case to define the behaviour at the boundaries of subdomains
- This can be done using ‘ghost’ zones, which must be passed to ‘adjacent' processes



Parallelism

Message Passing Interface (MPI)

Domain Decomposition:

- We generally perform decomposition as follows:
  - Split the domain into blocks (bear in mind which order is most scalable)
  - Assign blocks to MPI-processes one-to-one
  - Provide a map of neighbours to each process
  - Modify your code so that it only updates a single block
  - Insert communication subroutine calls where needed
  - Adjust the boundary condition code
  - Implement ghost cells

Parallelism

Message Passing Interface (MPI)

Examples: See Task Parallelism in Python (https://github.com/JamesFergusson/Research-Computing/blob/master/14_Parallelisation.ipynb)

- Pattern 4 - splitting up a matrix multiplication
- Patterns 5 and 6 - Cartesian and graph topologies

Note: it is important to check these codes for *convergence* - i.e. you obtain the same result with a more refined grid

Parallelism

Message Passing Interface (MPI)

Divide and Conquer:

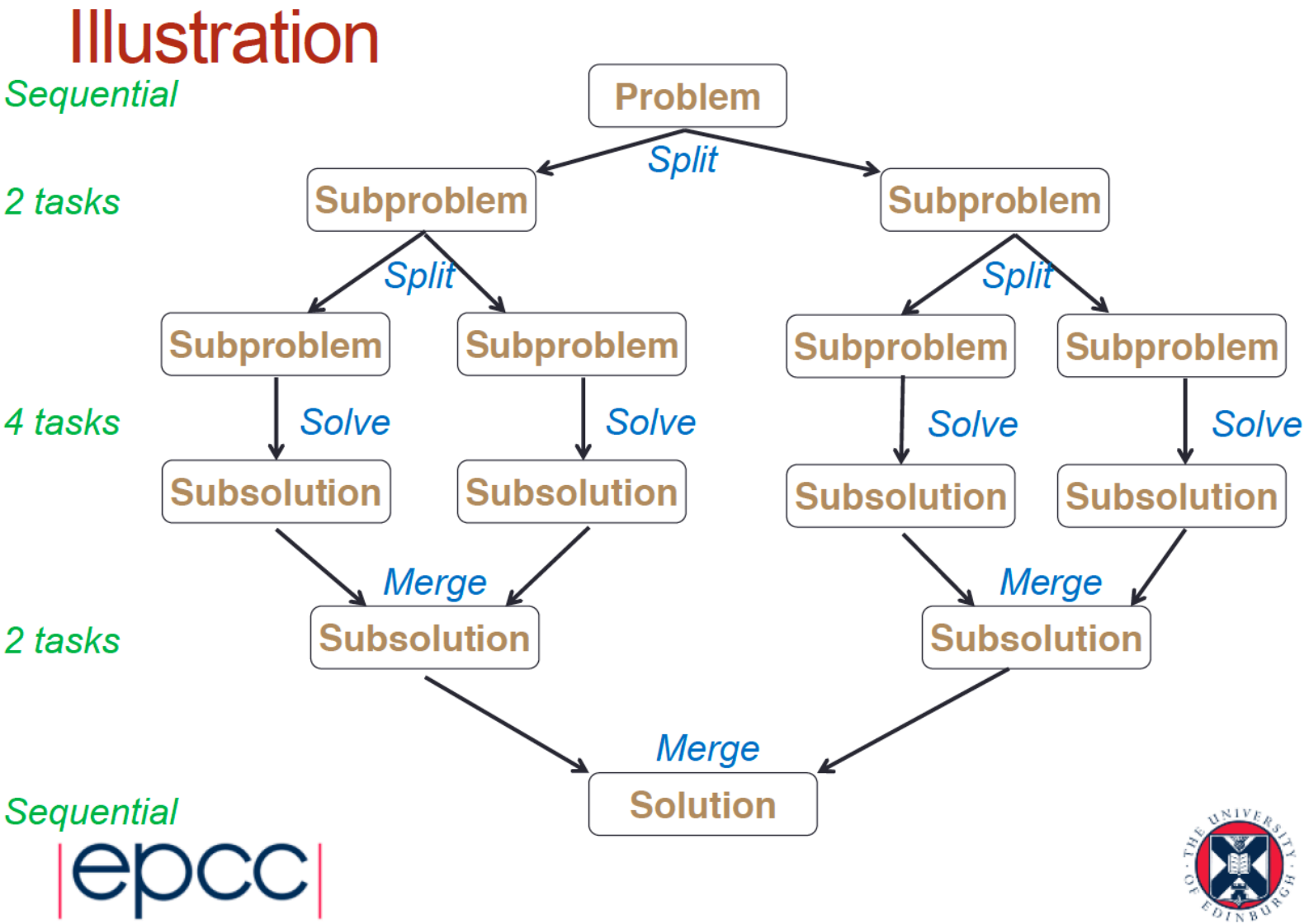
- This is used when dividing or recombining tasks/data in a non-trivial way
- We must scatter and gather the data iteratively

See Task Parallelism in Python (https://github.com/JamesFergusson/Research-Computing/blob/master/14_Parallelisation.ipynb)

- Pattern 7 - sorting a long list

Parallelism

Message Passing Interface (MPI)



epcc

THE UNIVERSITY OF EDINBURGH

Parallelism

Message Passing Interface (MPI)

Recursive Data:

- We may need to perform recursive operations on data which are inherently serial

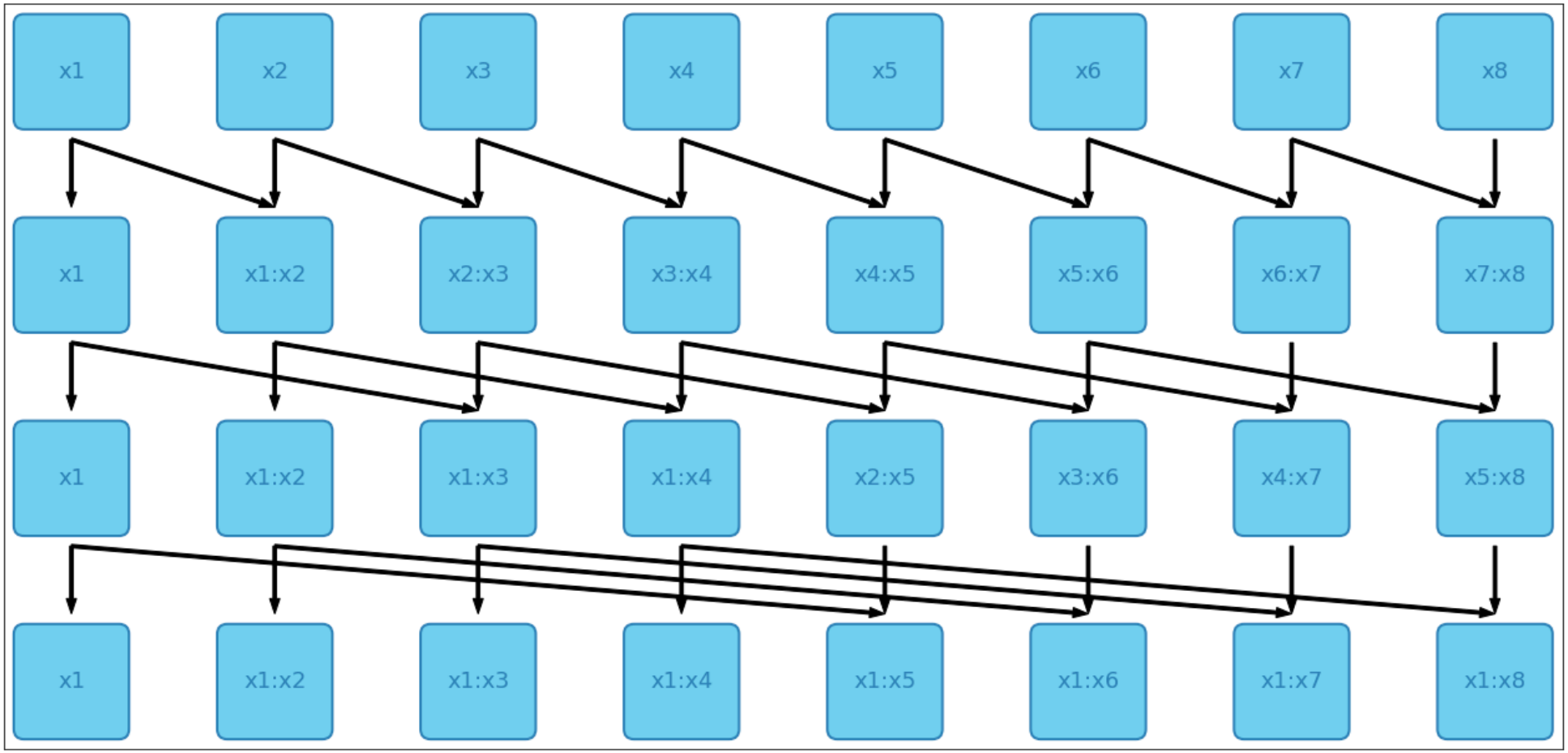
- We can recast these problems to expose concurrency - this can result in a slower algorithm but which runs faster once enough cores are used

See Task Parallelism in Python (https://github.com/JamesFergusson/Research-Computing/blob/master/14_Parallelisation.ipynb)

  - Pattern 8 - calculating a cumulative sum for a list

Parallelism

Message Passing Interface (MPI)



Parallelism

Message Passing Interface (MPI)

Pipelines:

- Data flows through the ranks with different operations applied at each stage
- Can work well for heavy compute algorithms or GPUs
- Eg. We can input a stream of vectors that can be multiplied by several matrices with different operations in between - pass the vectors along a production line while the matrices remain fixed
- The communication is non-blocking send and recvs between the ranks

Parallelism

Threads and OpenMP

- In summary:
  - MPI is essential if you wish to parallelise your program over multiple distributed memory cores
  - As with OpenMP, you must THINK before you implement
  - Important things to look out for are deadlocks, and other issues that can degrade scalability

Parallelism

GPUs and Heterogeneous Programming

Parallelism

Threads and OpenMP

- Some useful resources:
  - Codeplay SYCL Academy (https://github.com/codeplaysoftware/syclacademy/tree/main/Lesson_Materials)
  - Many other online lecture courses

Visualisation