# CFD exercise

Regular domain decomposition






# Reusing this material




epcc



THE UNIVERSITY OF EDINBURGH

# Aims

- An introduction to geometric decomposition
  - Partitioning into sub-grids and assigning these to difference processes
  - Halo swapping for communications
- Gain hands on experience with performance metrics
- Understand in more detail how specific configuration choices can impact our performance
  - The choice of compiler
  - Level of optimisation

# Computational Fluid Dynamics

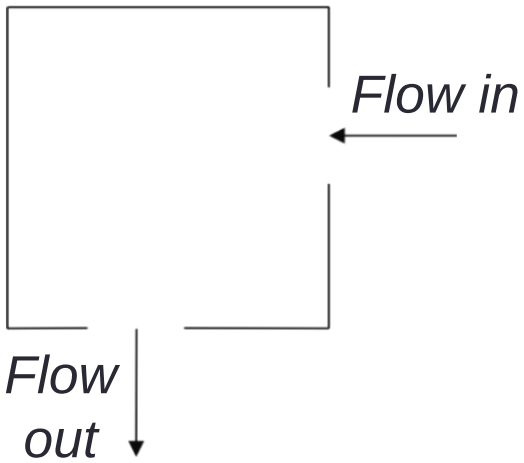
Algorithm, implementation and the problem

# Fluid Dynamics

- Study of the mechanics of fluid flow, liquids and gases in motion.
- Commonly requires HPC.
- Continuous systems typically described by partial differential equations.
- For a computer to simulate these systems, these equations must be *discretised* onto a grid.
- One such discretisation approach is the *finite difference method*.
- This method states that the value at any point in the grid is some combination of the neighbouring points

# The Problem

- Determining the flow pattern of a fluid in a cavity
  - a square box
  - inlet on one side
  - outlet on the other




- For simplicity, we assume zero viscosity for this exercise

# The Maths

- In two dimensions, easiest to work with the stream function $\Psi$
- At zero viscosity, $\Psi$ satisfies:

$$\nabla^2\Psi = \frac{\partial^2\Psi}{\partial x^2} + \frac{\partial^2\Psi}{\partial y^2} = 0$$

- With finite difference form:

$$\Psi_{i-1,j} + \Psi_{i+1,j} + \Psi_{i,j-1} + \Psi_{i,j+1} - 4\Psi_{i,j} = 0$$

- Jacobi iterative method can be used to find solutions
- With boundary values fixed, stream function can be calculated for each point by averaging value at that point with its four nearest neighbours.
  - process continues until the algorithm converges on a solution which stays unchanged by the averaging.
  - iterative methods are a very common computational approach used for solving systems of equations

# Jacobi iterative method

- To solve: $\Psi_{i+1,j} + \Psi_{i-1,j} + \Psi_{i,j+1} + \Psi_{i,j-1} - 4\Psi_{i,j} = 0$

  Repeat for many iterations:

  loop over all points *i* and *j:*

```
psinew[i][j] = 0.25*(  psi[i+1][j] + psi[i-1][j] +
                       psi[i][j+1] + psi[i][j-1]   )
```

  copy psinew back to psi for next iteration

- In the Fortran version of the code, array notation (arrays of size m x n) removes explicit loops:

```
psinew(1:m,1:n) = 0.25*(psi(2:m+1, 1:n) + psi(0:m-1, 1:n) +
                        psi(1:m, 2:n+1) + psi(1:m, 0:n-1)   )
```

# Notes

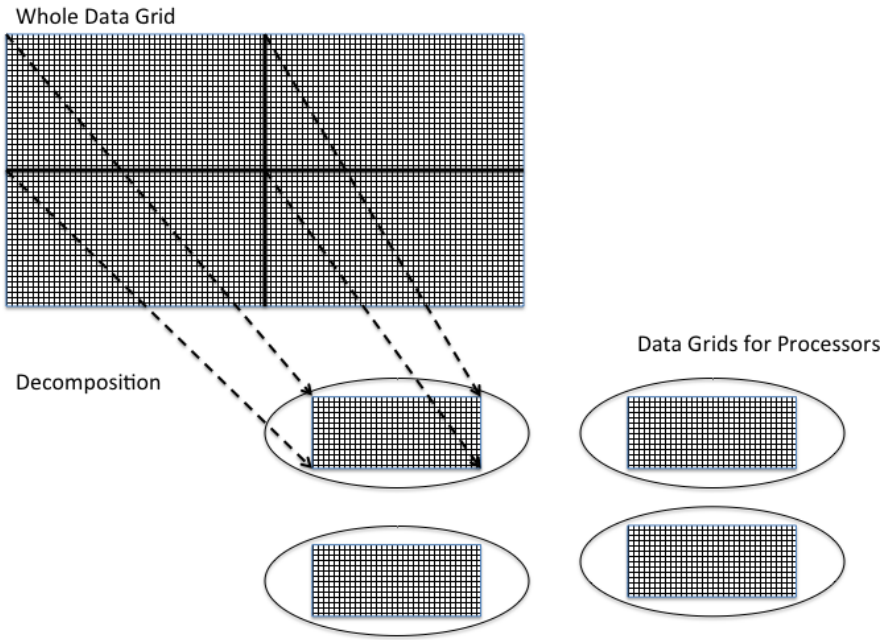
- Finite viscosity gives more realistic flows
  - introduces a new field zeta related to the vorticity
  - equations a bit more complicated but same basic approach
- Terminating the process
  - larger problems require more iterations
  - fixed number of iterations OK for performance measurement but not if we want an accurate answer
  - compute the RMS change in psi and stop when it is small enough
- There are many more efficient methods than Jacobi
  - But Jacobi is the simplest and easy to parallelise

# Parallelisations

How does our code take advantage of multiple processes?

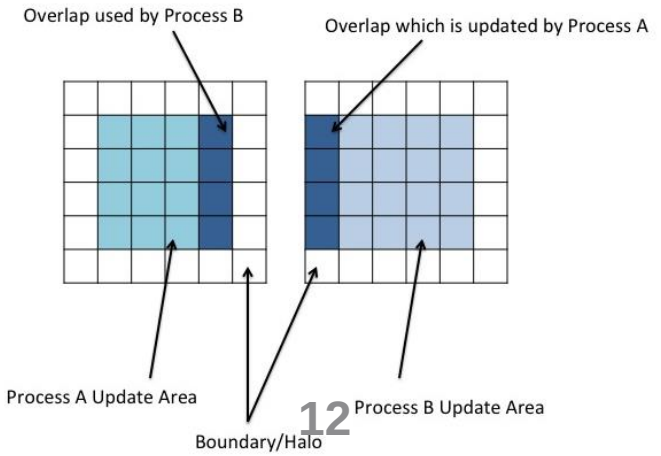
# Parallel Programming – Grids

- The algorithm involves calculating the value at each grid point by combining it with the value of its neighbours.
- Same amount of work needed to calculate each grid point – ideal for the geometric decomposition approach.
- Grid is broken up into smaller grids and one is allocated to each process.



# Parallel Programming – Halo Swapping

- Points on the edge of a grid present a challenge. Required data is shipped to a remote processor. Processes must therefore communicate.
- Solution is for processor grid to have a boundary layer on adjoining sides.
- Layer is not writable by the local process.
- Updated by another process which in turn will have a boundary updated by the local process.
- Layer is generally known as a *halo* and the inter-process communication which ensures their data is correct and up to date is a *halo swap*.



# Characterising Performance

- Speedup (*S*) is how much faster the parallel version runs compared to a non-parallel version.
- Efficiency (*E*) is how effectively the available processing power is being used.

$$S(N,P) = \frac{T(N,1)}{T(N,P)} \qquad E(N,P) = \frac{S(N,P)}{P} = \frac{T(N,1)}{P\,T(N,P)}$$

- Where:
  - *P* = number of processors
  - *N* = problem size (number of grid points)
  - *T*(*N*,1) time taken on 1 processor
  - *T*(*N*,*P*) time taken on *P* processors

# Over to you

Details of the exercise

# Practical

- Compile and run the code on ARCHER
  - on different numbers of cores
  - for different problem sizes
- Will return to this later to study compiler optimisation
  - following slides are for interest only

# Exercise outcomes

What do the timings tell us about HPC machines?

# Parallel Scaling – Number of Processors

- Addition of parallel resources subject to diminishing returns.
- Depends on scalability of underlying algorithms.
- Any sources of inefficiency are compounded at higher numbers of processes.
- In the CFD example, run time can become dominated by MPI communications rather than actual processing work.

| CFD Code | Iterations: 10,000 | Scale Factor: 40 | Reynolds number: 2 |
|---|---|---|---|
| MPI procs | Time | Speedup | Efficiency |
| 1 | 100.5 | 1.00 | 1.00 |
| 2 | 53.61 | 1.87 | 0.94 |
| 4 | 35.07 | 2.87 | 0.72 |
| 8 | 31.34 | 3.21 | 0.40 |
| 16 | 17.81 | 5.64 | 0.35 |

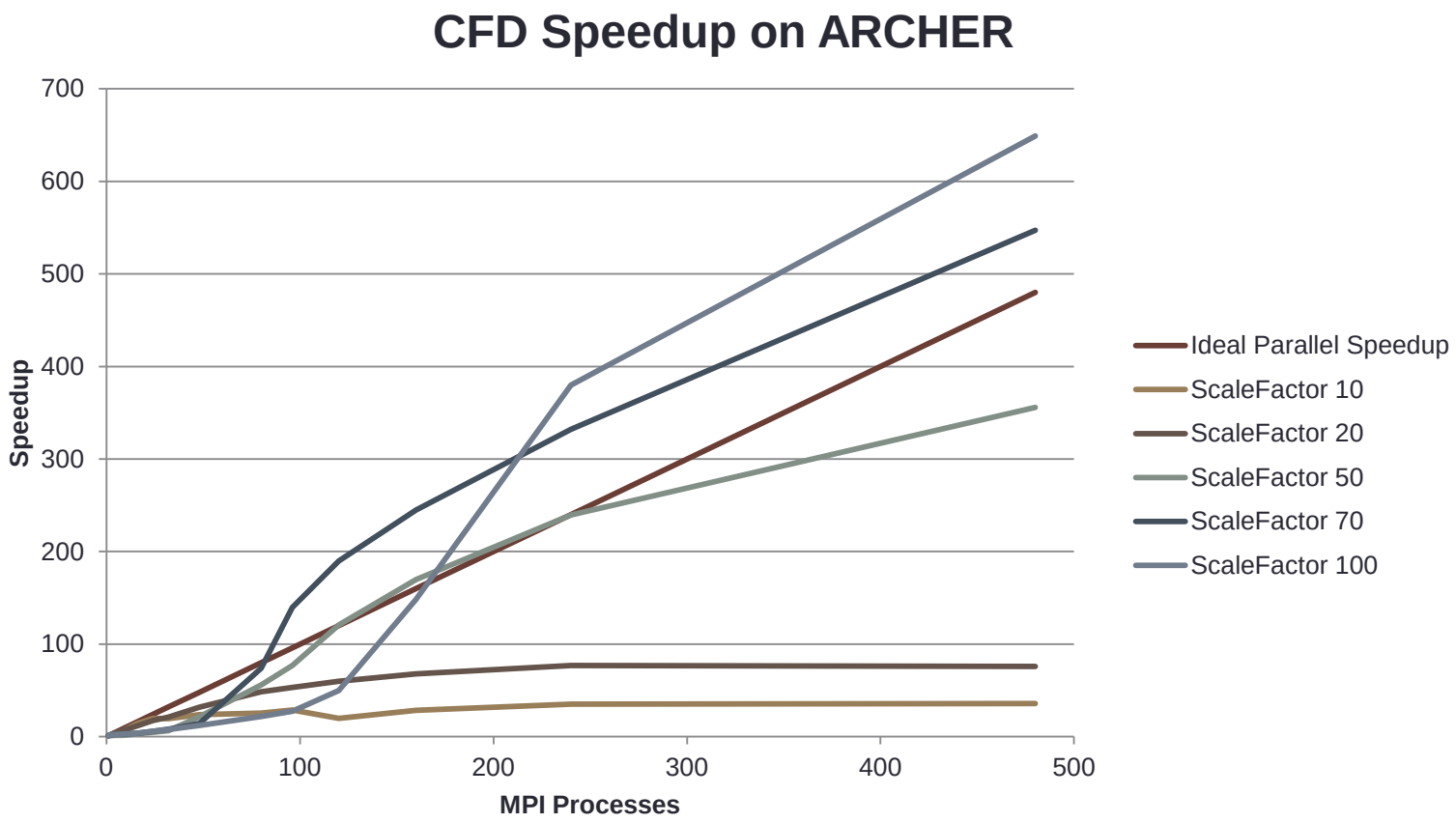
# Parallel Scaling – Problem Size

- Problem scale affects memory interactions – notably cache accesses.
- Additional processors provide additional cache space.
- Can lead to more, or even all, of a program's working set being available at the cache level.
- Configurations that achieve this will show a sudden efficiency "spike".

CFD Code Iterations: 10000 Scale Factor: 70

| MPI procs | Time | Speedup | Efficiency |
|---|---|---|---|
| 1 | 331.34 | 1.00 | 1.00 |
| 48 | 23.27 | 14.24 | 0.30 |
| 96 | 2.37 | 139.61 | 1.45 |

- 2x the number of MPI processes gives ~9.8x the speed up.

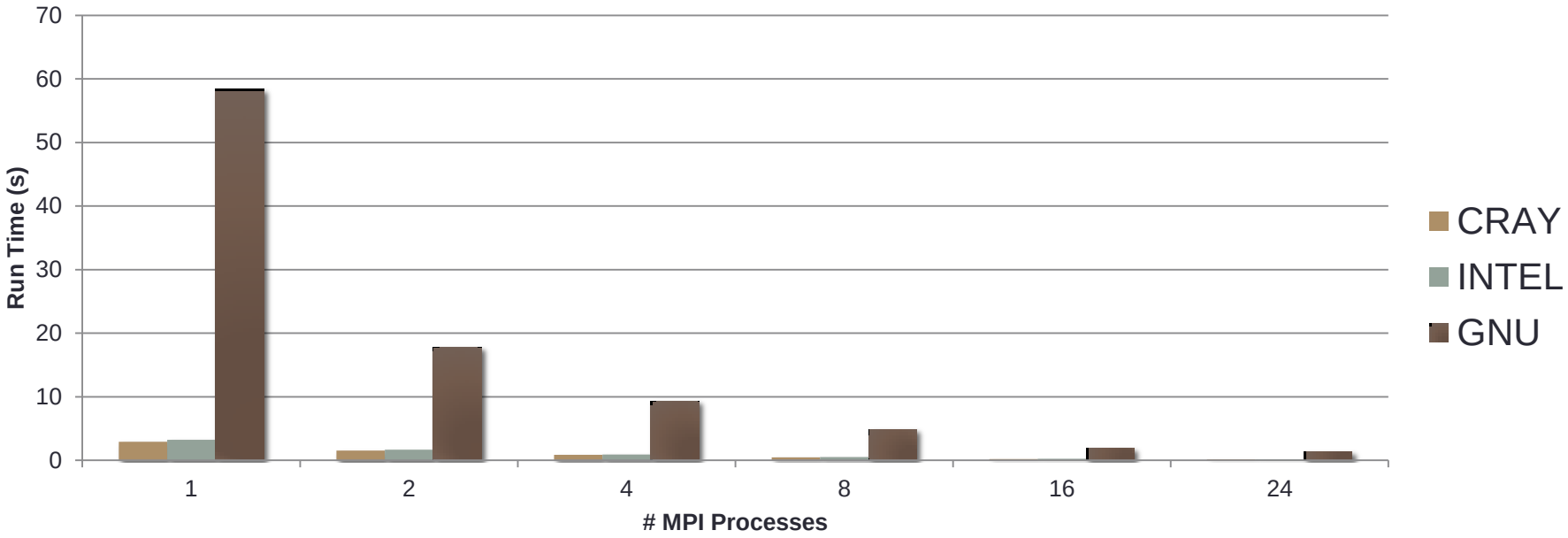

CFD Speedup on ARCHER
Speedup
MPI Processes
0
100
200
300
400
500
600
700
Ideal Parallel Speedup
ScaleFactor 10
ScaleFactor 20
ScaleFactor 50
ScaleFactor 70
ScaleFactor 100

# The impact of configuration choices

Different compilers, optimisations and hyper-threading

# Compiler Implementation and Platform

- Use ARCHER as an example, where we have the Cray, Intel and GNU compilers.
- Cray and Intel: more optimisations on by default, likely to give more performance out-of-the-box.
- ARCHER is a Cray system using Intel processors. Cray compiler tuned for the platform, Intel compiler tuned for the hardware.




- GNU compiler likely to require additional compiler options...

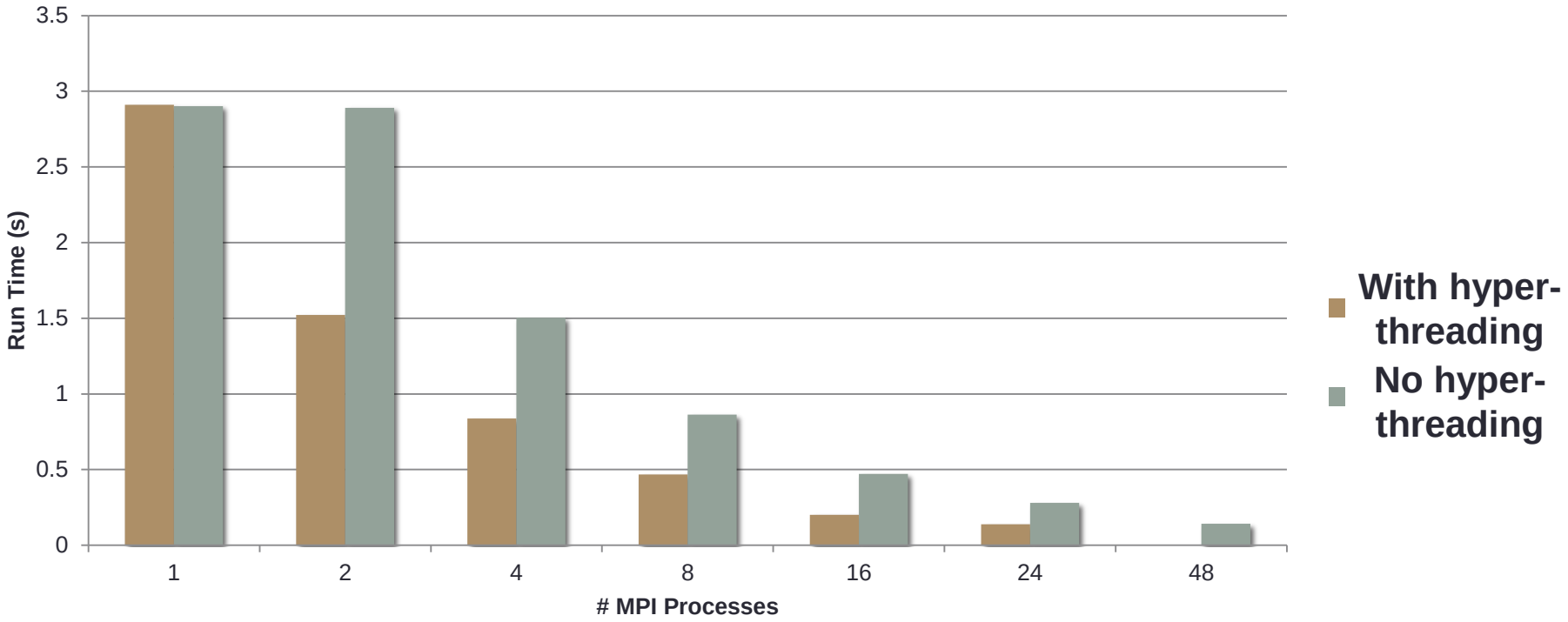
# Hyper-Threading

- Intel technology – designed to increase performance using simultaneous multi-threading (SMT) techniques.
- Presented as one additional *logical core* per physical one on the system.
- Each node therefore reports double available processors (48 on ARCHER, 72 on Cirrus).
- Must be explicitly requested with the “-j 2” option:

```
#PBS -l select=1
aprun -n 48 -j 2 ./myMPIProgram
```

- Hyper-Threading doubles the number of available parallel units per node at no additional resource cost.
- However, performance effects are highly dependent on the application

# Hyper-Threading Performance




- Can have a positive or negative effect on run times.
- Hyper-Threading is a bad idea for the CFD problem.
- Experimentation is key to determining if this technique would be suitable for your code.

# Process Placement

- Many HPC machines are NUMA systems – processors access different regions of memory at different speeds.
- In ARCHER compute nodes have two NUMA regions – one for each CPU. Hence 12 cores per region.
- It may be desirable to control which NUMA regions processes are assigned to.

- For example, with hybrid MPI and OpenMP jobs, it is suggested that processes are placed such that shared-memory threads in the same team access the same local memory.
- Can be controlled with *aprun* flags such as:
  - -N [parallel processes per node]
  - -S [parallel processes per NUMA region]
  - -d [threads per parallel process]