Basic Principles of Parallelism

Gavin Pringle
gavin@epcc.ed.ac.uk

Outline

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

Performance

- A key aim is to solve problems faster
  - To improve the time to solution
  - Enable new a new scientific problems to be solved

- To exploit parallel computers, we need to split the program up between different cores

- Ideally, would like program to run N times faster on N cores
  - 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 cores are equally busy

- “Tightly coupled” problems require lots of interaction between their parallel tasks

- “Pleasingly” parallel problems require very little (or no) interaction between their parallel tasks
  - E.g. ensemble simulations

- In reality most problems sit somewhere between two extremes

Decomposition

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

Decomposition

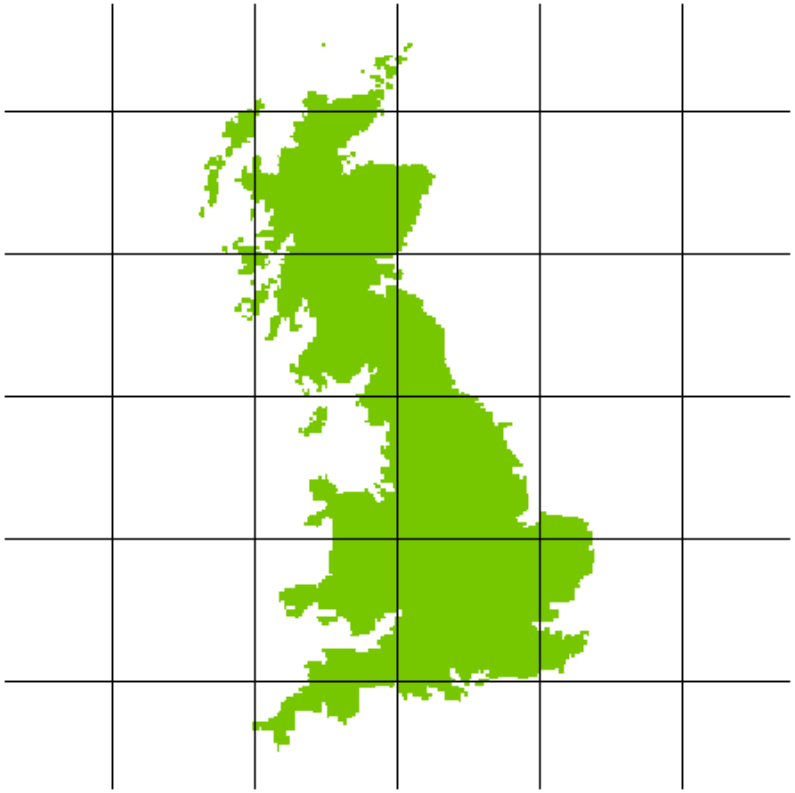
- One of the most challenging, but also most important, decisions is how to split the problem up

- 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

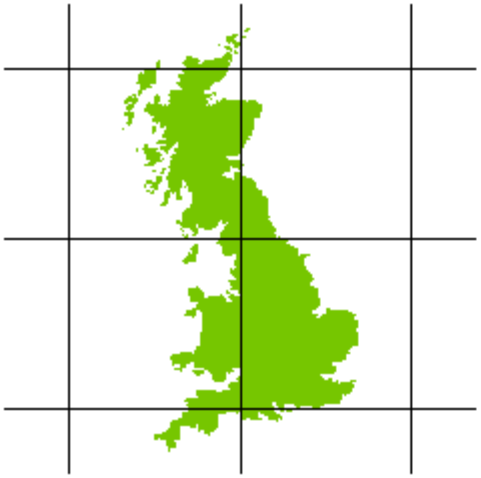
Geometric decomposition

- Take advantage of the geometric properties of a problem



Geometric decomposition

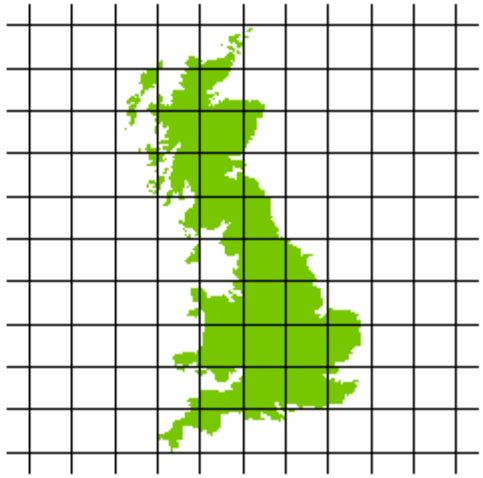
- Splitting the problem up does have an associated cost
  - Namely communication between cores
  - Need to carefully consider granularity
  - Aim to minimise communication and maximise computation



too large: little parallelism

Granularity

Size of chunks of work



too small: communications rule

epcc

THE UNIVERSITY OF EDINBURGH

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

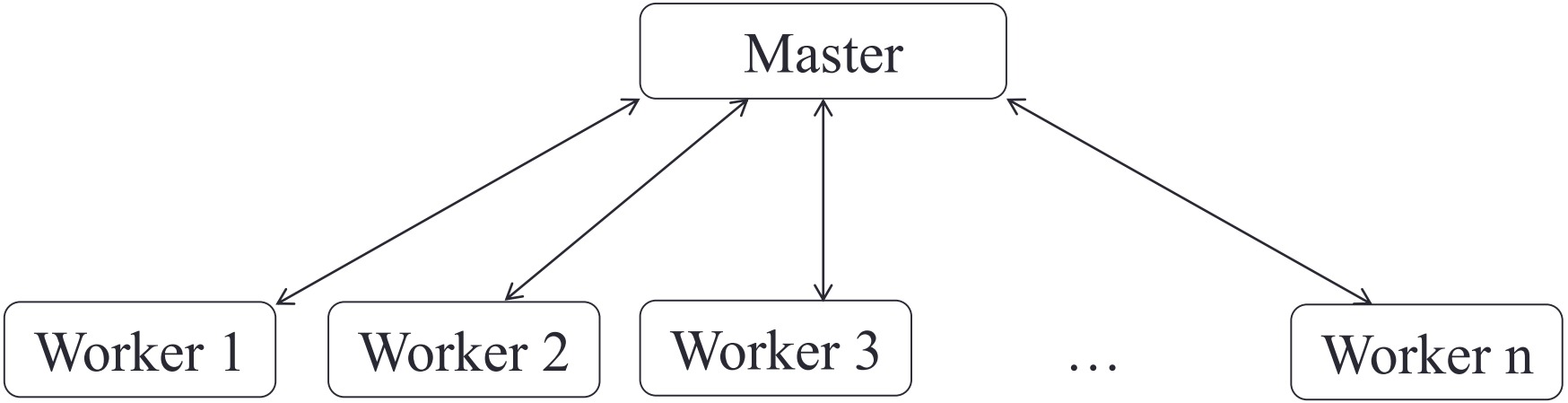
Load imbalance

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

- Address by multiple partitions per core
  - Additional techniques such as work stealing available

Task farm (master worker)

- Split the problem up into distinct, independent, tasks



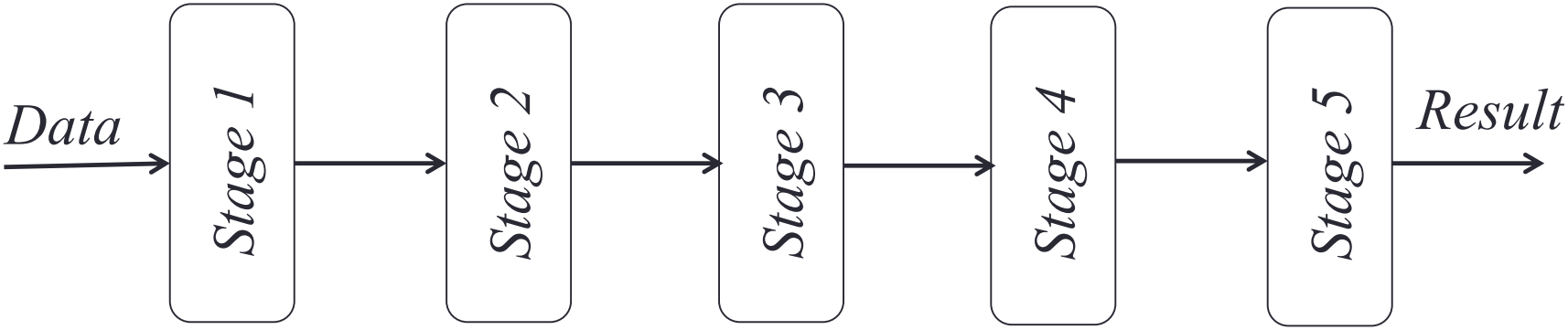
- Master process sends task to a worker
- Worker process sends results back to the master
- The number of tasks is often much greater than the number of workers and tasks get allocated to idle workers
- If known in advance, order jobs and send largest jobs first

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 core, each core communicates with the core 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

Performance metrics

How is my parallel code performing and scaling?

Why care about parallel performance?

- Why do we run applications in parallel?
  - so we can get solutions more quickly
  - so we can solve larger, more complex problems
- If we use 10x as many cores, ideally
  - we'll get our solution 10x faster
  - we can solve a problem that is 10x bigger or more complex
  - unfortunately this is not always the case…
- Measuring parallel performance can help us understand
  - whether an application is making efficient use of many cores
  - what factors affect this
  - how best to use the application and the available HPC resources

Performance Metrics

- How do we quantify performance when running in parallel?
- Consider **execution time** *T(N,C)* measured whilst running on *C* cores with problem size/complexity *N*
- **Speedup***:*
  - typically S(N,C)=T(N,1)/T(N,C)
- **Parallel efficiency**:
  - Typically E(N,C)=S(N,C)/C T(N,1)/(C*T(N,C))

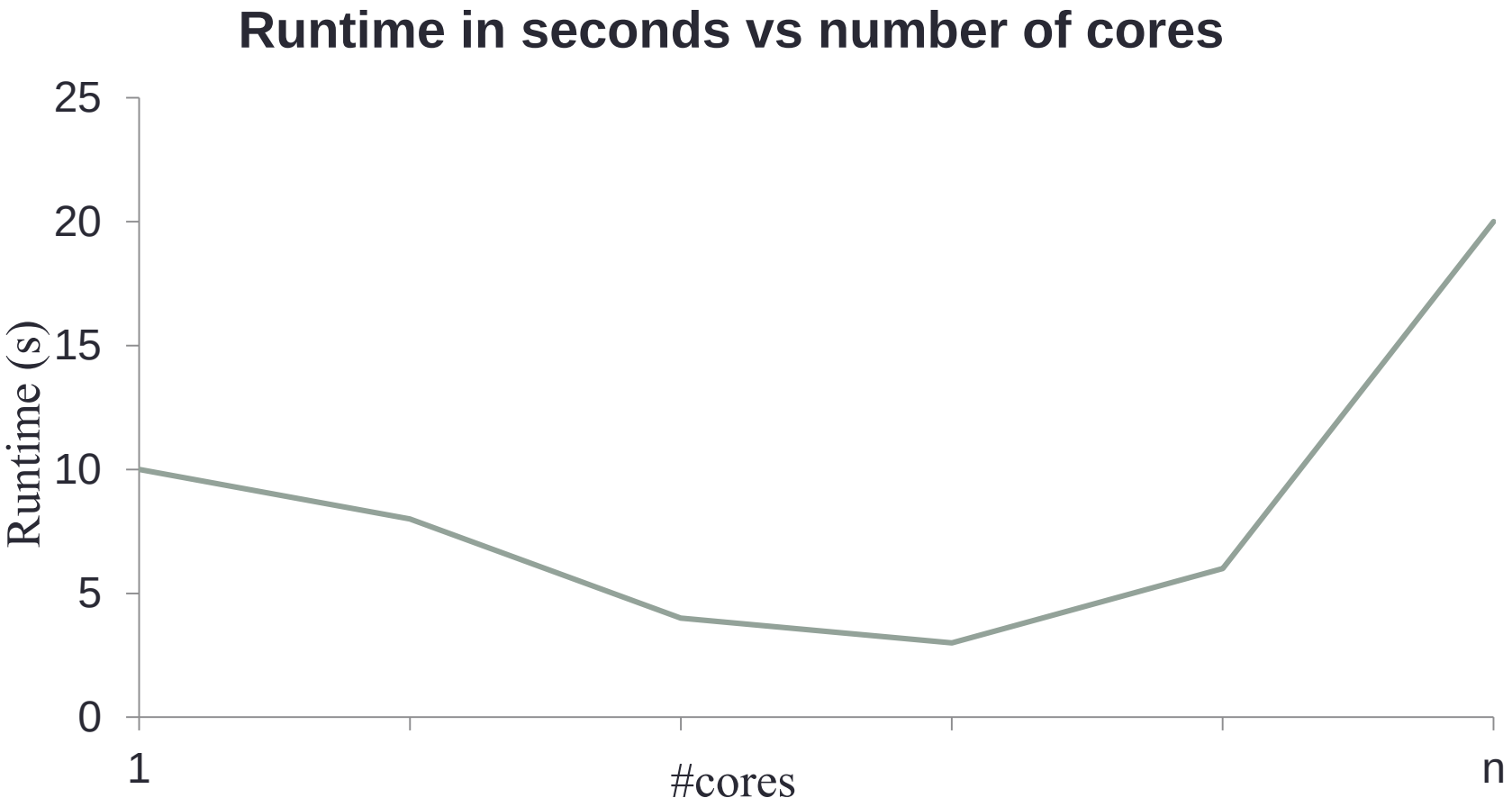
Parallel Scaling

- ***Scaling*** describes how the runtime of a parallel application changes as the number of cores is increased

- **Strong Scaling** (increasing *C,* **constant** *N*)
  - problem size/complexity stays the same as the number of cores increases, decreasing the work per core

- **Weak Scaling** (increasing C, increasing N)
  - problem size/complexity increases as the number of cores increases, keeping the amount of work per core the same.

Strong scaling



Load Imbalance

- These laws all assumed all cores are equally busy
  - what happens if some run out of work?
- Specific case
  - four people pack boxes with cans of soup: 1 minute per box

Person	Anna	Paul	David	Helen	Total
# boxes	6	1	3	2	12

  - takes 6 minutes as everyone is waiting for Anna to finish!
  - if we gave everyone same number of boxes, would take 3 minutes
- Scalability isn't everything
  - make the best use of the cores at hand before increasing the number of cores

Quantifying Load Imbalance

- Define **Load Imbalance Factor**

$$LIF = maximum\ load / average\ load$$

  - for perfectly balanced problems $LIF = 1.0$, as expected
    - in general, $LIF > 1.0$
  - LIF tells you how much faster your calculation could be with balanced load

- Box packing
  - LIF = 6/3 = 2
  - initial time = 6 minutes
  - best time = initial time / LIF = 6/2 = 3 minutes

Using HPC Systems

Example: how many cores should I employ for my code?

I have a simulation: how many cores should I employ?

- Let's say you have access to a machine with 500 nodes, with 32 cores per a node.
- Your code is a Python code parallelised using threads
  - Typically one thread per core is best.
- We can only use one node for our python code
  - Threads cannot communicate between nodes
- But, how many cores to use?
  - Typically one thread per core is best
- We need to benchmark!?

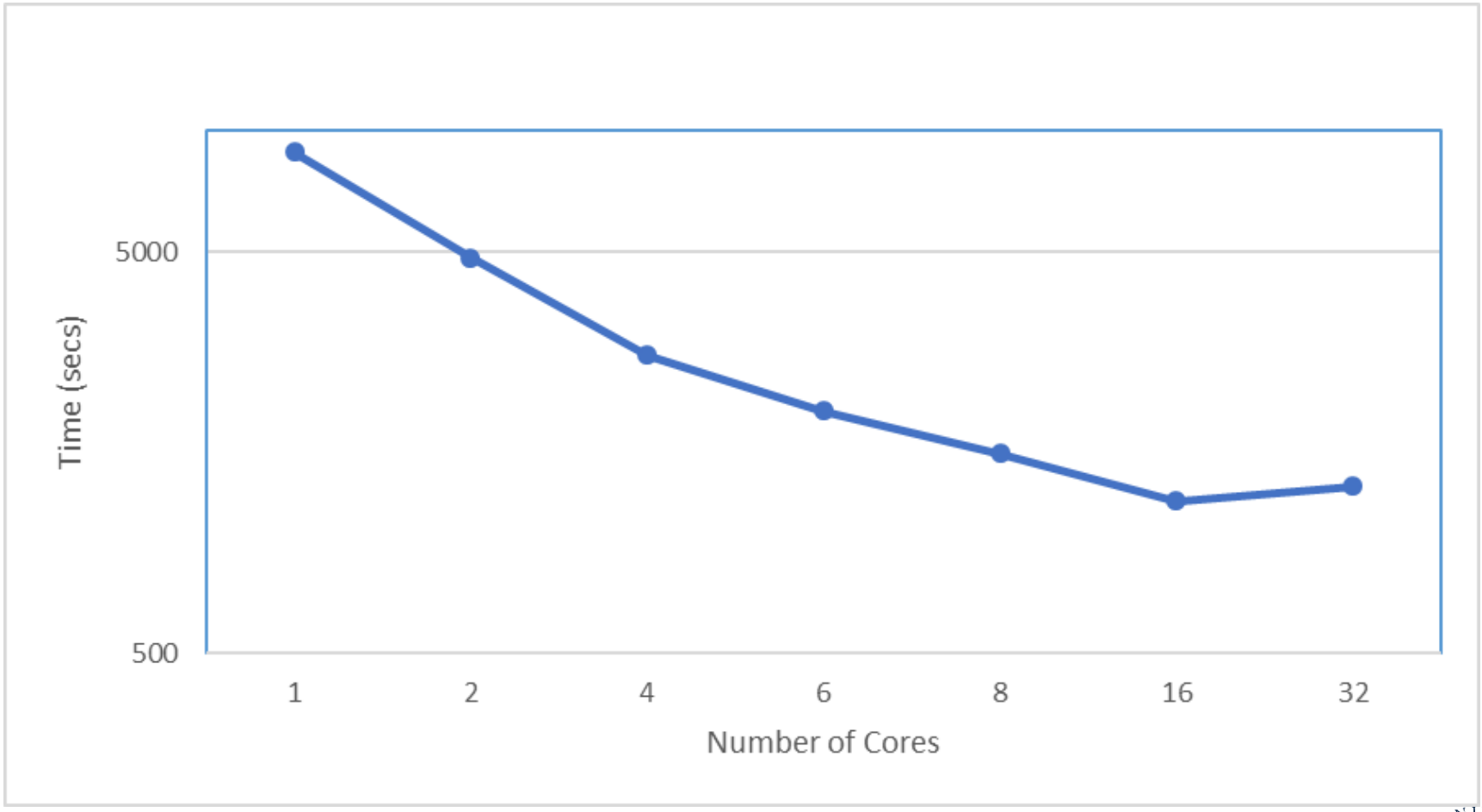
How to benchmark? (1/2)

- First, pick a **typical** simulation
- **Large enough** to reflect he **characteristics** of your **target simulation**
- **Small enough** not to burn too much resources
  - Aim for a simulation which does not run faster than 1 second
    - To avoid OS flutter
  - But not longer than, say, 10 hours
    - So we don't waste cycles *not* producing science

How to benchmark? (2/2)

- All benchmarks should be run multiple times
- Either **run three times**, in exclusive mode, and **take minimum**
  - Measures the fastest time your code ran
- Or run 10 times and note, min, max, mean and 95% confidence limits, say.
  - Measures how “busy” the machine was at the time of benchmarking

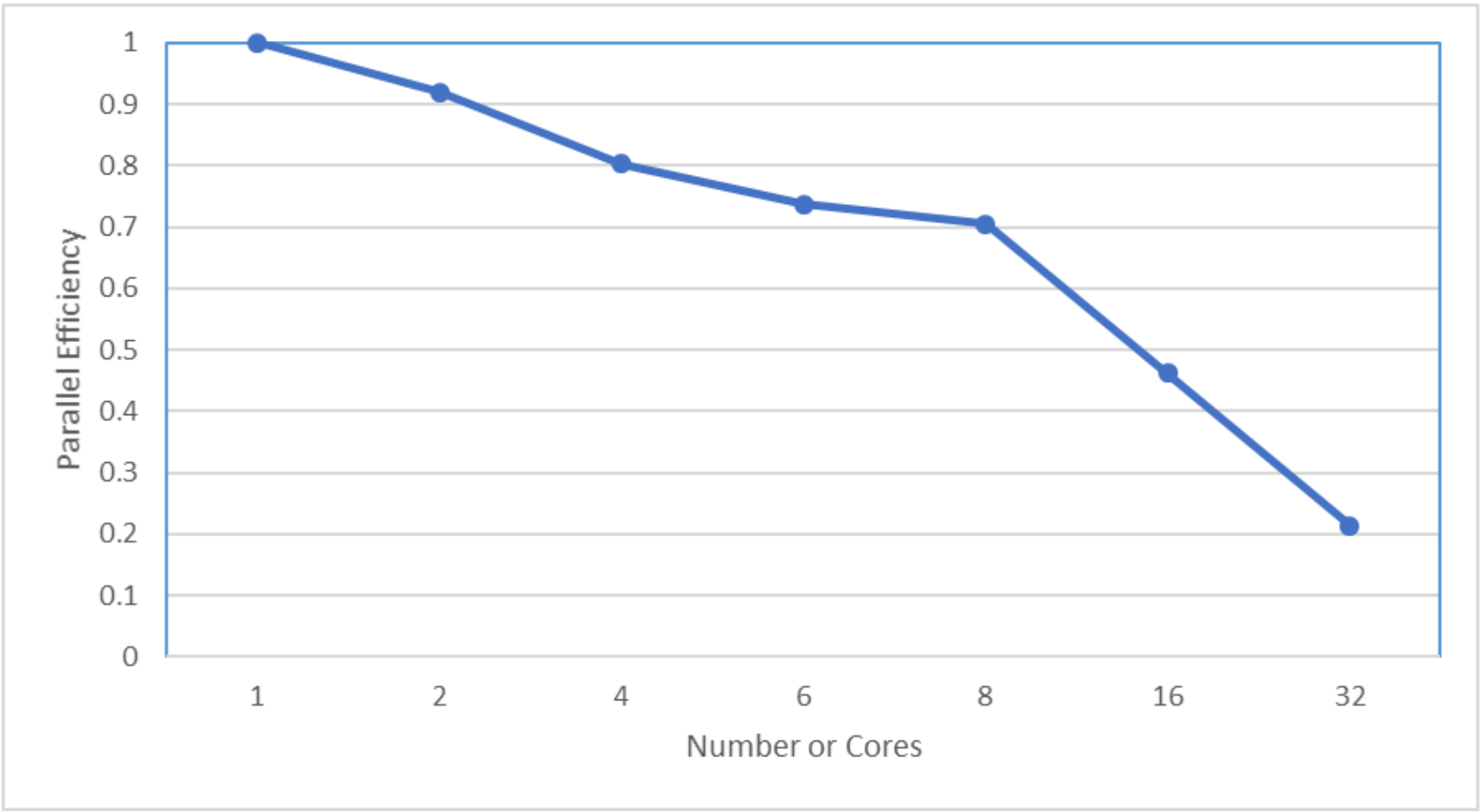
Example execution times



Execution times discussion

- Typical Execution Times graph
  - Log/log graphs present data clearly
  - As the core count increases: time reduces, levels out and then starts increasing
    - Communication starts to dominate over computation
- Execution times alone shows that 16 cores is the fastest
  - but is that an efficient use of the cores?
  - what about the Parallel Efficiencies?

Parallel Efficiencies



Discussion of Parallel Efficiencies

- A typical target parallel efficiency is 70%.
  - Some groups use 50%, others use 80%.
  - Using 70% gives the target number of cores as 8
- The efficiency for the fastest execution time
  - 16 cores has ~46%
    - Very poor efficiency
- The efficiency when using all the cores in one node?
  - 32 cores has ~21%
- Parallel Efficiency plots often reveal much more than execution times alone
  - i.e. “jump” from 8 to 16 cores may be due to NUMA region affects

So, target core count is 8?

- Maybe! It depends…

- Most efficient core count is 8 for this example
  - 16 may be faster but it is not efficient
  - Do not waste your project's shared time budget
    - "cycles"
- However, some HPC platforms charge you by the node
  - We would pay for all 32 cores
  - Fastest result now wins
    - 16 cores is best choice
      - NB further testing might show 24 cores is even faster

Benchmarking mixed-mode codes, i.e. MPI+OpenMP

- If code uses both MPI (or “tasks”) and OpenMP (or “threads”)
  - Set OMP_NUM_THREADS=1
  - Produce times for MPI tasks = 1,2,4,8,16,…
  - Then set OMP_NUM_THREADS=2
  - Then produce times for MPI tasks = 1,2,4,8,16,…
  - Then set OMP_NUM_THREADS=4
  - Then produce times for MPI tasks = 1,2,4,8,16,…
  - Continue until parallel efficiencies show adding more threads is futile
- It’s a lot of work
  - Tedious but essential

Summary

- There are a variety of considerations when parallelising code
- Scaling is important, as the more a code scales the larger a machine it can take advantage of
  - Scaling isn't everything: remember load balancing too!
- Metrics exist to give you an indication of how well your code performs and scales
  - Benchmarking is essential
  - Code's don't scale, simulations do.
  - Repeat for every simulation and for every machine
- Last word…

Last word