Parallel Programming Patterns

Overview and Concepts

Reusing this material

Outline

- Why parallel programming?
- Parallel decomposition patterns
  - Geometric decomposition
  - Task farm / worker queue
  - Pipeline
  - Loop parallelism

Why parallel programming?

- More difficult than “normal” (serial) programming, so why bother?
- We are reaching limitations in speed of individual processors
  - Limitations to size and speed of a single chip (heat!)
  - Developing new processor technology is very expensive
  - Ultimately enounter fundamental limits: speed of light and size of atoms
  - Processor designs moving away from few fast cores to many slower ones
- Increasingly need parallel applications to continue advancing science
- Parallelism is not a silver bullet
  - There are many additional considerations
  - Careful thought is required to take advantage of parallel machines

"But I'm not a programmer"

- This course will *not* teach you how to program in parallel
- It *will* provide you with an overview of some of the common ways this is done
- There are two aspects to this:
  1. how computational work can be split up and divided amongst processors/cores in an abstract sense (the topic of this lecture)
  2. how this can actually be implemented in hardware and software (later lectures)

"But I'm not a programmer"

- Understanding how programs run in parallel should help you:
  - make better-informed choices what software to use for your research
  - understand what problems can emerge (parallel performance or errors)
  - make better use of high-performance computers (and even your laptop!)
  - get your research done more quickly

- You may decide to learn to program in parallel!
  - you will already have an overview of many key concepts

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, need to split the program up between different processors (cores)
  - distinguish between processors and cores when it matters, otherwise use interchangeably - more in lecture on hardware.
- 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 into tasks to run in parallel is critical
    1. Ideally limit interaction (information exchange) between tasks as this takes time and may require processors to wait for each other
    2. Want to balance the workload so all processors are equally busy - if they are all equally powerful this gives the shortest time to solution

- “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. sequence alignment queries for multiple independent sequences

- In reality most problems sit somewhere between the two extremes

Parallel Decomposition

How do we split problems up to solve them 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 required amount and frequency of interaction (information exchange) between tasks
  - Support from implementation technologies

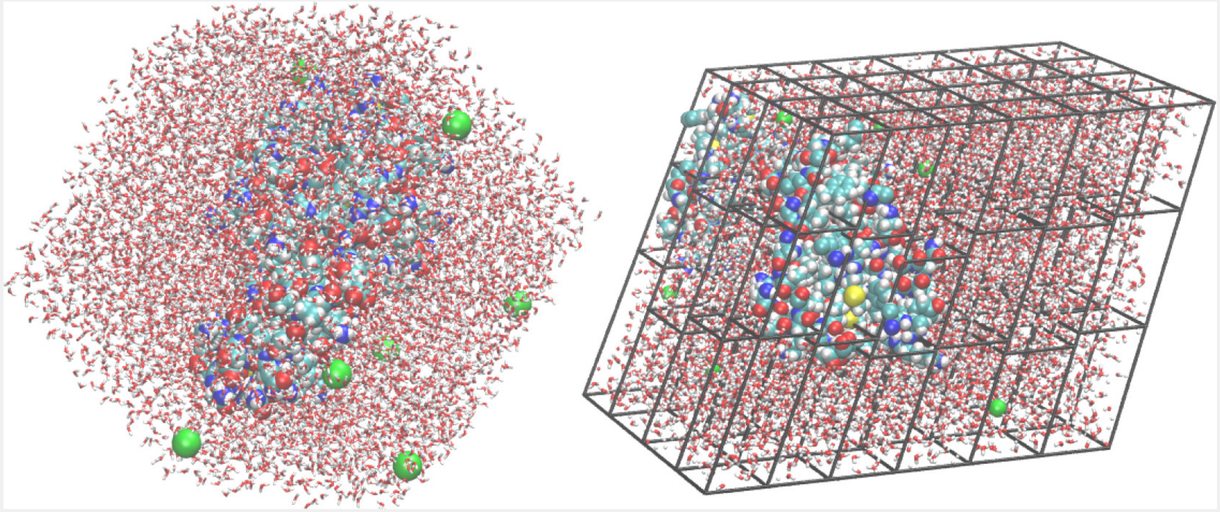
- We are going to look at some frequently used parallel decomposition patterns

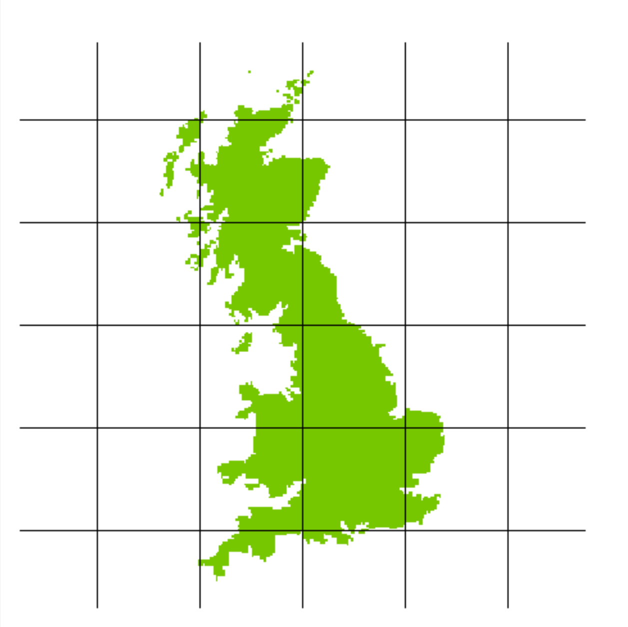
1. Geometric Decomposition

Based on a geometric division of the spatial domain of a problem

Biomolecular simulation

Weather Simulation





From: **GROMACS: High performance molecular simulations through multi-level parallelism from laptops to supercomputers.** SoftwareX, Volumes 1–2, 2015, 19–25. http://dx.doi.org/10.1016/j.softx.2015.06.001 (reuse permitted under CC BY 4.0)

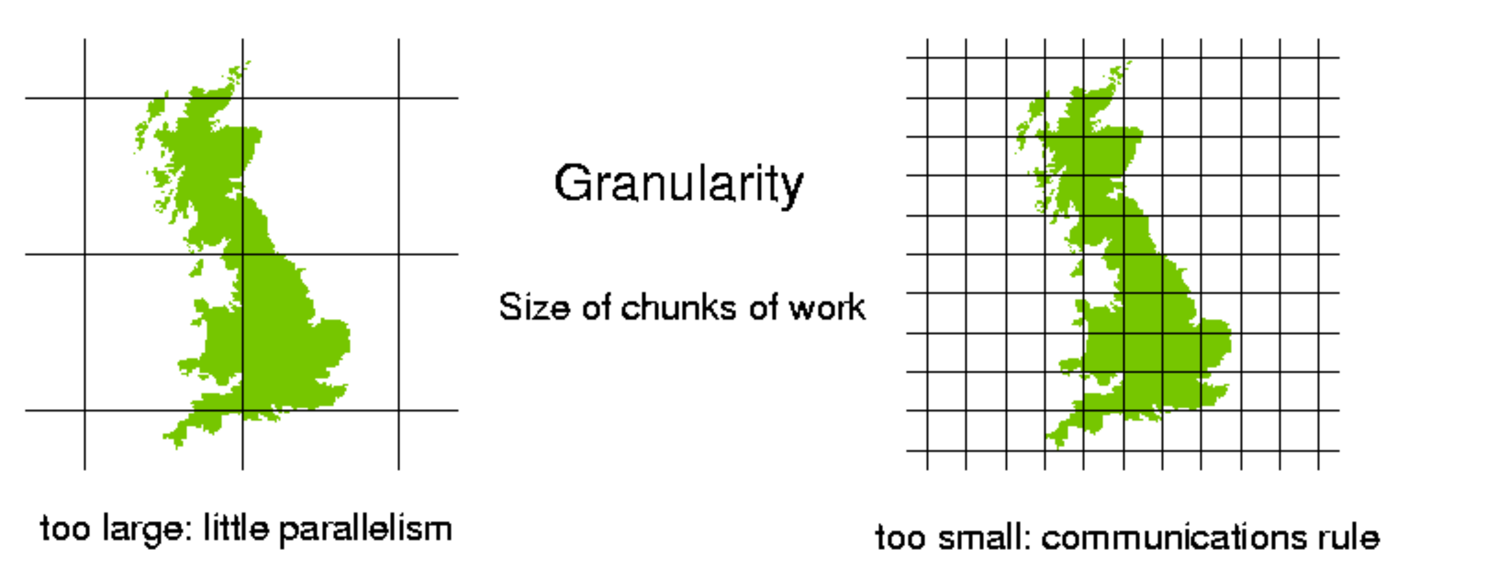
1. Geometric Decomposition

- Spatial domain divided geometrically into cells
- Simplest case:
  - all cells same size and shape
  - one cell per processor
- More adaptable:
  - variably-sized and shaped cells
  - more cells than processors
- Information exchange required between cells:
  - temperature, pressure, humidity etc. for weather simulation
  - atomic/molecular charges to compute forces in biomolecular simulation

1. Geometric Decomposition

Molecular Dynamics Practical

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



1. Geometric Decomposition

- Swap data between cells
- Often only need information on cell boundaries
- Many small messages result in far greater overhead
    - instead exchange all boundary values periodically by swapping cell "halos".

Load Imbalance

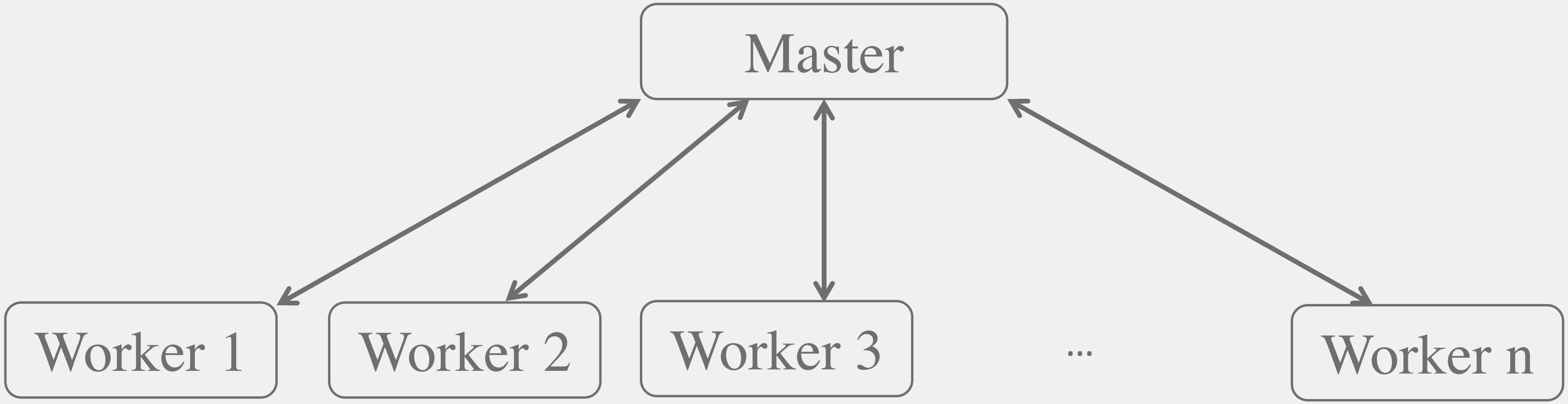
- Overall execution time worse if some processors take longer than the rest
- Each processor should have (roughly) the same amount of work, i.e. they should be load balanced
- For many problems it is unlikely that all cells require same amount of computation
  - Expect “hotspots” – regions where more compute is needed, e.g.:
    - localised high-low pressure fronts (weather simulation)
    - cells containing complex protein segments (biomolecular simulation)

Load Imbalance

- Can measure degree of load imbalance
  - see lecture on measuring parallel performance
- Techniques exist to deal with load imbalance:
  - Assign multiple cells to each processor
  - Use variably-sized cells in the first place to compensate for hotspots
  - Allow processors to dynamically “steal” work from others
- Load balancing can be done
  - once at the start of program execution (static)
  - throughout execution (dynamic)

2. Task farm (master / worker)

- Split a problem up into distinct, independent, tasks



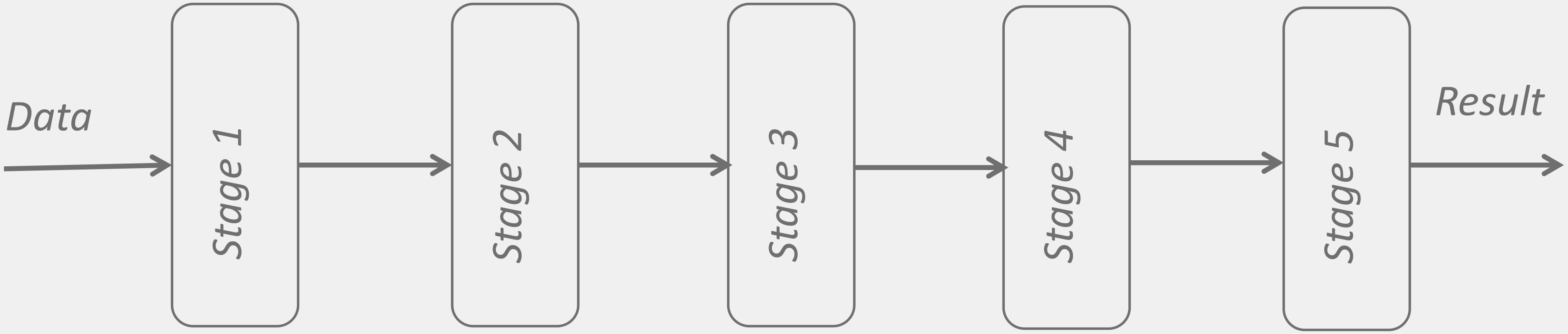
- Master process sends task to a worker
- Worker process sends results back to the master
- The number of tasks is often much greater than the number of workers and tasks get allocated to idle workers dynamically

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 worker’s work

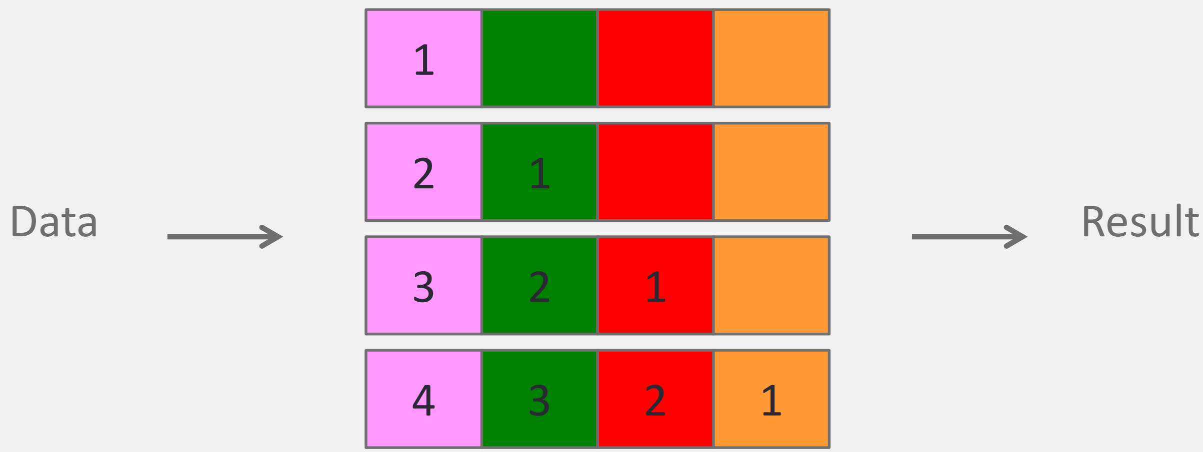
3. Pipelines

- Some problems involve 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

Example: pipeline with 4 processors



- Each processor (one per colour) is responsible for a different task or stage of the pipeline
- Each processor acts on data (numbered) as they move through the pipeline

Examples of pipelines

- CPU architectures
  - Fetch, decode, execute, write back
  - Intel Pentium 4 had a 20 stage pipeline
- Unix shell
  - i.e. cat datafile I grep “energy” I 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*

4. Loop Parallelism

- Serial scientific applications are often dominated by computationally intensive loops
- Some of these can be parallelised directly
  - e.g.10 cores simultaneously perform 1000 iterations each instead of 1 core performing 10 000 iterations
- Simple techniques exist to do this incrementally, i.e. in small steps whilst maintaining a working code
  - This makes the decomposition very easy to implement
  - 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

Summary

- A variety of common decomposition patterns exist that provide well-known approaches to parallelising a serial problem
  - You can see examples of some of these during the practical sessions
- There are many considerations when parallelising code:
  - Granularity of the decomposition
  - Tradeoff of communication and computation
  - Load imbalance
- Parallel applications implement clever variations on these decomposition schemes to optimise parallel performance
  - Knowing some of this will help you understand what is going on