CFD Example

Aims

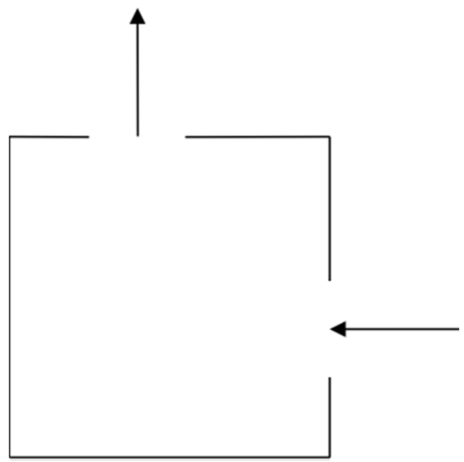
- To familiarise yourself with factors that affect code performance
    - compiler implementation and platform
    - compiler optimisation options
    - hyper-threading on ARCHER
    - process placement
    - parallel scaling
        - number of processors
        - problem size

Fluid Dynamics

- The 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



The Cavity

- For simplicity, assuming zero viscosity.

The Maths

- In two dimensions, easiest to work with the stream function Ψ
- At zero viscosity, Ψ 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 Method can be used to find solutions:
  - With boundary values fixed, stream function can be calculated for each point in the grid by averaging the value at that point with its four nearest neighbours.
  - Process continues until the algorithm converges on a solution which stays unchanged by the averaging.

The Maths

- In order to obtain the flow pattern of the fluid in the cavity we want to compute the velocity field: u
- The x and y components are related to the stream function by:

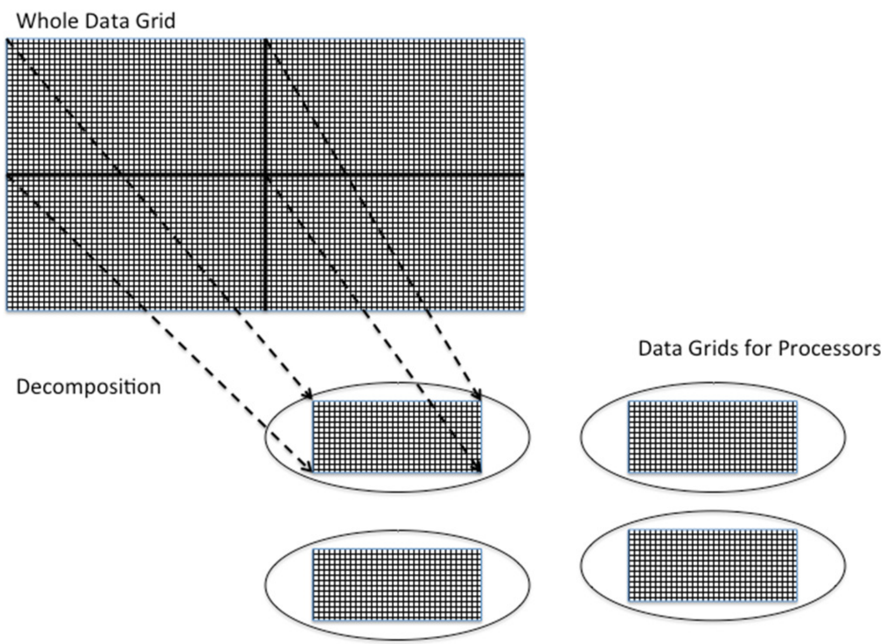
$$u_x = \frac{\partial \Psi}{\partial y} = \frac{1}{2}(\Psi_{i,j+1} - \Psi_{i,j-1})$$

$$u_y = -\frac{\partial \Psi}{\partial x} = \frac{1}{2}(\Psi_{i-1,j} - \Psi_{i+1,j})$$

- General approach is therefore:
  - Calculate the stream function.
  - Use this to calculate the two dimensions of the velocity.

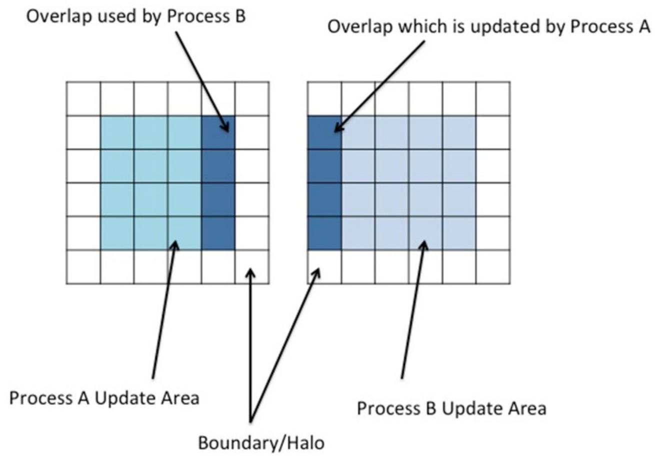
Parallel Programming – Grids

- Both stages involve 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 regular domain decomposition approach.
- Grid is broken up into smaller grids for each processor.



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

- Speed up (*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.

$$S = \frac{T_1}{T_N} \qquad E = \frac{S}{N} = \frac{T_1}{NT_N}$$

- Where:
    - N number of processors
    - T_1 time taken on 1 processor
    - T_N time taken on *N* processors

Compiling and Running the Practical

- A tar file is provided with
  - a Fortran CFD code
  - example job scripts
  - a Makefile for use with any PrgEnv module
- You should:
  - 1st Practical (Efficient compilation)
    - add optimisation flags to the Makefile, recompile and re-compare
    - compare different compilers
    - vary the number of processes
    - change the number of iterations and scale factor
  - 2nd Practical (Using the Intel Ivy-Bridge CPU)
    - experiment with hyper-threading on/off
    - spread processes between NUMA regions
    - vary the number of processes
    - change the number of iterations and scale factor
- See the exercise sheet for full details!