How to Parallelise an Application

How to Parallelise a Code

Designing and developing parallel programs has characteristically been a highly manual process

- the programmer is typically responsible for both identifying and actually implementing parallelism
- time consuming, complex, error-prone and iterative process

Tools have been available to assist the programmer with converting serial programs into parallel programs

- e.g. Vector compilers were successful in the 80's
  - so why not parallel?

The most common type of tool used to automatically parallelize a serial program is a parallelizing compiler or pre-processor

How to Parallelise a Code

Parallelizing compiler – two flavours:

- Fully Automatic
  - the compiler analyzes the source code and identifies opportunities for parallelism
  - analysis includes identifying inhibitors to parallelism and possibly a cost weighting on the benefit OR NOT to the performance
  - **loop parallel** paradigm is the strategy usually adopted
- Programmer Directed
  - using directives or possibly compiler flags
  - the programmer explicitly tells the compiler how to parallelize the code
  - can be used in conjunction with some degree of automatic parallelization

How to Parallelise a Code

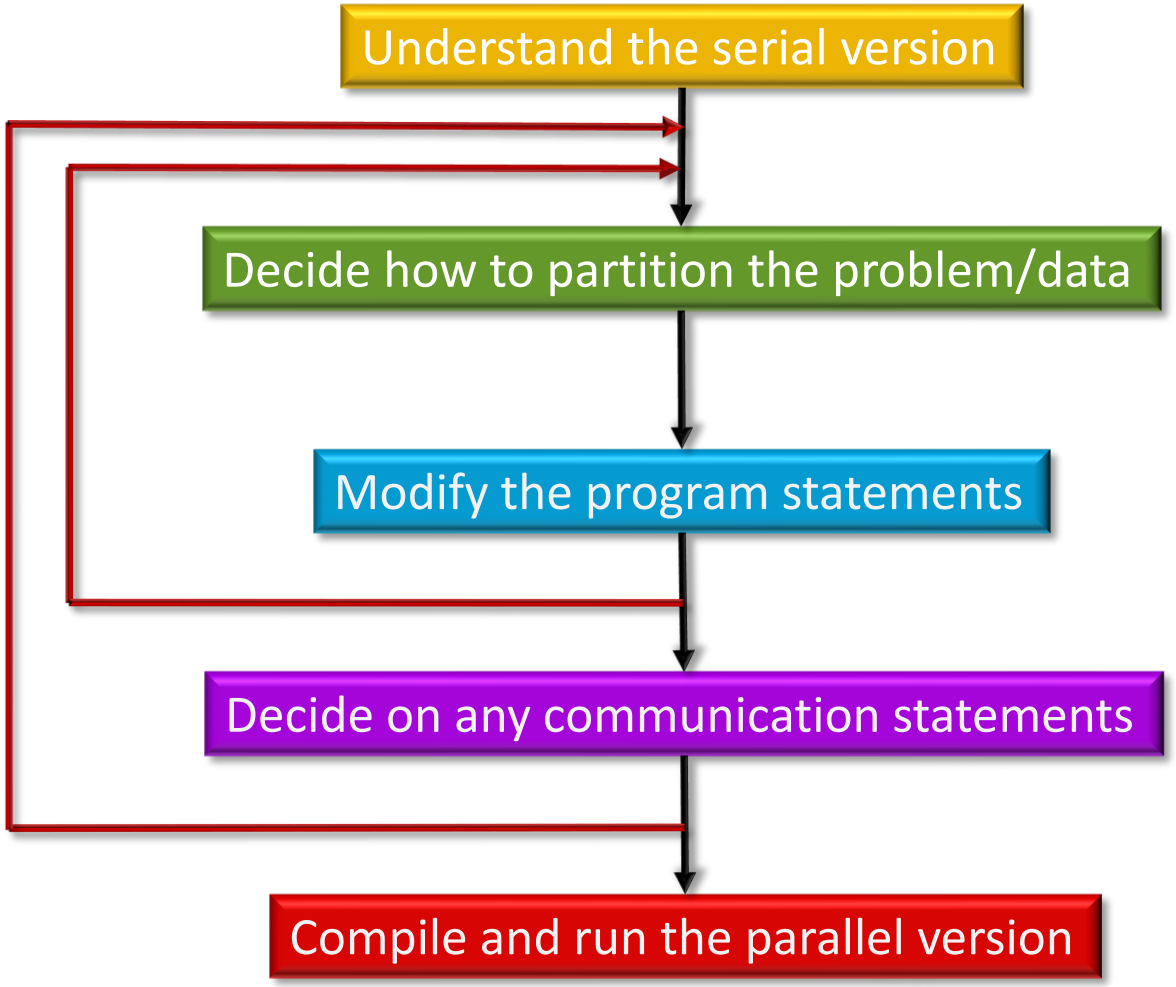
If you start with an existing serial code and have time or budget constraints, then automatic parallelization may be the answer ..but there are dangers...

- wrong results may be produced (opt level, parallel sum)
- performance may actually degrade (overhead may outweigh performance gain)
- significantly less flexible than manual parallelization
- limited to a subset (mostly loops) of code
- may actually not parallelize code if the analysis suggests there are inhibitors or the code is too complex

Limited success of automatic approach

- manual parallelisation a better strategy

How to Parallelise a code?



Understand The Serial Version

Understand the problem that you wish to solve in parallel

- we will assume that you are starting with a serial program
  - not just to make the task harder!

Determine whether or not the problem is one that can actually be parallelized!

Example of Parallelizable Problem:

Change the contrast of a bitmap image by calculating the new colour shade of each pixel in a bitmap image and also determine the minimum and maximum shade

This problem can be solved in parallel. Each of the pixel shades are independently determinable. The calculation of the minimum and maximum shades can also be obtained in parallel.

Understand The Serial Version

Example of a non-Parallelizable Problem:

Calculation of the Fibonacci series (1,1,2,3,5,8,13,21,...) by use of the formula:

$$F(k + 2) = F(k + 1) + F(k)$$

This is a non-parallelizable problem because the calculation of the Fibonacci sequence as shown requires dependent calculations rather than independent ones. The calculation of the k + 2 value uses those of both k + 1 and k. These three terms cannot be calculated independently and therefore, not in parallel

Understand The Serial Version

Identify the program's hotspots:

Know where most of the real work is being done. The majority of scientific and technical programs usually accomplish most of their work in a few places

Simple timing of code sections or a profiler such as **gprof** or a performance analysis tool can help

Focus on parallelizing the hotspots and **ignore those sections of the program that account for little CPU usage**

- caveat - can be significant depending on performance goal

Understand The Serial Version

Identify bottlenecks in the program

Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred?

- for example, I/O is usually something that slows a program down
  - is it diagnostic?
  - can it be deferred to a later point of execution?

It may be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas

Identify inhibitors to parallelism

- one common class of inhibitor is ***data dependence***, as demonstrated by the Fibonacci sequence above
- more advanced topic - lecture on its own

Investigate other algorithms if possible

- this may be the single most important consideration when designing a parallel application.

Decide How To Partition The Problem/Data

Break the problem into discrete pieces or chunks of work that can be distributed to multiple tasks. This is known as **decomposition** or **partitioning**

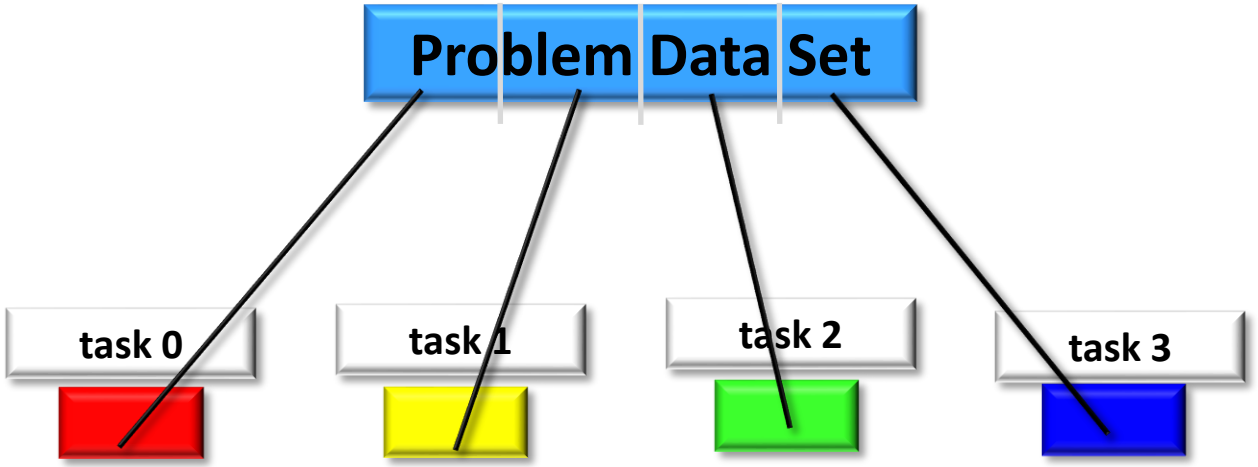
There are two basic ways to partition computational work among parallel tasks:

- domain decomposition
- functional decomposition

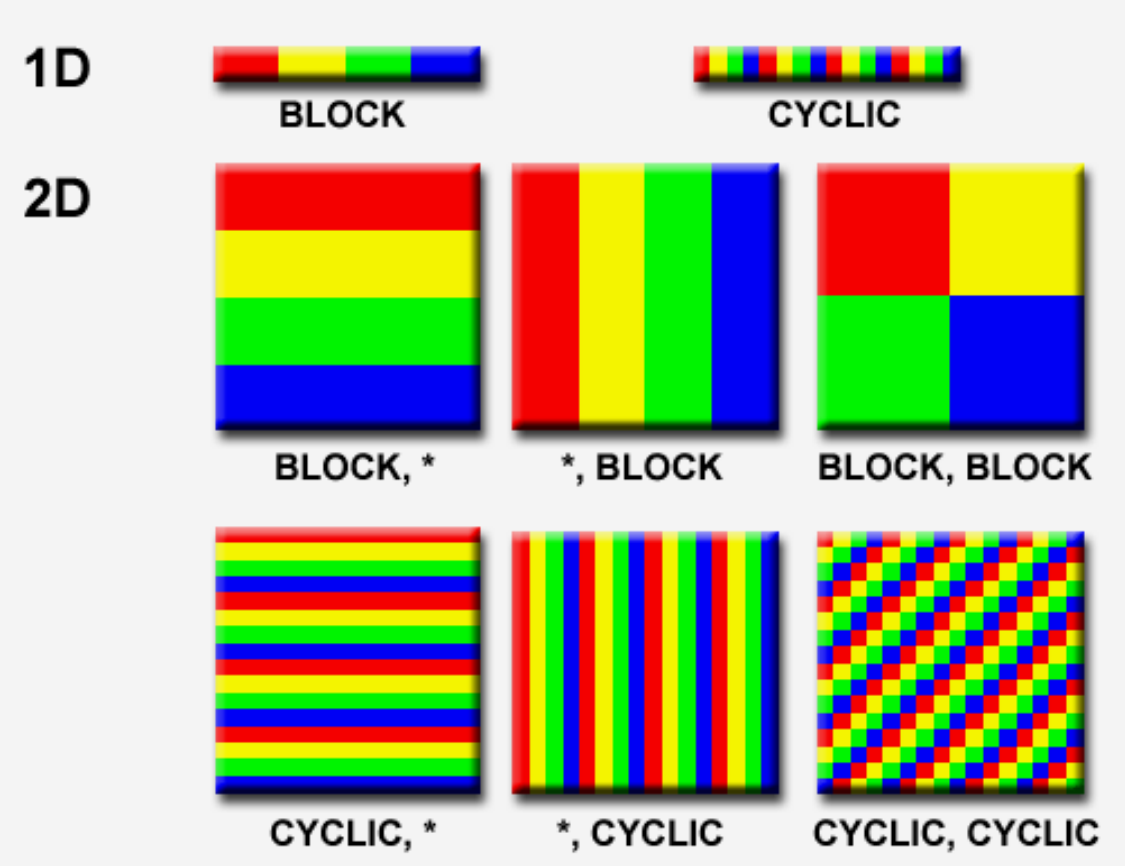
Decide How To Partition The Problem/Data

Domain decomposition

- The data associated with a problem is **decomposed**
- Each parallel task then works on a portion of the data

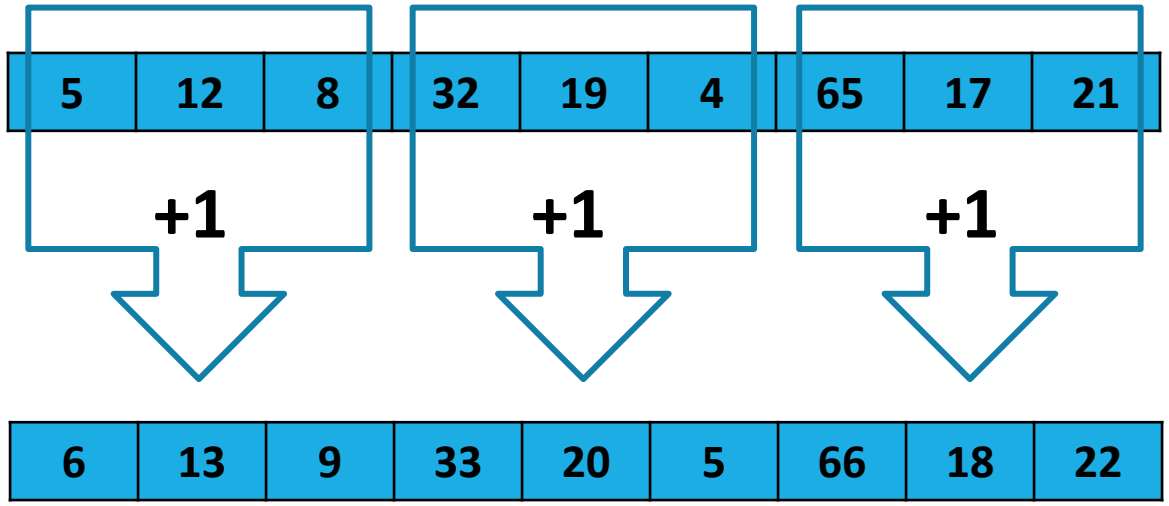


Domain Decomposition Example (1)



Domain Decomposition Example (2)

- Add 1 to each element of an array of integers
  - Partition the data among the parallel tasks
  - Each task works on a portion of the data

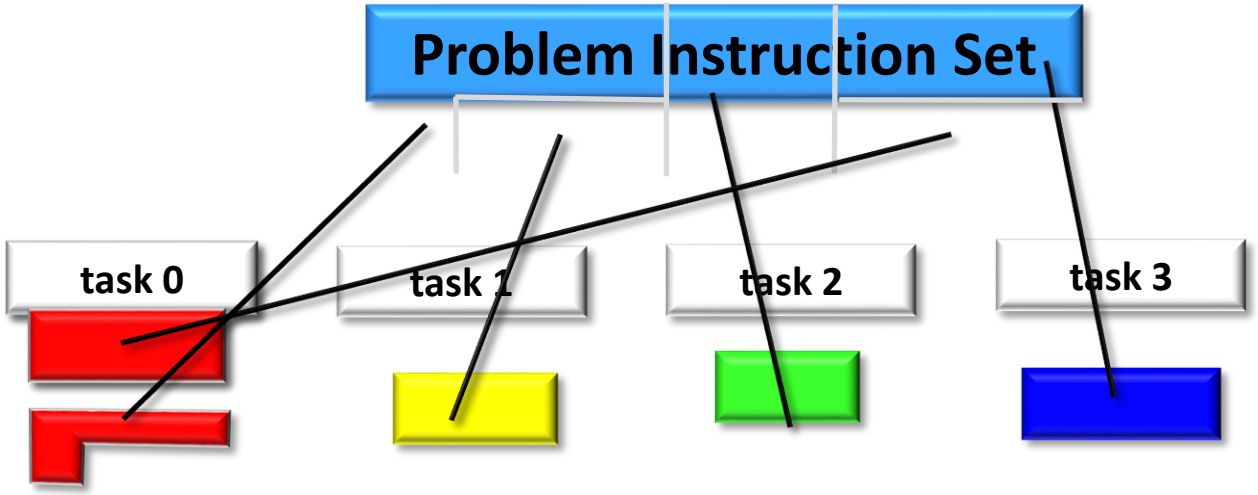


Functional Decomposition

Problem is decomposed according to the work (function) that must be done

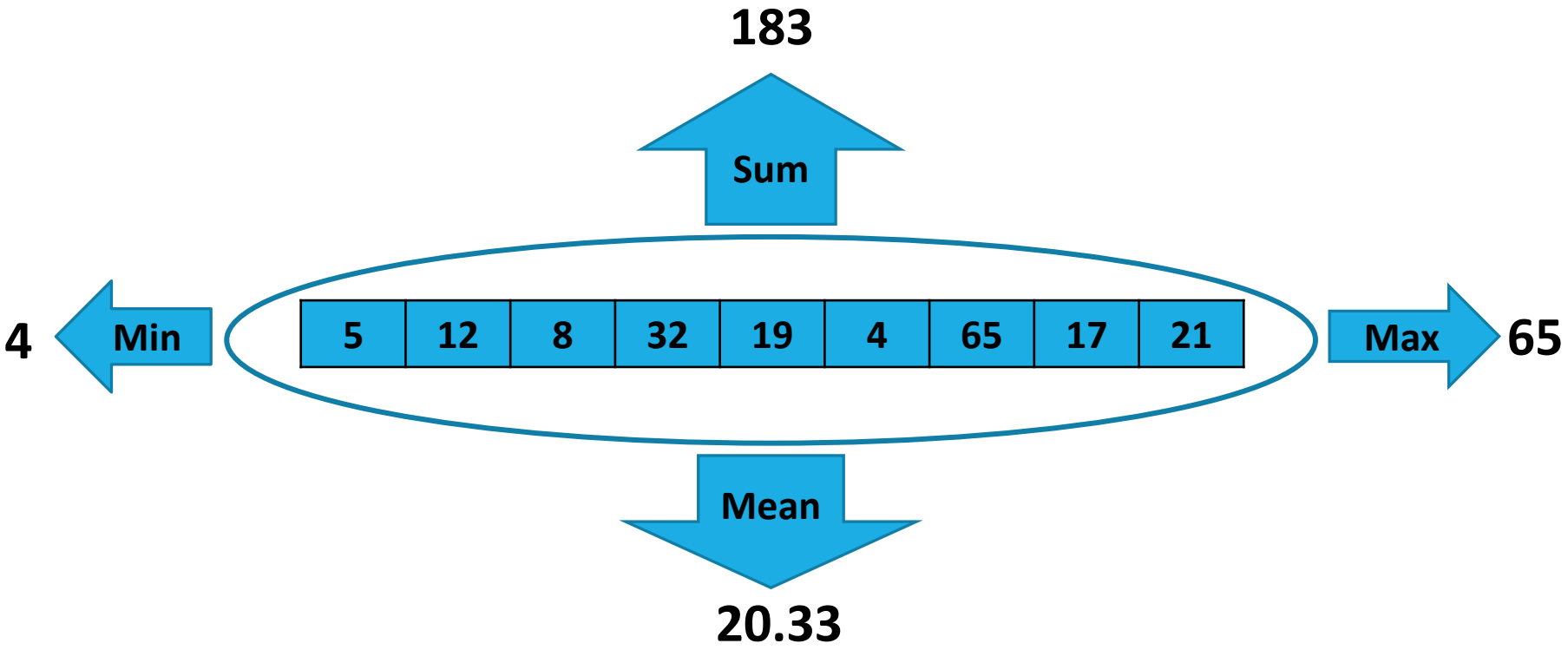
- each task then performs a portion of the overall work

Lends itself to problems that may be split into different tasks



Functional Decomposition Example (1)

- Perform different functions on the same data, in parallel
  - Sum, Max, Min, Mean

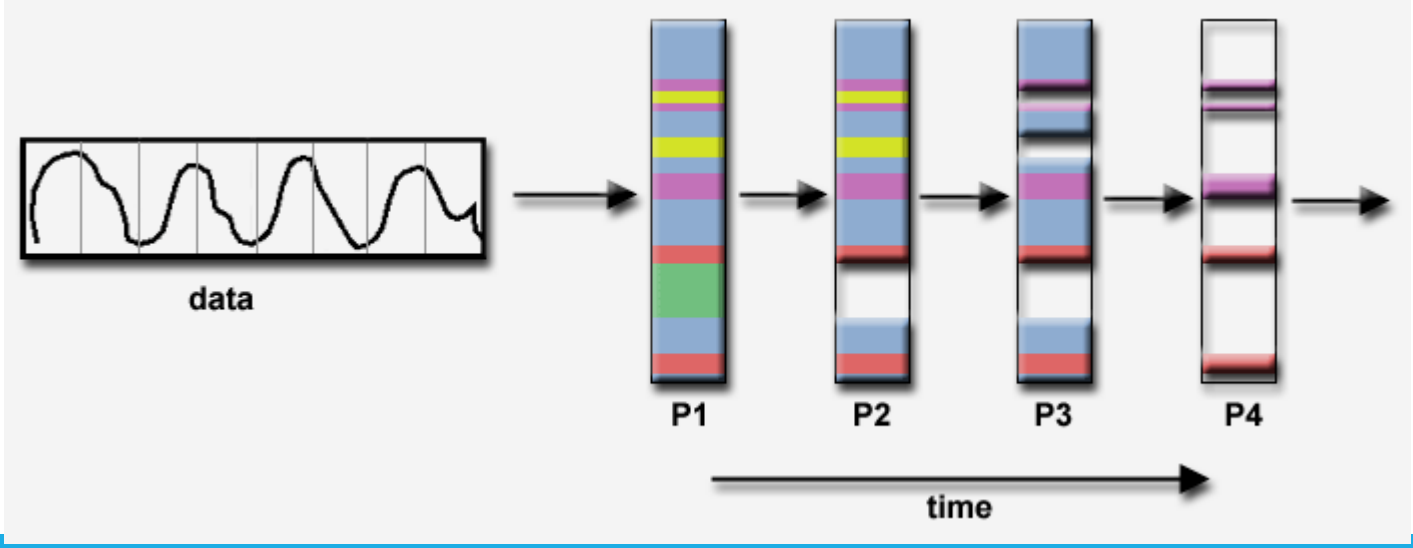


Functional Decomposition Example (2)

An audio signal data set is passed through four distinct computational filters (processes)

The first segment of data must pass through the first filter before progressing to the second. When it does, the second segment of data passes through the first filter. By the time the fourth segment of data is in the first filter, all four tasks are busy

- pipeline processing



Modify The Program Statements

- Largely dictated by decisions on the partitioning of data
- Statements could read like ”...if I own the data being assigned, then I execute these statements...”

```
 if (myprocnum.eq.0)  {
   printf(...

if( i.ge.low && i.le.high) {
    a[i] = ...

if(master) {
    send(...
    }
else if (worker) {
```

- Statements of this kind are termed **execution control masks** or **masks** for short

Modify The Program Statements

- The most time consuming and error-prone of the stages in developing a parallel code
- Need to ensure all data is **assigned** by only the allowable tasks
  - may require judgemental decisions to be made
  - e.g. whether to replicate work or to communicate information (get access to data) from another process?
- Decisions made in the masking stage will have a knock on effect with the placement of necessary communication of data between tasks to ensure consistency across all tasks
  - does this still follow the original serial computations?

Communication Statements

The need for data communication is problem dependent

You DON'T need communications

Some types of problems can be decomposed and executed in parallel with virtually no need for tasks to share data

e.g. imagine an image processing operation where every pixel in a black and white image needs to have its colour reversed. The image data can easily be distributed to multiple tasks that then act **independently** of each other to do their portion of the work

These types of problems are often called **embarrassingly parallel** because they are so straight-forward

Little or no inter-task communication is required

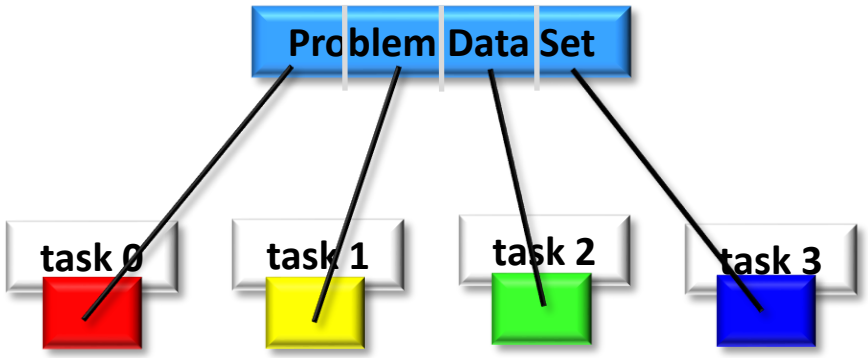
Communication Statements

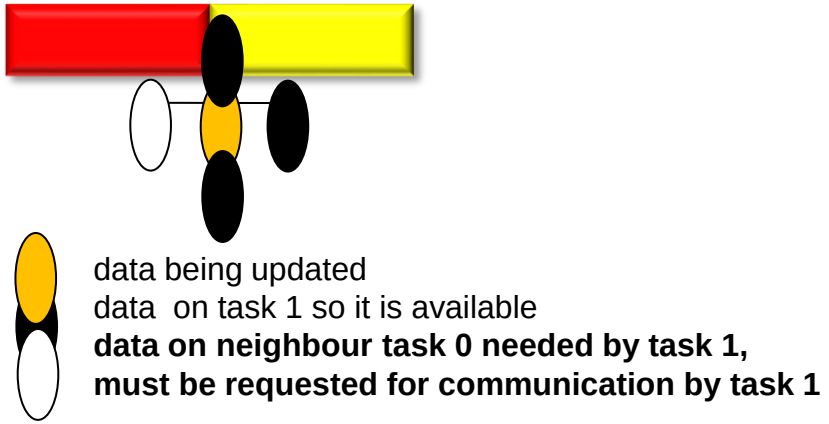
You DO need communications

Most real-world parallel applications are not so simple

They do require tasks to share data with each other

e.g. 3-D heat diffusion problem requires a task to know the temperatures calculated by the tasks that have neighbouring data. Changes to neighbouring data has a direct effect on that task's data





Communication Statements

Factors to consider in decision process

- Cost of communications
- Latency vs. Bandwidth
- Visibility of communications
- Synchronous vs. asynchronous communications
- Scope of communications
- Efficiency of communications

...

- Communication can have a significant effect on performance – this will be discussed in a later lecture

Communication Statements

Cost of communications

- Inter-task communication implies overhead
  - it’s NOT free!
- Machine cycles and resources that could be used for computation are instead used to package and transmit data
- Communications very often need some synchronization between tasks
  - can result in tasks spending time "waiting" instead of doing work
- Competing communication traffic can saturate the available network bandwidth
  - further aggravating performance problems

Communication Statements

Latency vs. Bandwidth

- **Latency** is the time it takes to send a minimal (0 byte) message from point A to point B
  - units in microseconds (µs) or nanoseconds (ns)
- **Bandwidth** is the amount of data that can be communicated per unit of time
  - units in Mb/sec or Gb/sec.
- Sending many small messages can cause **latency** to dominate communication overheads
- Often it is more efficient to package small messages into a larger message, this increases the effective communications **bandwidth** and also reduces the overall latency cost

Communication Statements

Visibility of communications

- when using **message passing** the communications are explicit and visible
  - under the control of the programmer
  - applies in MPI case
- When using **threads** the communications occur transparently to the programmer
  - the programmer is unlikely to know exactly how the inter-task communications are being accomplished
  - applies in OpenMP case

Communication Statements

Synchronous vs. asynchronous communications

- **Synchronous** communications are often referred to as **blocking** communications since other work must wait until the communications have completed
- Synchronous communications require some type of "handshaking" between tasks that are sharing data
- Can be explicitly structured in code by the programmer, or may happen at a lower level unknown to the programmer

Communication Statements

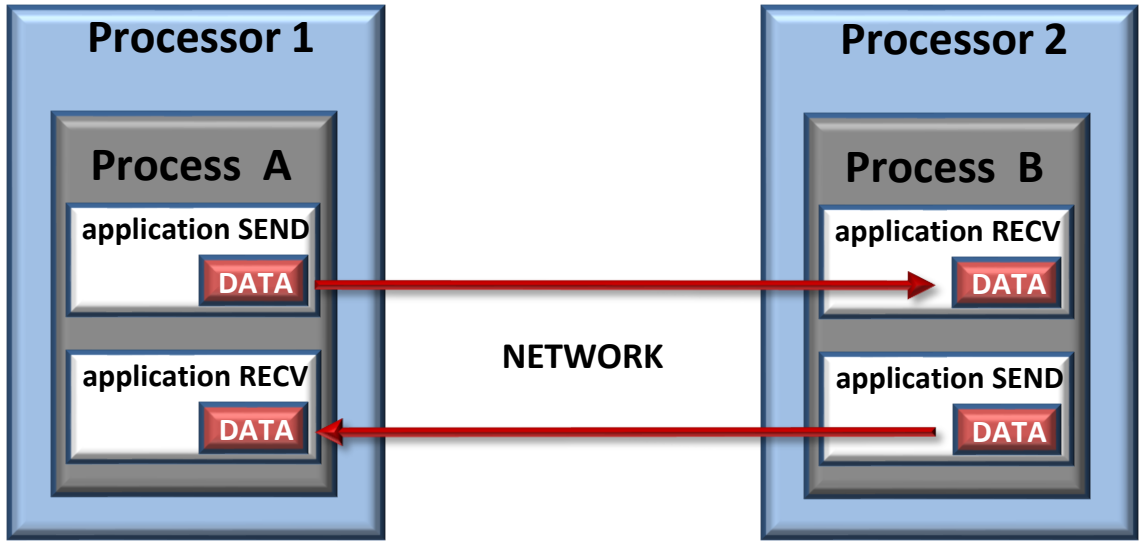
Synchronous vs. asynchronous communications

- **Asynchronous** communications are often referred to as **non-blocking** communications
  - other work can be done while the communications are taking place
- Asynchronous communications allow tasks to transfer data independently from one another
  - For example, task 1 can prepare and send a message to task 2, and then immediately begin doing other work. When task 2 actually receives the data doesn't matter to task 1
- Interleaving computation with communication is the single greatest benefit for using asynchronous communications

Communication Statements

Scope of communications

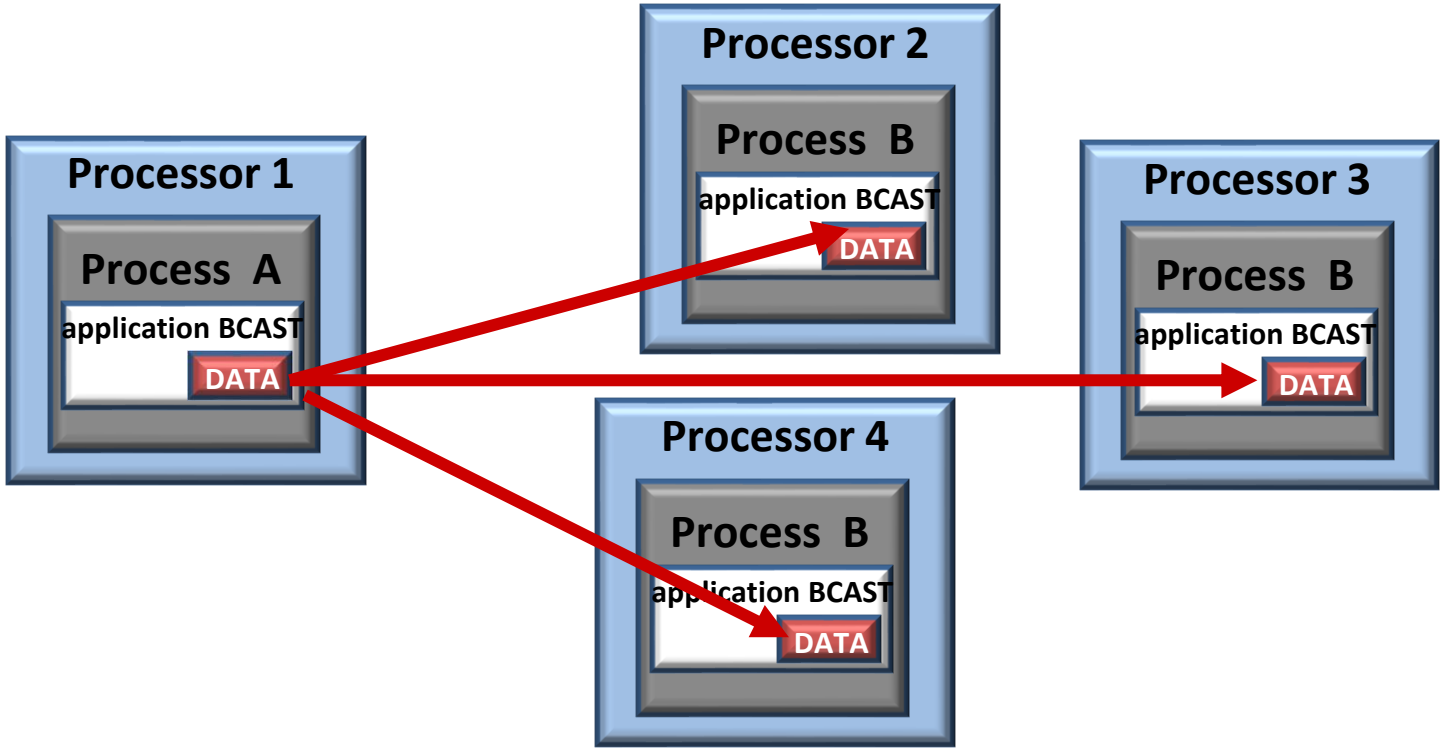
- Knowing which tasks must communicate with each other is critical during the design stage of a parallel code
- **Point-to-point** communication involves two tasks with one task acting as the **sender/producer** of data, and the other acting as the **receiver/consumer**



Communication Statements

Scope of communications

- **Collective** communication involves data sharing between more than two tasks
  - often specified as being members of a common group or collective



Communication Statements

Efficiency of communications

Many factors can affect ***communications performance***

- which implementation for a given model should be used?
  - for example, using MPI message passing one implementation may be faster on a given hardware platform than another
- what type of communication operations should be used?
  - as mentioned previously, asynchronous communication operations may improve overall program performance
    - but are they more difficult to implement?
- some platforms may offer more than one network for communications
  - which one is best?

Compile and Run the Parallel Version

Compiling

- This will usually be done as a command line instruction
  - be prepared for syntax and other errors
- When compilation is completed with no errors and any appropriate libraries have been linked an executable should be produced

Running

- Usually a command line instruction
  - will probably require some environment variables to also be set
- If it does not work as you expect (highly likely) then you will need to debug
  - the fun part ☹!