Parallel Programming

Overview and Concepts

Outline

- Decomposition
  - Geometric decomposition
  - Task farm
  - Pipeline
  - Loop parallelism
- General parallelisation considerations
- Parallel code performance metrics and evaluation
- Parallel scaling models

Why use parallel programming?

It is harder than serial so why bother?

Why?

- Parallel programming is more difficult than its sequential counterpart

- However we are reaching limitations in uniprocessor design
  - Physical limitations to size and speed of a single chip
  - Developing new processor technology is very expensive
  - Some fundamental limits such as speed of light and size of atoms

- Parallelism is not a silver bullet
  - There are many additional considerations
  - Careful thought is required to take advantage of parallel machines

Performance

- A key aim is to solve problems faster
  - To improve the time to solution
  - Enable new scientific problems to be solved
- To exploit parallel computers, we need to split the program up between different processors
- Ideally, would like program to run P times faster on P processors
  - Not all parts of program can be successfully split up
  - Splitting the program up may introduce additional overheads such as communication

Parallel tasks

- How we split a problem up in parallel is critical
    1. Limit communication (especially the number of messages)
    2. Balance the load so all processors are equally busy

- Tightly coupled problems require lots of interaction between their parallel tasks
- Embarrassingly parallel problems require very little (or no) interaction between their parallel tasks
    - E.g. the image sharpening exercise

- In reality most problems sit somewhere between two extremes

Decomposition

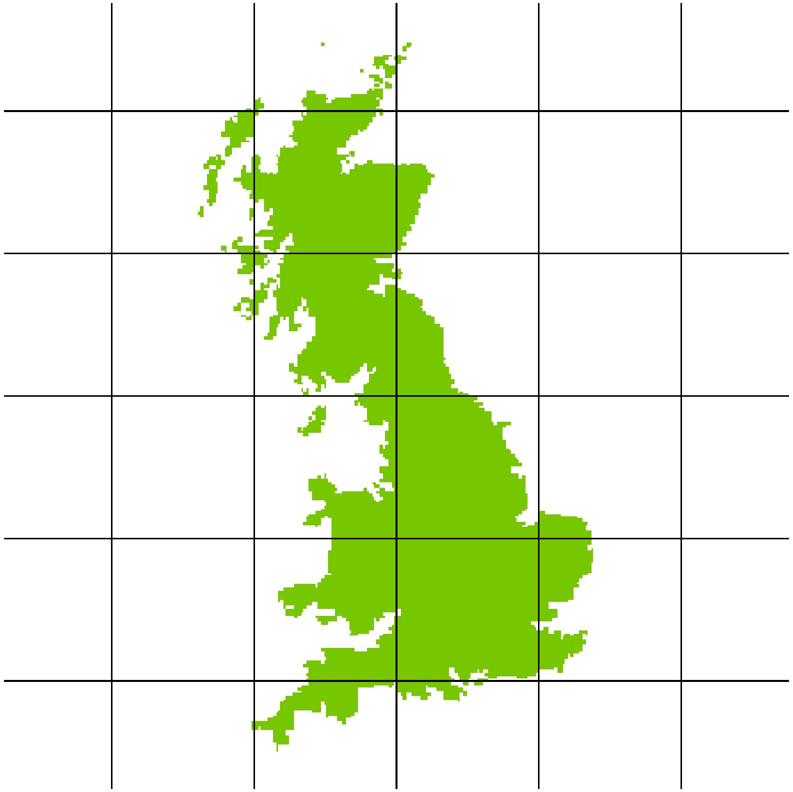
How do we split problems up to solve efficiently in parallel?

Decomposition

- One of the most challenging, but also most important, decisions is how to split the problem up
- How you do this depends upon a number of factors
  - The nature of the problem
  - The amount of communication required
  - Support from implementation technologies
- We are going to look at some frequently used decompositions

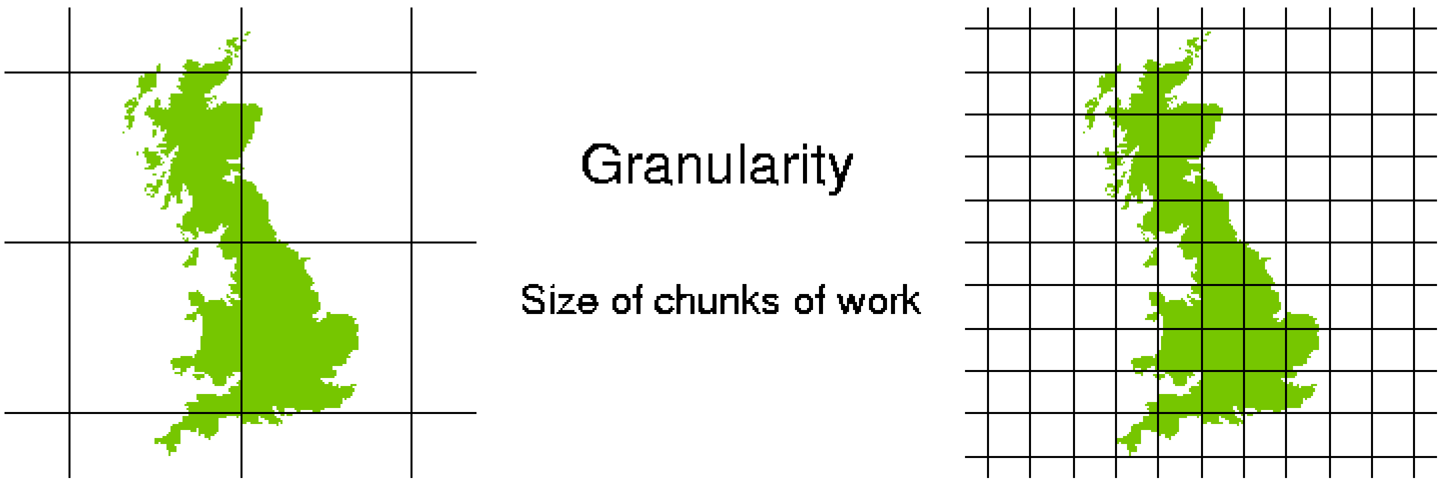
Geometric decomposition

- Take advantage of the geometric properties of a problem



Geometric decomposition

- Splitting the problem up does have an associated cost
  - Namely communication between processors
  - Need to carefully consider granularity
  - Aim to minimise communication and maximise computation



too large: little parallelism

too small: communications rule

Halo swapping

- Swap data in bulk at pre-defined intervals
- Often only need information on the boundaries
- Many small messages result in far greater overhead

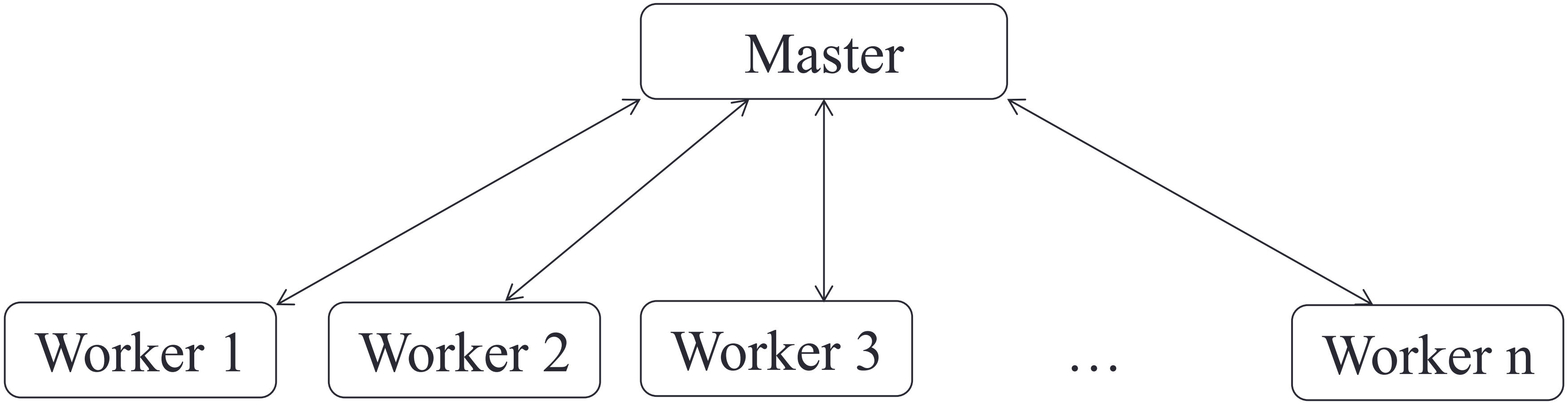
Load imbalance

- Execution time determined by slowest processor
  - each processor should have (roughly) the same amount of work, i.e. they should be load balanced

- Assign multiple partitions per processor
  - Additional techniques such as work stealing available

Task farm (master worker)

- Split the problem up into distinct, independent, tasks



- Master process sends task to a worker
- Worker process sends results back to the master
- The number of tasks is often much greater than the number of workers and tasks get allocated to idle workers

Task farm considerations

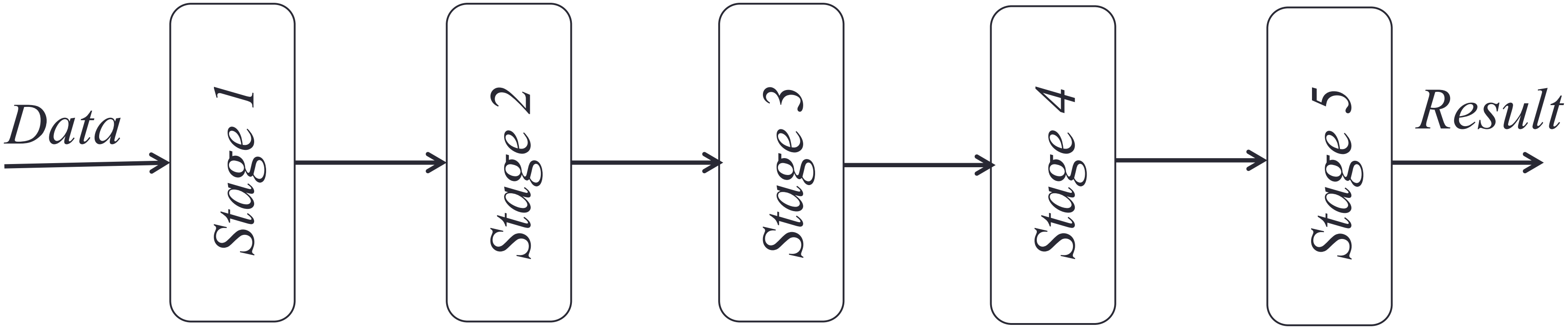
- Communication is between the master and the workers
  - Communication between the workers can complicate things

- The master process can become a bottleneck
  - Workers are idle waiting for the master to send them a task or acknowledge receipt of results
  - Potential solution: implement work stealing

- Resilience – what happens if a worker stops responding?
  - Master could maintain a list of tasks and redistribute that work's work

Pipeline

- A problem involves operating on many pieces of data in turn. The overall calculation can be viewed as data flowing through a sequence of stages and being operated on at each stage.



- Each stage runs on a processor, each processor communicates with the processor holding the next stage
- One way flow of data

Examples of pipeline

- CPU architectures
  - Fetch, decode, execute, write back
  - Intel Pentium 4 had a 20 stage pipeline
- Unix shell
  - i.e. cat datafile | grep “energy” | awk ‘{print $2, $3}’
- Graphics/GPU pipeline

- *A generalisation of pipeline (a workflow, or dataflow) is becoming more and more relevant to large, distributed scientific workflows*
- *Can combine the pipeline with other decompositions*

Loop parallelism

- Serial programs can often be dominated by computationally intensive loops.
- Can be applied incrementally, in small steps based upon a working code
  - This makes the decomposition very useful
  - Often large restructuring of the code is not required
- Tends to work best with small scale parallelism
  - Not suited to all architectures
  - Not suited to all loops
- If the runtime is not dominated by loops, or some loops can not be parallelised then these factors can dominate (Amdahl's law.)

Example of loop parallelism:

```
int main(int argc, char *argv[]) {
    const int N = 100000;
    int i, a[N];

    #pragma omp parallel for
    for (i = 0; i < N; i++)
        a[i] = 2 * i;

    return 0;
}
```

- If we ignore all parallelisation directives then should just run in serial
- Technologies have lots of additional support for tuning this

Summary

- There are many considerations when parallelising code

- A variety of patterns exist that can provide well known approaches to parallelising a serial problem
  - You will see examples of some of these during the practical sessions