CFD example

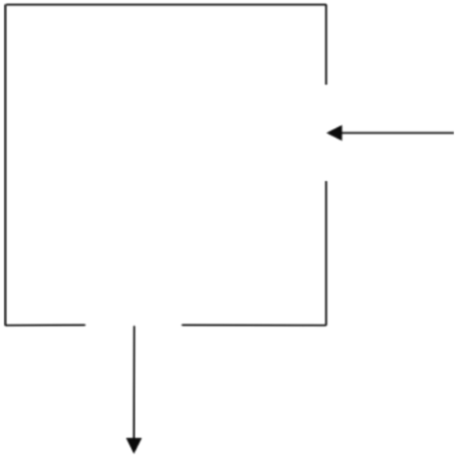
Regular domain decomposition

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.

Jacobi 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))`
  - end loop
  - copy psinew back to psi for next iteration
- until finished

- Fortran 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 algorithms than Jacobi
  - but Jacobi is very simple and easy to parallelise

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:

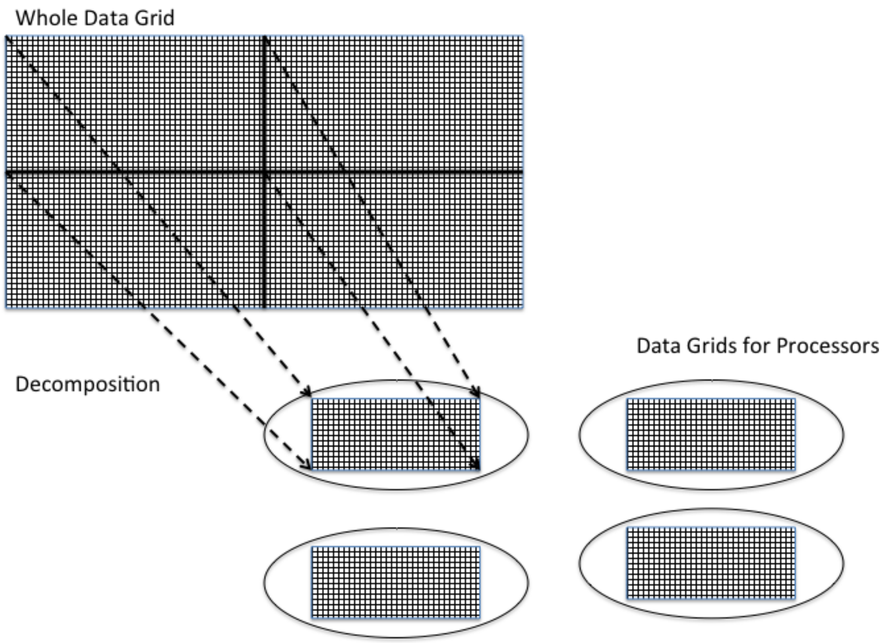
$$u_x = \frac{\partial \Psi}{\partial y} = \frac{1}{2}(\Psi_{i,j+1} - \Psi_{i,j-1})$$

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

- General approach is therefore:
  - Calculate the stream function Ψ
  - Use this to calculate the *x* and *y* components of the velocity *u*

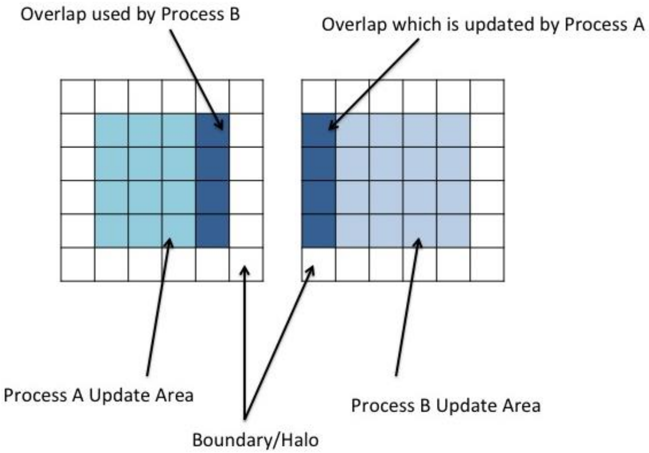
Parallel Programming – Grids

- 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*.



Overlap used by Process B
Overlap which is updated by Process A
Process A Update Area
Boundary/Halo
Process B Update Area

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

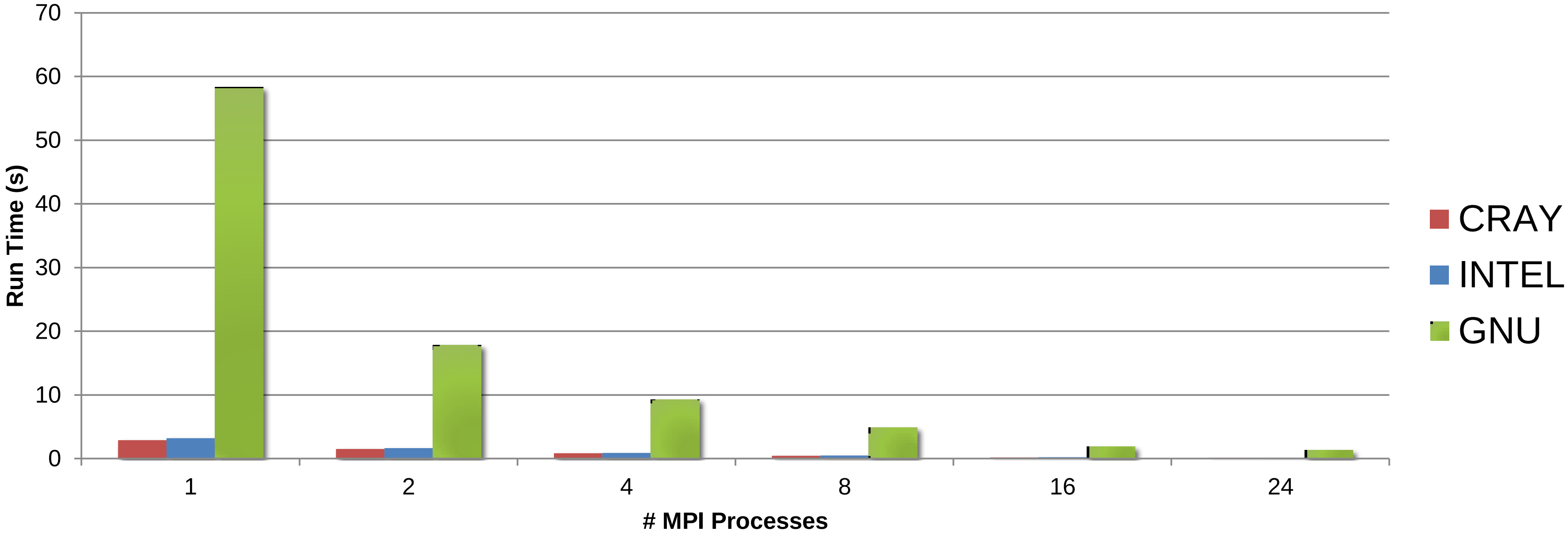
Practical

- 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

Compiler Implementation and Platform

- Three compilers on ARCHER: Cray, Intel and GNU.
- 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...

Compiler Optimisation Options

- Flags for the compiler. Can be set on the command line or in the Makefile.
- Standard levels:
  - O3 Aggressive
  - O2 Suggested
  - O Conservative
  - O0 Off (for debugging)
- Finer tuning available. Details in compiler man pages.
- Higher levels aren't always better. Increased code size from some optimisations may negatively impact cache interactions.
- Can increase compilation time.

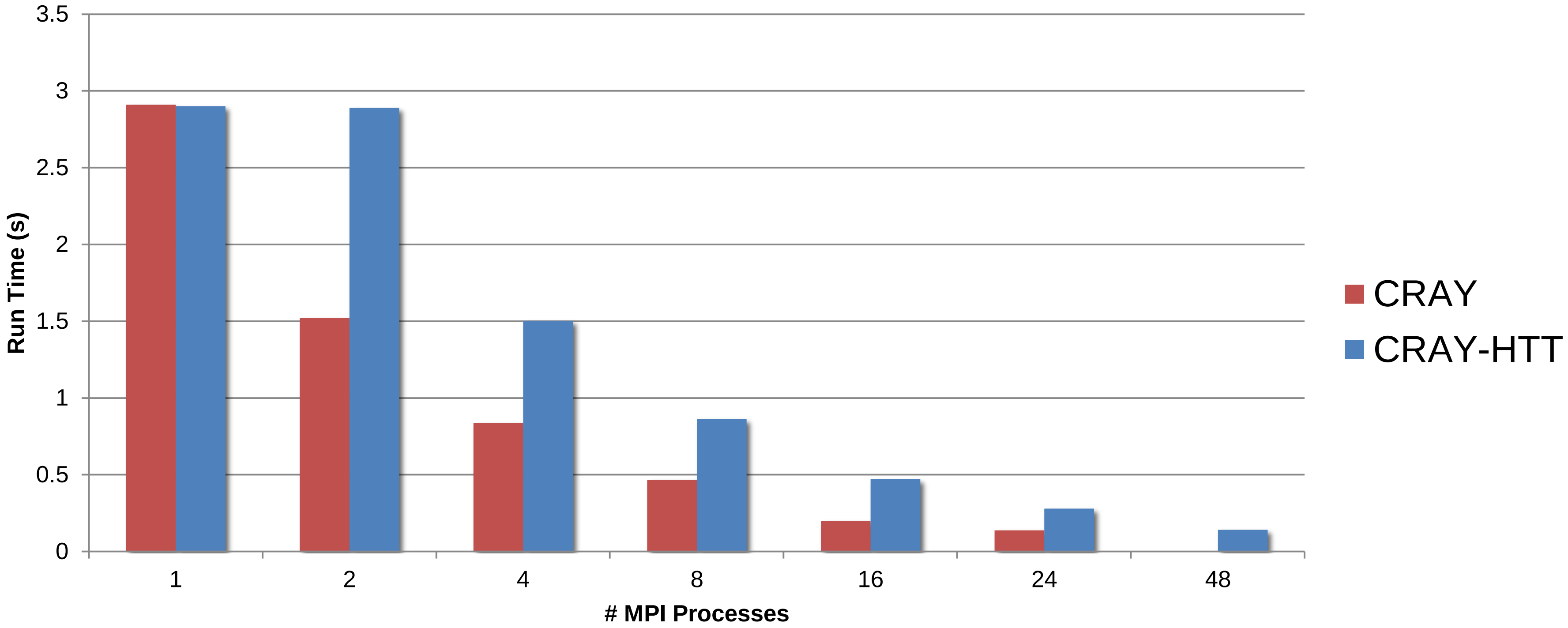
Hyper-Threading

- 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 ARCHER node therefore reports a total of 48 available processors (can be confirmed by checking /proc/cpuinfo).
- 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

- ARCHER is a NUMA system – processors access different regions of memory at different speeds.
- 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 hyrbid 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]

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.

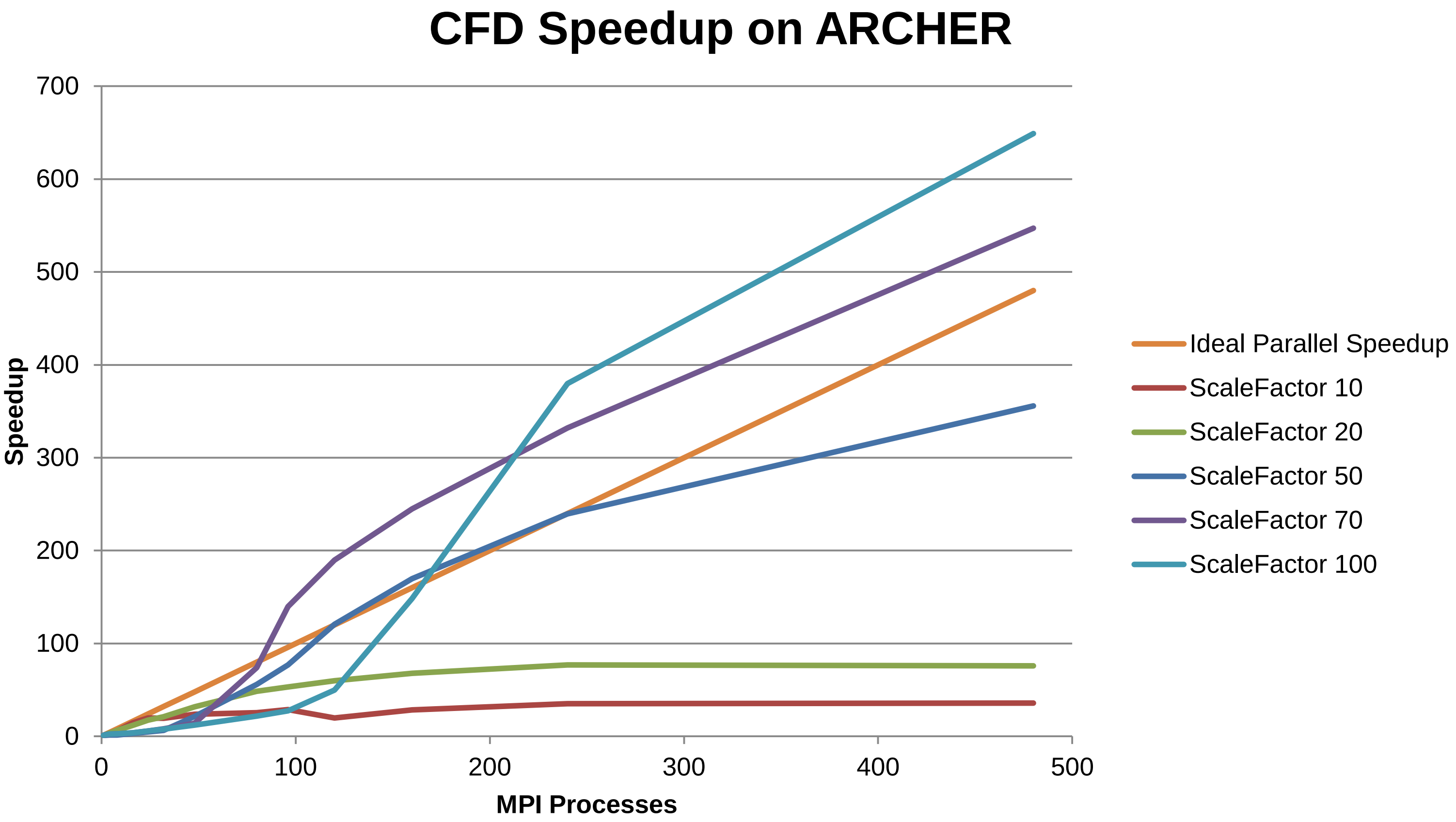
CFD Code	Iterations: 10,000	Scale Factor: 40	Reynolds number: 2
MPI procs	Time	Speedup	Efficiency
1	100.5	1.00	1.00
2	53.61	1.87	0.94
4	35.07	2.87	0.72
8	31.34	3.21	0.40
16	17.81	5.64	0.35

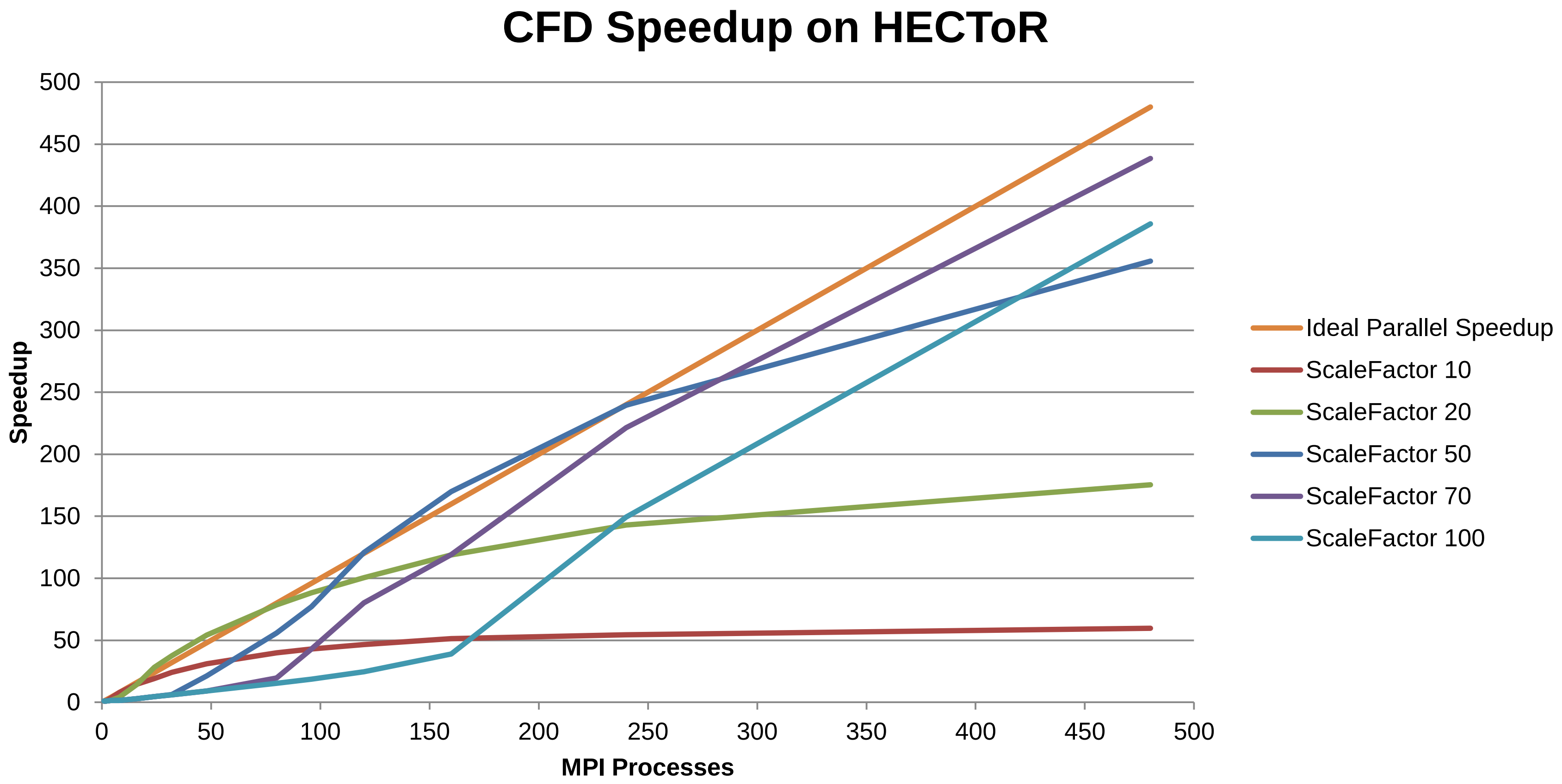
Parallel Scaling – Problem Size

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


CFD Speedup on HECToR
Speedup
500
450
400
350
300
250
200
150
100
50
0
0
50
100
150
200
250
300
350
400
450
500
MPI Processes
Ideal Parallel Speedup
ScaleFactor 10
ScaleFactor 20
ScaleFactor 50
ScaleFactor 70
ScaleFactor 100

ARCHER-ScaleFactor 10

MPI procs	Time	Speedup	Efficiency
1	2.91	1.00	1.00
2	1.52	1.91	0.96
4	0.84	3.47	0.87
8	0.47	6.22	0.78
16	0.20	14.46	0.90
24	0.15	19.92	0.83
32	0.15	19.45	0.61
48	0.12	23.90	0.50
80	0.11	25.63	0.32
96	0.10	28.95	0.30
120	0.15	19.78	0.16
160	0.10	28.36	0.18
240	0.08	35.14	0.15
480	0.08	35.87	0.07

ARCHER-ScaleFactor 20

MPI procs	Time	Speedup	Efficiency
1	11.92	1.00	1.00
2	6.21	1.92	0.96
4	3.38	3.52	0.88
8	1.86	6.41	0.80
16	1.00	11.91	0.74
24	0.68	17.52	0.73
32	0.57	21.03	0.66
48	0.37	31.95	0.67
80	0.25	48.43	0.61
96	0.22	53.17	0.55
120	0.20	59.86	0.50
160	0.18	67.90	0.42
240	0.16	76.77	0.32
480	0.16	75.94	0.16

HECToR-ScaleFactor 10

MPI procs	Time	Speedup	Efficiency
1	8.91	1.00	1.00
2	8.01	1.11	0.56
4	2.77	3.21	0.80
8	1.12	7.99	1.00
16	0.61	14.56	0.91
24	0.46	19.16	0.80
32	0.37	24.28	0.76
48	0.29	31.00	0.65
80	0.22	39.80	0.50
96	0.21	43.06	0.45
120	0.19	46.47	0.39
160	0.17	51.25	0.32
240	0.16	54.58	0.23
480	0.15	59.81	0.12

HECToR-ScaleFactor 20

MPI procs	Time	Speedup	Efficiency
1	48.42	1.00	1.00
2	44.30	1.09	0.55
4	30.68	1.58	0.39
8	11.97	4.04	0.51
16	3.34	14.49	0.91
24	1.71	28.27	1.18
32	1.29	37.59	1.17
48	0.89	54.28	1.13
80	0.62	78.63	0.98
96	0.55	88.33	0.92
120	0.48	100.57	0.84
160	0.41	118.94	0.74
240	0.34	143.04	0.60
480	0.28	175.50	0.37

ARCHER-ScaleFactor 100

MPI procs	Time	Speedup	Efficiency
1	694.66	1.00	1.00
2	378.47	1.84	0.92
4	272.62	2.55	0.64
8	250.92	2.77	0.35
16	184.39	3.77	0.24
24	121.45	5.72	0.24
32	88.64	7.84	0.24
48	56.98	12.19	0.25
80	31.66	21.94	0.27
96	25.26	27.50	0.29
120	13.89	50.02	0.42
160	4.68	148.34	0.93
240	1.83	379.89	1.58
480	1.07	648.81	1.35

ARCHER-ScaleFactor 150

MPI procs	Time	Speedup	Efficiency
1	1577.00	1.00	1.00
2	856.87	1.84	0.92
4	617.34	2.55	0.64
8	569.49	2.77	0.35
16	423.34	3.73	0.23
24	280.15	5.63	0.23
32	207.53	7.60	0.24
48	134.89	11.69	0.24
80	77.95	20.23	0.25
96	69.59	22.66	0.24
120	53.61	29.42	0.25
160	37.43	42.14	0.26
240	19.89	79.30	0.33
480	4.96	317.79	0.66

HECToR-ScaleFactor 100

MPI procs	Time	Speedup	Efficiency
1	1229.85	1.00	1.00
2	1135.95	1.08	0.54
4	810.08	1.52	0.38
8	803.56	1.53	0.19
16	404.02	3.04	0.19
24	270.39	4.55	0.19
32	203.32	6.05	0.19
48	135.61	9.07	0.19
80	80.72	15.24	0.19
96	66.10	18.61	0.19
120	50.12	24.54	0.20
160	31.63	38.88	0.24
240	8.23	149.44	0.62
480	3.19	385.72	0.80

HECToR-ScaleFactor 150

MPI procs	Time	Speedup	Efficiency
1	2794.46	1.00	1.00
2	2545.46	1.10	0.55
4	1823.64	1.53	0.38
8	1803.73	1.55	0.19
16	903.92	3.09	0.19
24	604.05	4.63	0.19
32	454.35	6.15	0.19
48	304.80	9.17	0.19
80	183.54	15.23	0.19
96	152.96	18.27	0.19
120	122.20	22.87	0.19
160	91.26	30.62	0.19
240	58.37	47.87	0.20
480	11.20	249.48	0.52