DiRAC: Applying for computing time

Chris Johnson
Resource Allocation Manager, DiRAC HPC Facility

Swansea, 3rd July 2026

The CCP-TEPP Summer School in Software Skills for Particle Physics

DiRAC

UKRI Science and Technology Facilities Council

Outline

- Who am I
- epcc
- DiRAC
- Other compute
- RAC calls
- Applying to RAC
  - Scaling case
  - Amount of resources
- Running a parallel code
  - CFD example
  - Discretisation
  - Parallelisation
  - Exercise

epcc

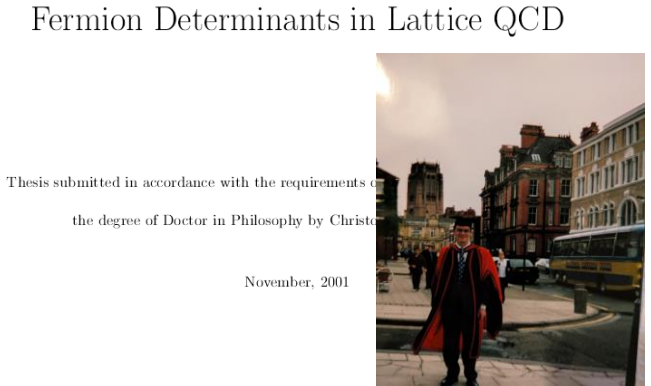
Acknowledgements

- Material comes from DiRAC
  - Mark Wilkinson (DiRAC Director)
  - Simon Hands (DiRAC Community Development Director)
- EPCC MSc courses
  - David Henty (leads EPCC's HPC training activities)
  - Weronika Filinger (EPCC training)
- *Et al.*

Background

- Maths/Physics degree (1994-97)
- Computing MSc (1997-98)
- Theoretical particle physics (Lattice QCD) PhD (1998-2001)
  - Involved eigensolvers (Lanczos) and parallel computers (Cray T3E)

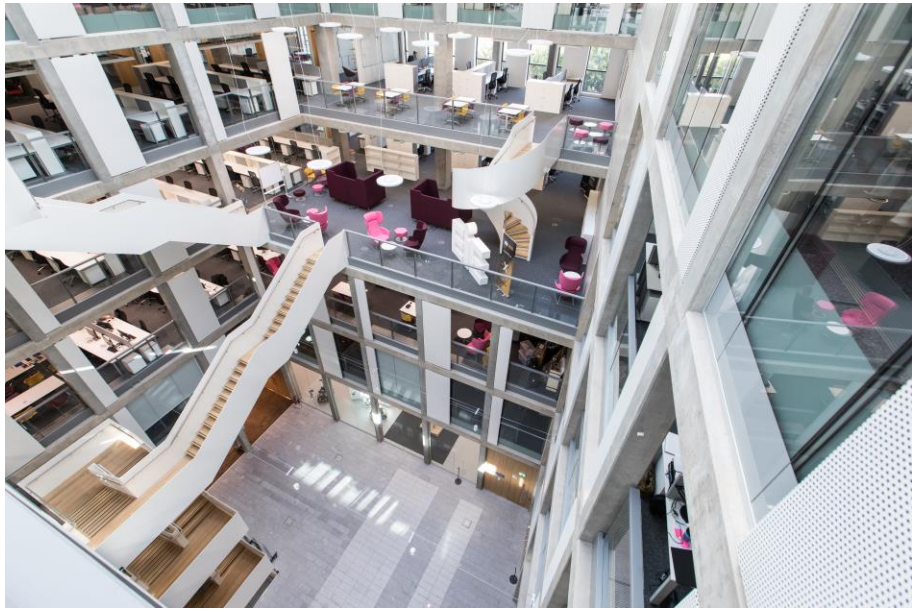
THE UNIVERSITY of LIVERPOOL



- Joined EPCC in 2001
  - Worked on HPC projects and courses, coord MSc numerical algorithms course, user support, focussed on funding calls
  - Presently manage resources for DiRAC (including helping with yearly RAC calls), jointly run MSc projects course and coord eCSE software development calls

What is EPCC?

- Founded in 1990 (originally Edinburgh Parallel Computing Centre)
- A self-funding institute at The University of Edinburgh
- Running national parallel systems since Cray T3D in 1994



- Around 140 staff
- Now located in the Bayes Centre just south of city centre
- Advanced Computing Facility (ACF) located ~6 miles south of city centre

EPCC's Advanced Computing Facility

- ACF was built in 1970s
- 4 computer rooms
- Also associated plant rooms

- Cooling is a major part of running a supercomputer…
- …fortunately in Scotland it's often cold!

We host and support ARCHER2, the UK's National Supercomputing Service.

With 750,080 AMD Rome cores, it is one of the largest cores-only systems in the world.

The DiRAC HPC Facility

UKRI Science and Technology Facilities Council

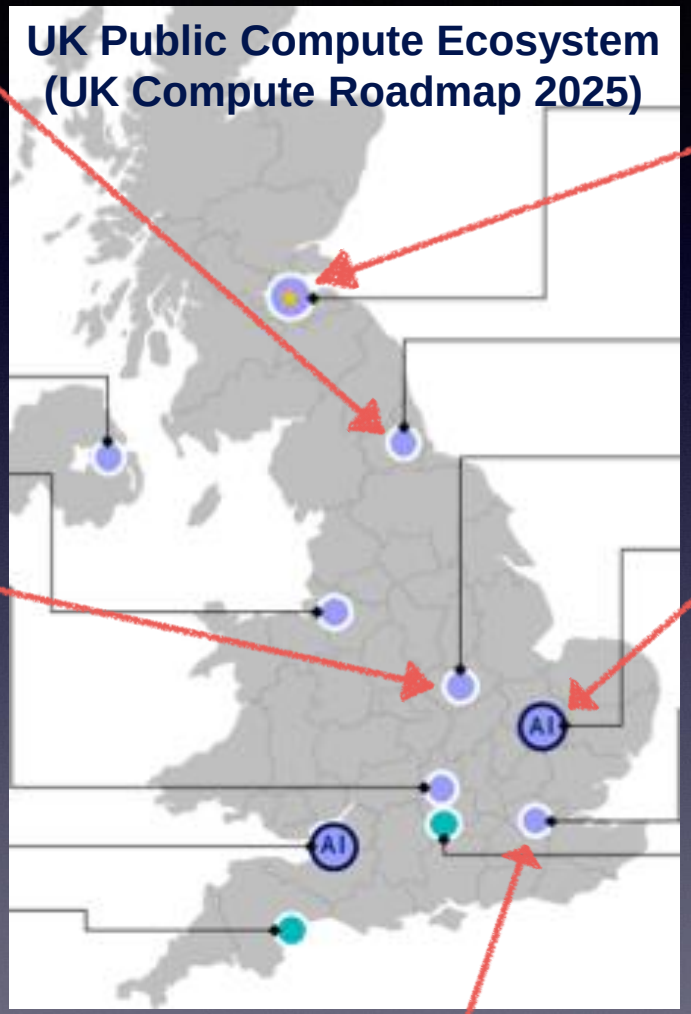
DiRAC

Memory Intensive "COSMA8" (Durham)

DELL EMC

- 528 TB RAM
- Large-scale cosmological simulations

UK Public Compute Ecosystem (UK Compute Roadmap 2025)



Facility Office (UCL)

Extreme Scaling "Tursa" (Edinburgh)

EVIDEN an atos business

- 704 Nvidia A100 GPUs
- Large lattice-QCD simulations

Data Intensive "DiaL" (Leicester)

- Heterogeneous architecture for complex simulation and modelling workflows

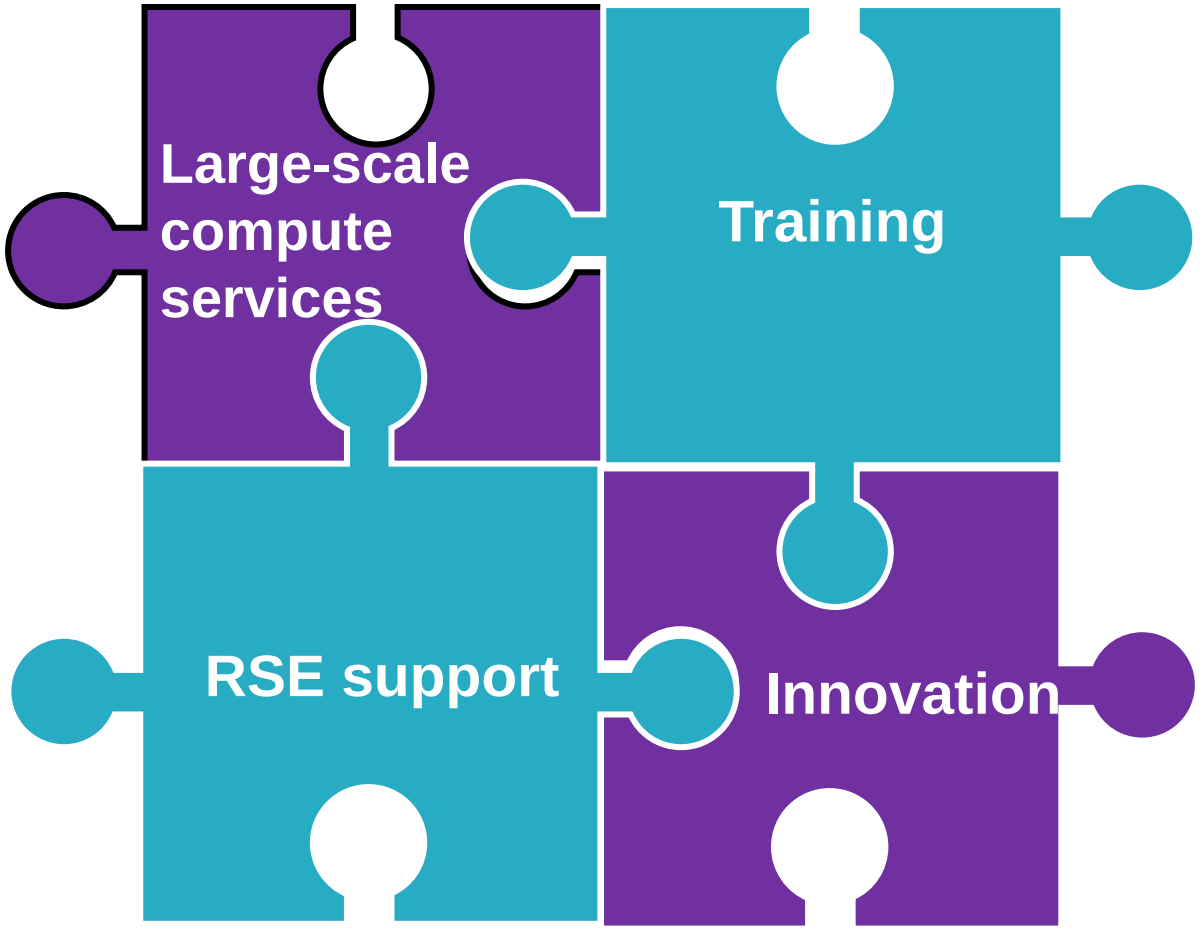
Hewlett Packard Enterprise

Data Intensive "CSD3" (Cambridge)

DELL EMC

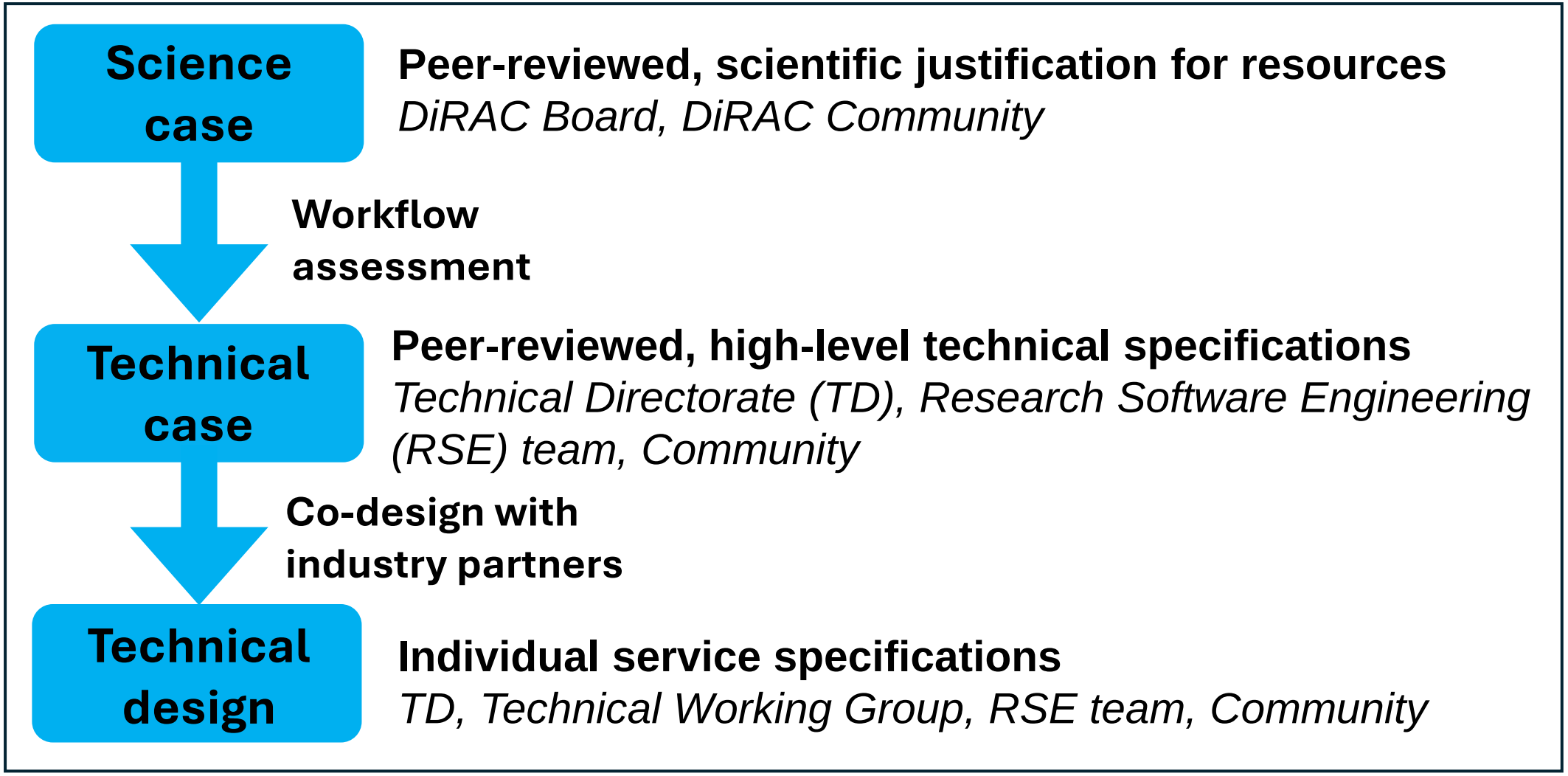
- Heterogeneous architecture for complex simulation and modelling workflows

The DiRAC HPC Facility



- National facility for PPAN Theory Communities
- Data challenges growing rapidly:
  - individual simulations 10Pb+
  - Increasing use of AI/ML techniques to enhance simulation methods
- Access for non-STFC researchers:
  - AI models of the cancer genome (Miller et al)
  - UK-LLM (Stenetorp et al.), the first LLM to be trained solely using UK-based computing resources, was trained on DiRAC (Tursa cluster)

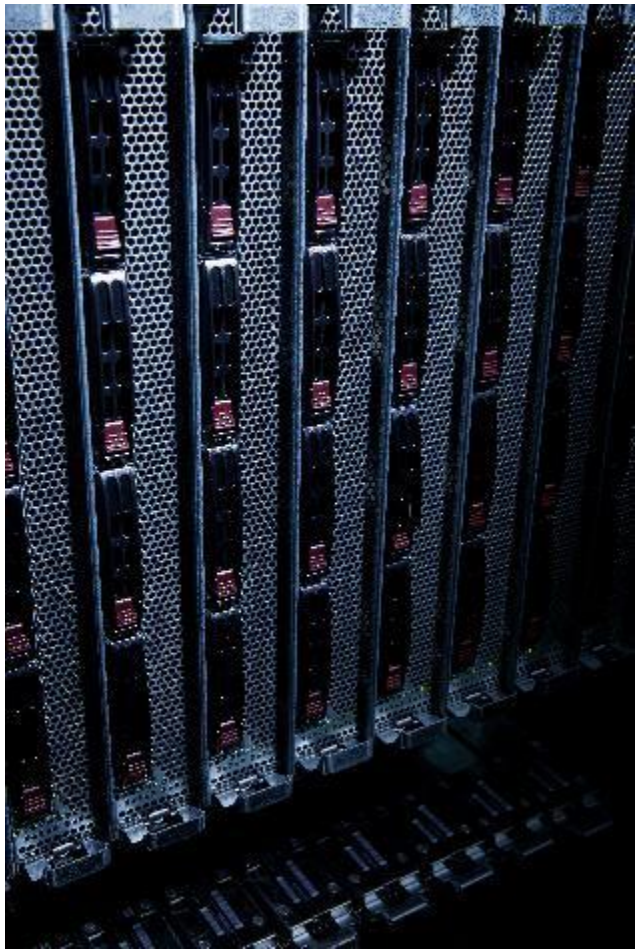
Collaborative co-design of DiRAC services



- Science case determines *both* scale and design of DiRAC services
- Co-design applies to *both* hardware and software (middleware and applications)

National Compute Resources (NCRs)

- UKRI is investing in NCRs to build an integrated public compute ecosystem to meet a wide range of needs
- NCRs will complement flagship investments – NNSS and AIRR. Smaller scale services that will support research across the UKRI remit
- NCR model is an evolution of approaches to supporting ‘Tier 2’-scale compute
- Phase 1 services will come online during FY26/27.
- **Phase 2 NCR specification during FY26/27 with deployments expected to start in FY27/28.**
- **Current DiRAC CPU services supported until at least March 2028**
- **DiRAC users may apply to use NCRs**
- **Tursa users may be awarded NCR time after end of Tursa service in September 2027 – subject to on-going discussions between STFC and UKRI DRI**



NCR Phase 1: specifications

CPU-based systems

- Cirrus NCR (University of Edinburgh): 640 compute nodes consisting of 2 x AMD Turin CPUs; 184,320 cores; 200Gbps interconnect
- Charger NCR (UCL/DataVita): 200-235 x Intel Xeon 6980P CPUs; 37000 cores; 400Gb/s interconnect; 8 Blackwell RTX Pro 6000 visualisation GPUs, ~850 CPU cores
- Isambard 3 (existing service, University of Bristol): 380 Nodes consisting of NVIDIA Grace-Grace CPUs; 54,720 cores; 200Gb/s interconnect

GPU-based systems

- Baskerville NCR (University of Birmingham): 200x Nvidia B200 GPUs; 23 nodes; 400Gb/s interconnect
- Zenith NCR (University of Cambridge): 28 nodes consisting of 8 AMD MI355X GPUs (224 total); 400 Gb/s interconnect
- Mary Coombs NCR (Daresbury): 628 NVIDIA H100 GPUs (157 nodes), Infiniband NDR200 interconnect

Wider compute ecosystem: AI Research Resource (AIRR)

- Isambard AI: 5,448 Nvidia GH200 superchips
- DAWN: 768 Intel Data Centre GPU Max 1550
- Access Routes open:
  - Rapid Access
  - Gateway Innovator

- **If your work uses any AI then AIRR is open to you**
  - If not – stay tuned….

Isambard-AI

DAWN

Wider compute ecosystem: EuroHPC

- UK is a full member of the EuroHPC Joint Undertaking

- Through EuroHPC Access Calls, UK-based applicants can access EuroHPC systems, including the JUPITER

- We encourage applications for access https://eurohpc-ju.europa.eu/access-our-supercomputers/eurohpc-access-calls_en

- Planning workshops on EuroHPC access.

Next National Supercomputing Service (NNSS)

- UKRI-led project for a new National Supercomputing Service that will be based at EPCC in Edinburgh
- Pan-UKRI service and part of a wider compute ecosystem
- Investment of up to £750m for:
  - Hardware
  - Host site and operating costs (e.g., electricity)
  - Service provision
  - Software programme
- Service currently planned to go live in Q4 2027

- **Lattice QFT requirements are part of specification**
  - **Grid is one of the procurement benchmarks**
  - **Exact hardware configuration will depend on tender bids**

- **Tursa supported until Sept 2027**
  - **mitigation plan under development for gap between Tursa and NNSS.**

RAC calls (compute resources)

DiRAC

- Seedcorn proposals:
  - Up to 3 months: code/system testing; proposal preparation
  - Light-touch review process: decisions within ~1 week
  - Allocations up to 100k CPU core hours or 1k GPU hours
- Resource Allocation Committee (RAC) calls
  - Project categories: small (up to 12 months) and large (up to 3 years)
  - Bids may request use of more than one DiRAC service
  - No bid may ask for >80% of a given DiRAC machine
  - Annual calls – open in June; close in September, projects start following April
  - Next call: RAC19 – for allocations starting in April 2027
- Discretionary proposals
  - Requests that don't fit into the above
  - Contact DiRAC Director

RAC calls (humans effort)

DiRAC

- RSE effort
  - Projects from 3 months to multiple years (typically 3-12 person months effort)
  - help optimise codes, improve workflows, and port to new hardware, but not to develop new science capability
  - 5 FTE of staff effort available

Applying for DiRAC resources – next call!

- Access to DiRAC resource via an annual call from the Resource Allocation Committee (RAC)
- RAC is an STFC-supported standing committee, completely independent of DiRAC
- Open to all researchers including ECRs
- This year's deadline for RAC19: **Thursday 17th September 2026 16:00 UK time** https://dirac.ac.uk/getting-access/
- Apply via UKRI's The Funding Service portal
  - Webinar: Wednesday 8th July at 1:30-3:30pm (zoom)
- RAC calls are routinely oversubscribed: the resource awarded may be tapered according to ranking, provided the project remains feasible

DiRAC RAC – constructing a technical case

- Resource Allocation Committee (RAC) calls
  - Every RAC proposal requires a scientific case and a technical case
  - We'll concentrate here on the technical case
    - In particular resources being requested and scaling case
  - Technical case form consists of 6 sections, including
    - codes to be used (status and underlying dependencies)
    - Performance and scaling of codes to be used
      - Previously had separate weak/strong scaling – now have one scaling section where you give whatever scaling is appropriate (weak, strong, or both weak and strong)
    - What jobs are to be run (typical and largest)…
    - …and where (request specific systems)
    - Compute resources requested
      - Split into years (and quarters for first year)
    - Storage requested
    - Data production/transfer

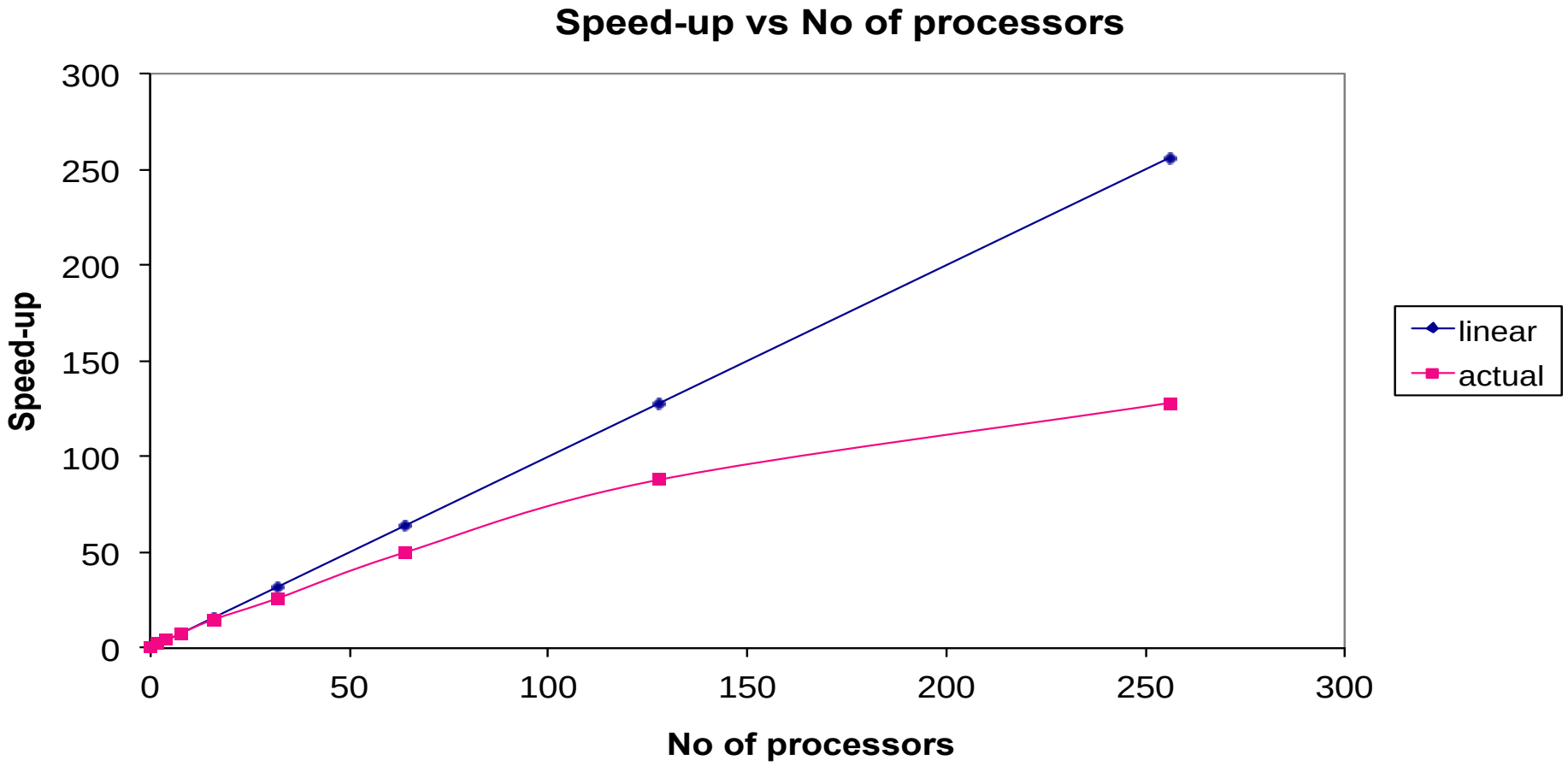
Performance metrics

- Fundamental measurement is the runtime *T*
  - we can vary the number of processes *P* or the problem size *N*
  - *N* measures the amount of computation, e.g. number of grid points
- Speed up $S(N,P) < P$ $$S(N,P) = \frac{T(N,1)}{T(N,P)}$$
  - typically
- Parallel efficiency $E(N,P) < 1$ $$E(N,P) = \frac{S(N,P)}{P} = \frac{T(N,1)}{P\,T(N,P)}$$
  - typically
- Serial efficiency $E(N) <= 1$ $$E(N) = \frac{T_{best}(N)}{T(N,1)}$$
  - typically

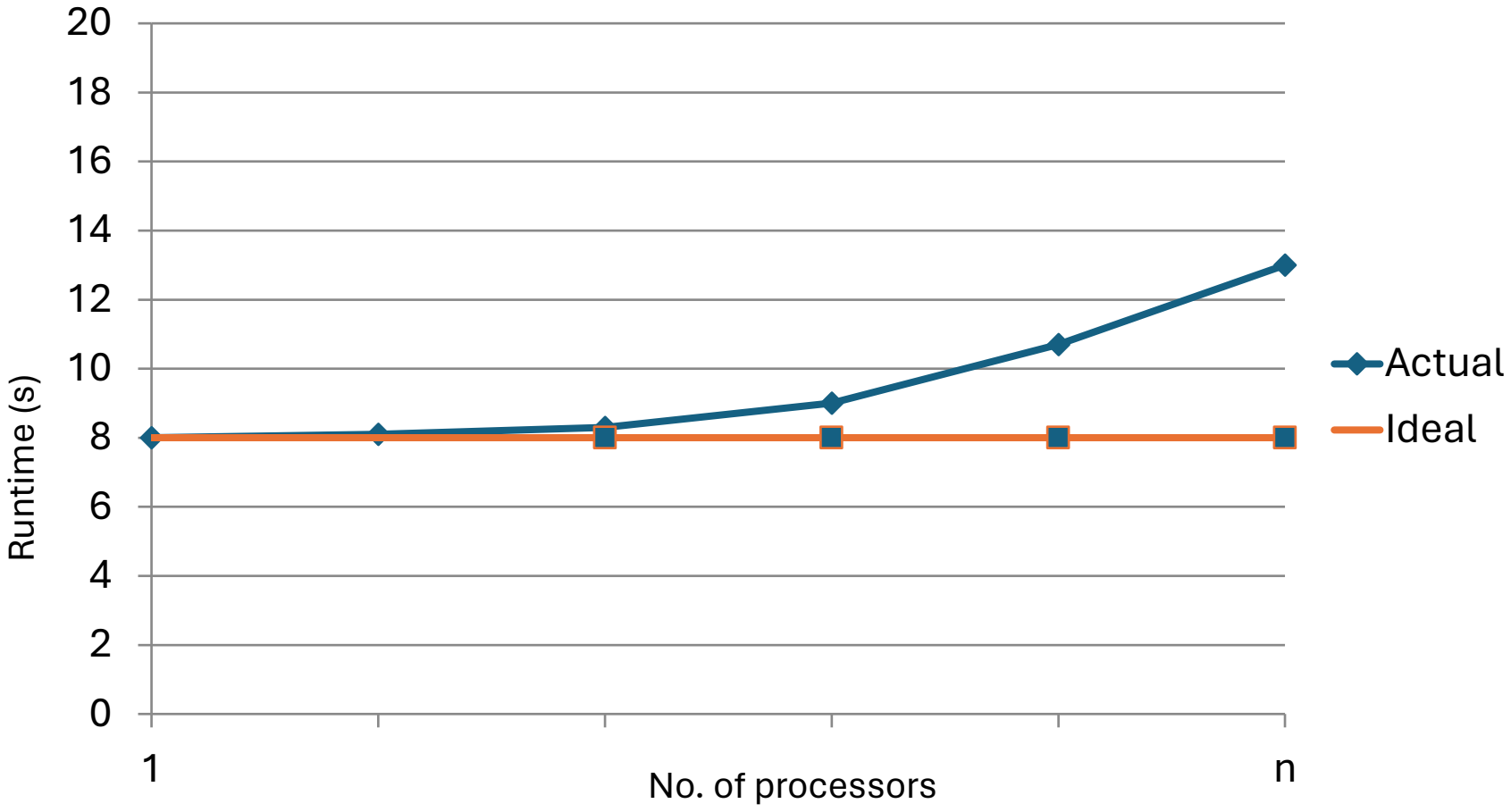
Scaling

- *Scaling* is how the performance of a parallel application changes as the number of processors is increased
- There are two different types of scaling:
  - *Strong Scaling* – total problem size stays the same as the number of processors increases
  - *Weak Scaling* – the problem size increases at the same rate as the number of processors, keeping the amount of work per processor the same
- Strong scaling is generally more useful and more difficult to achieve than weak scaling
- Need to consider intra-node scaling (making use of cache/shared memory within a node) and inter-node scaling (where data has to be sent to and from nodes)
  - Ideally show both!
- Should show scaling relative to minimum no. of cores or nodes that can be run for a given problem size
- Scaling isn't everything
  - Should also profile code and show optimisations that have been done
  - Poor codes (e.g. with lots of unnecessary computation) will scale better! However, not a good use of resources

Strong scaling



Weak scaling



Modelling speedup

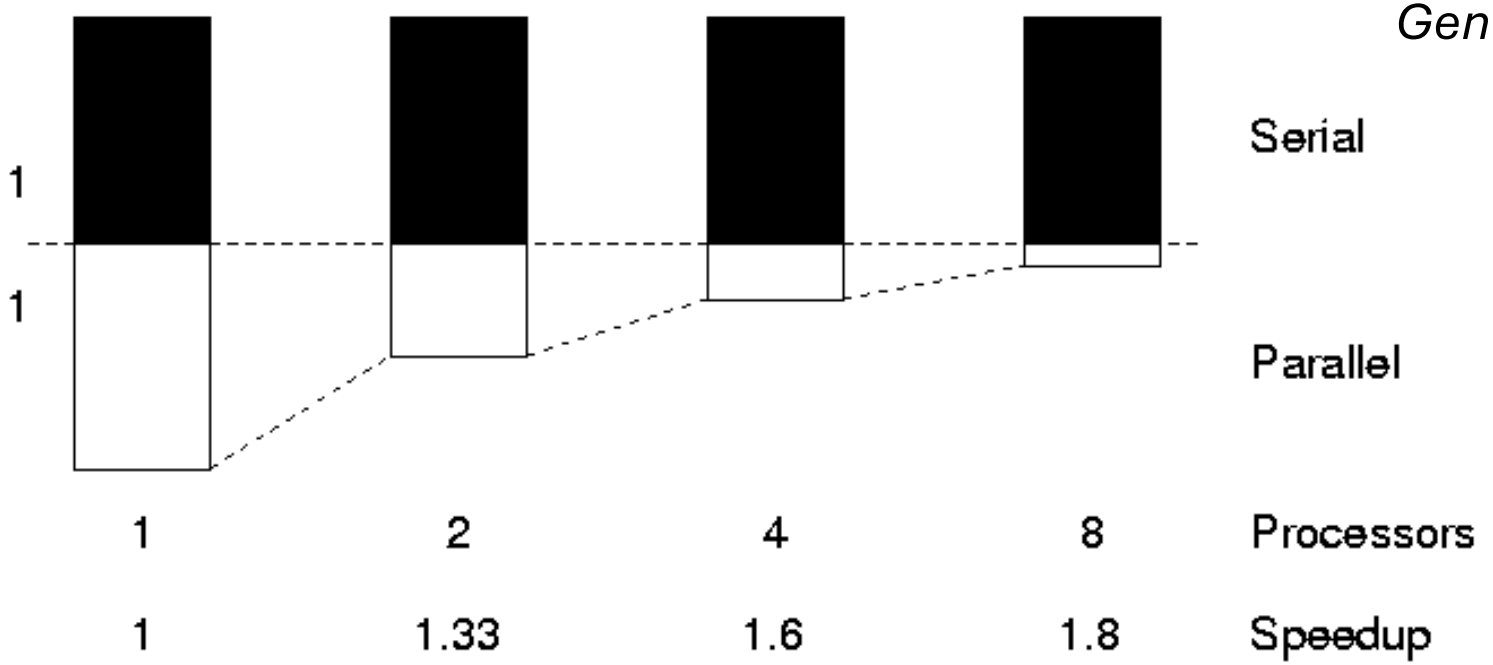
DiRAC

- A typical program has two categories of components
  - inherently serial sections: can't be run in parallel
  - potentially parallel sections
- Classic examples of serial
  - initialisation or file IO (in parallel just do it from a single process)
- In practice, "serial" includes all parallel overheads
  - any operation that does not benefit from parallelisation
  - this includes reduction operations, halo swapping, …

The serial section of code

"The performance improvement to be gained by parallelisation is limited by the proportion of the code which is serial"

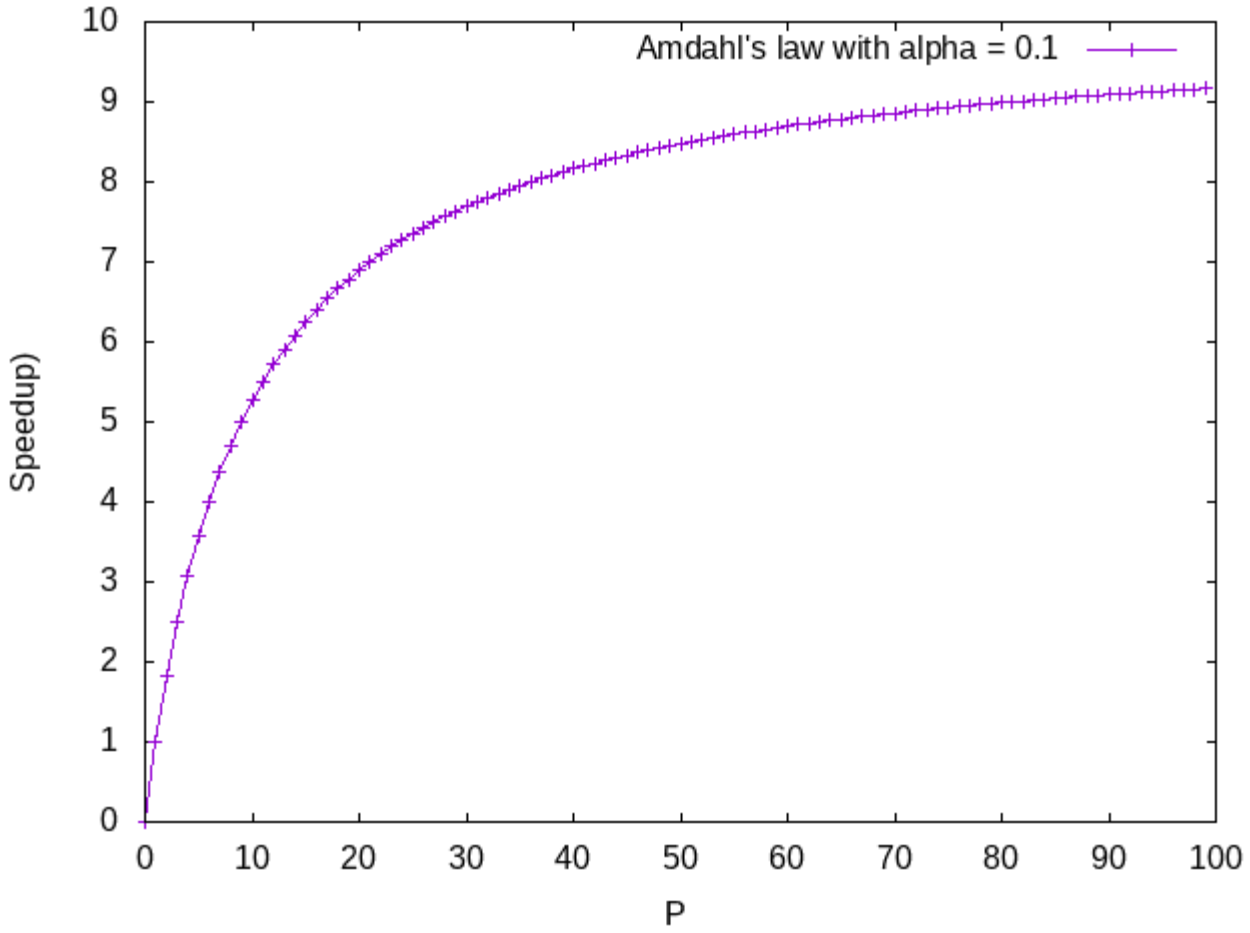
Gene Amdahl, 1967



Amdahl's law

- Assume that a fraction, α, is completely serial
- Parallel runtime $T(N,P) = \alpha T(N,1) + \frac{(1-\alpha)T(N,1)}{P}$
  - assuming parallel part is 100% efficient
- Parallel speedup $S(N,P) = \frac{T(N,1)}{T(N,P)} = \frac{P}{\alpha P + (1-\alpha)}$
- We are fundamentally limited by the serial fraction
  - For $\alpha = 0, S = P$ as expected (i.e. *efficiency* = 100%)
  - Otherwise, speedup limited by $\frac{1}{\alpha}$ for any P
    - For $\alpha = 0.1; \frac{1}{0.1} = 10$; therefore 10 times maximum speed up
    - For $\alpha = 0.1; S(N,10) = 6.4,\ S(N,1024) = 9.9$

Example


Amdahl's law with alpha = 0.1
Speedup)
P
0
1
2
3
4
5
6
7
8
9
10
0
10
20
30
40
50
60
70
80
90
100

Utilising Large Parallel Machines

DiRAC

- Assume parallel part scales with N, serial part constant
  - i.e. parallel part is $\mathcal{O}(N)$, serial is $\mathcal{O}(1)$
  - Time $T(N,P) = T_{serial}(N,P) + T_{parallel}(N,P)$
    $= \alpha T(1,1) + \frac{(1-\alpha)NT(1,1)}{P}$
  - Speedup $S(N,P) = \frac{T(N,1)}{T(N,P)} = \frac{\alpha+(1-\alpha)N}{\alpha+(1-\alpha)\frac{N}{P}}$
- Scale problem size with P, i.e. set $N = P$ (weak scaling)
  - Speedup $S(P,P) = \alpha + (1-\alpha)P$
  - Efficiency $E(P,P) = \frac{\alpha}{P} + (1-\alpha)$

Gustafson's Law

- If you can increase the amount of work done by each process / task, the serial component will not dominate
  - increase the problem size *N* to maintain scaling
  - this can be in terms of
    - increasing the overall problem size
    - adding extra complexity
- For instance, $\alpha = 0.1: S(N, 16) = 6.4,\ S(N, 1024) = 9.9$
  - using strong scaling:

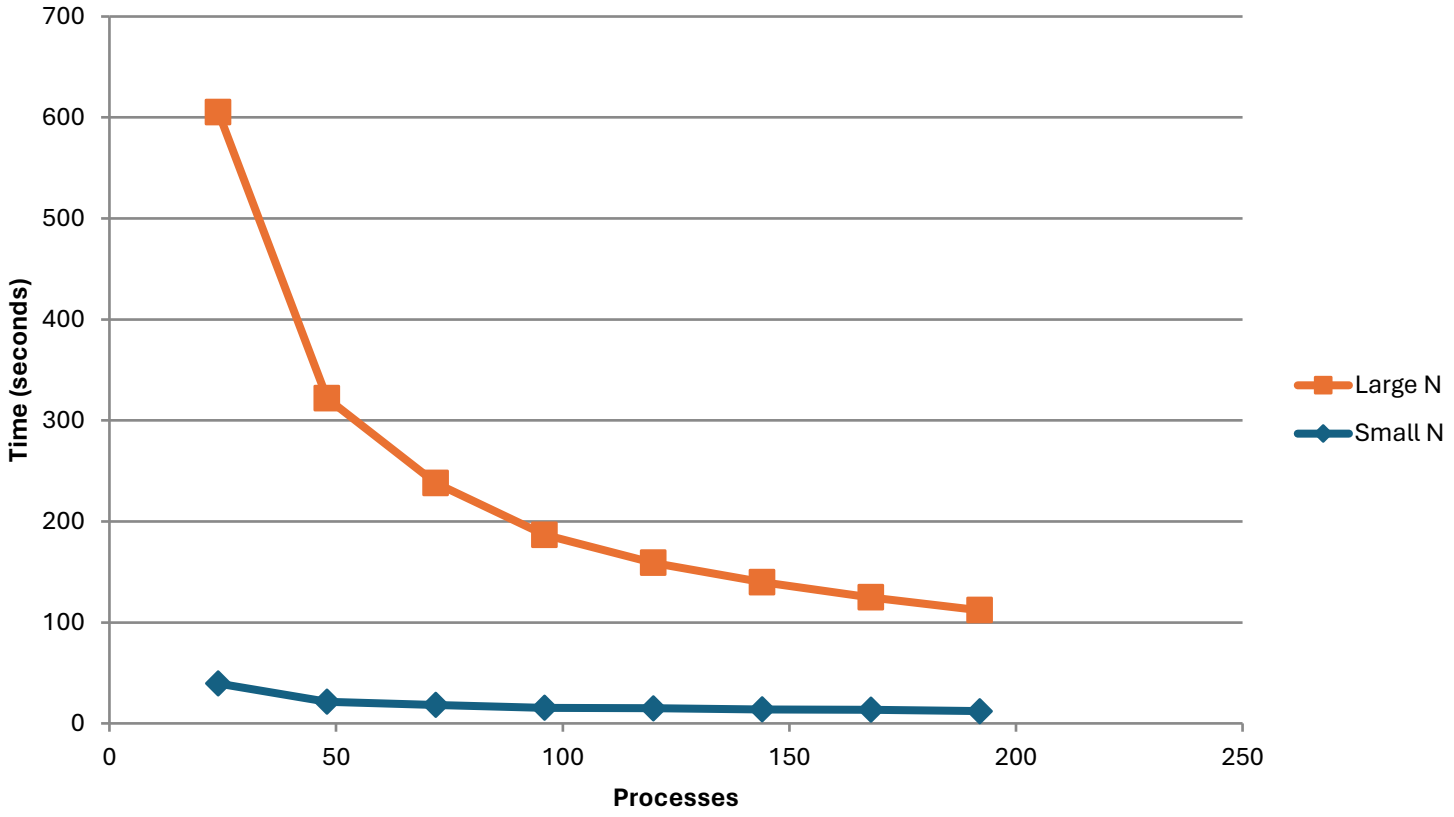
Plotting

- Think carefully whenever you plot data
  - what am I trying to show with the graph?
  - is it easy to interpret?
  - can it be interpreted quantitatively?
- Default plotting options are rarely what you want
  - default colours can be hard to read (e.g. yellow on white)
  - default axis limits may not be sensible
  - …
- Explain (in words) what it is you are trying to show
  - Is this strong or weak scaling data?
  - How does this relate to the jobs you will run?
- Example data
  - MPI traffic model on multiple (24-core) nodes of ARCHER

Performance metrics

DiRAC

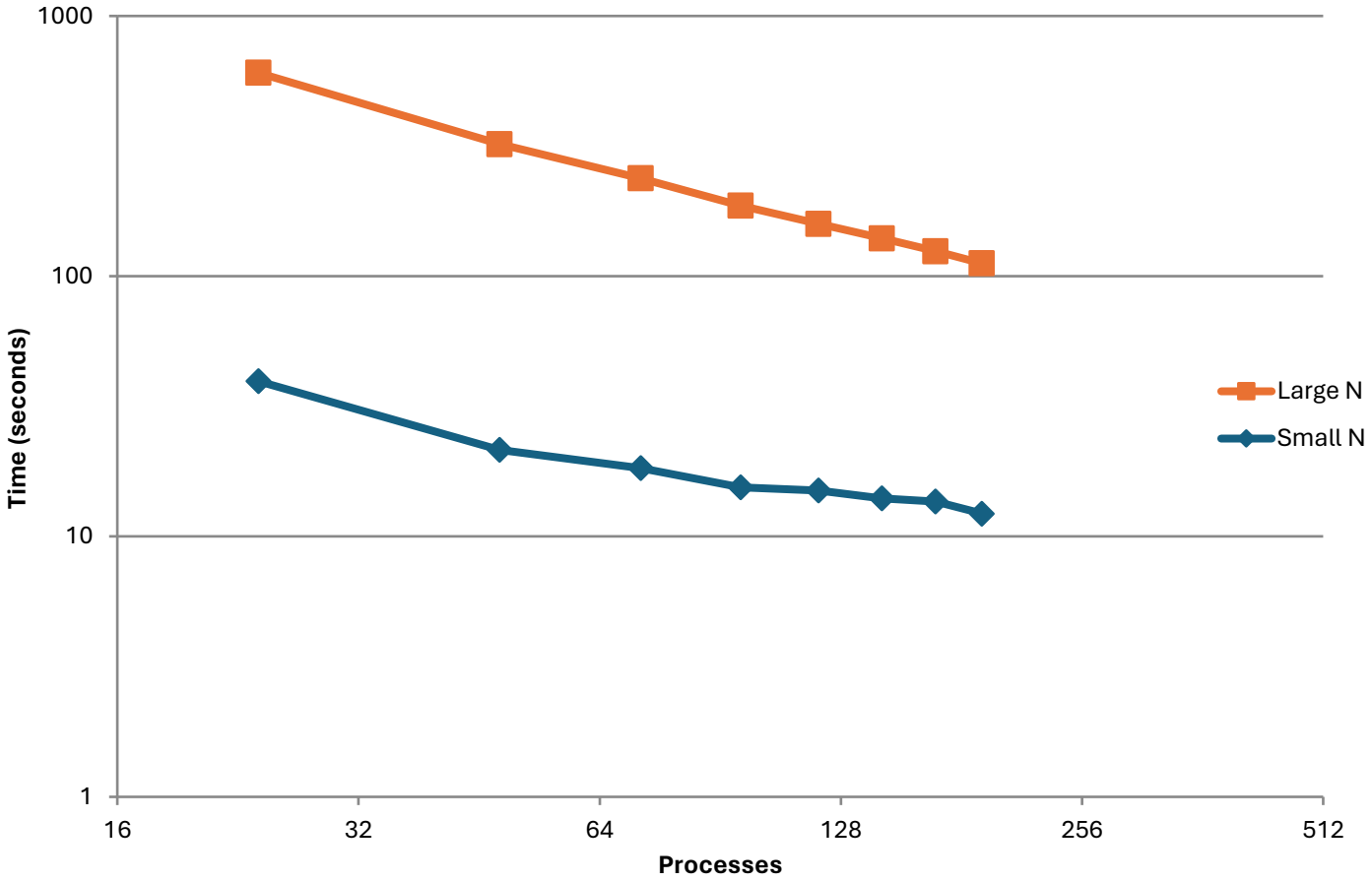
Hard to interpret small N data here



Performance metrics

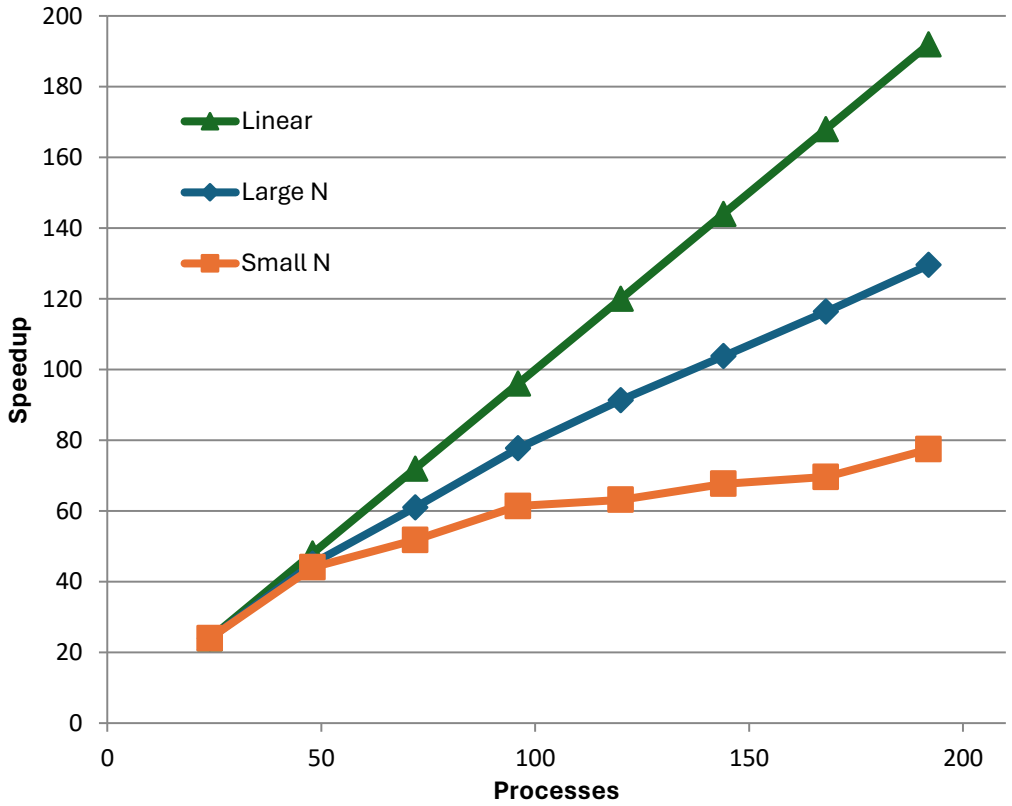
DiRAC

log/log can make trends in data too similar

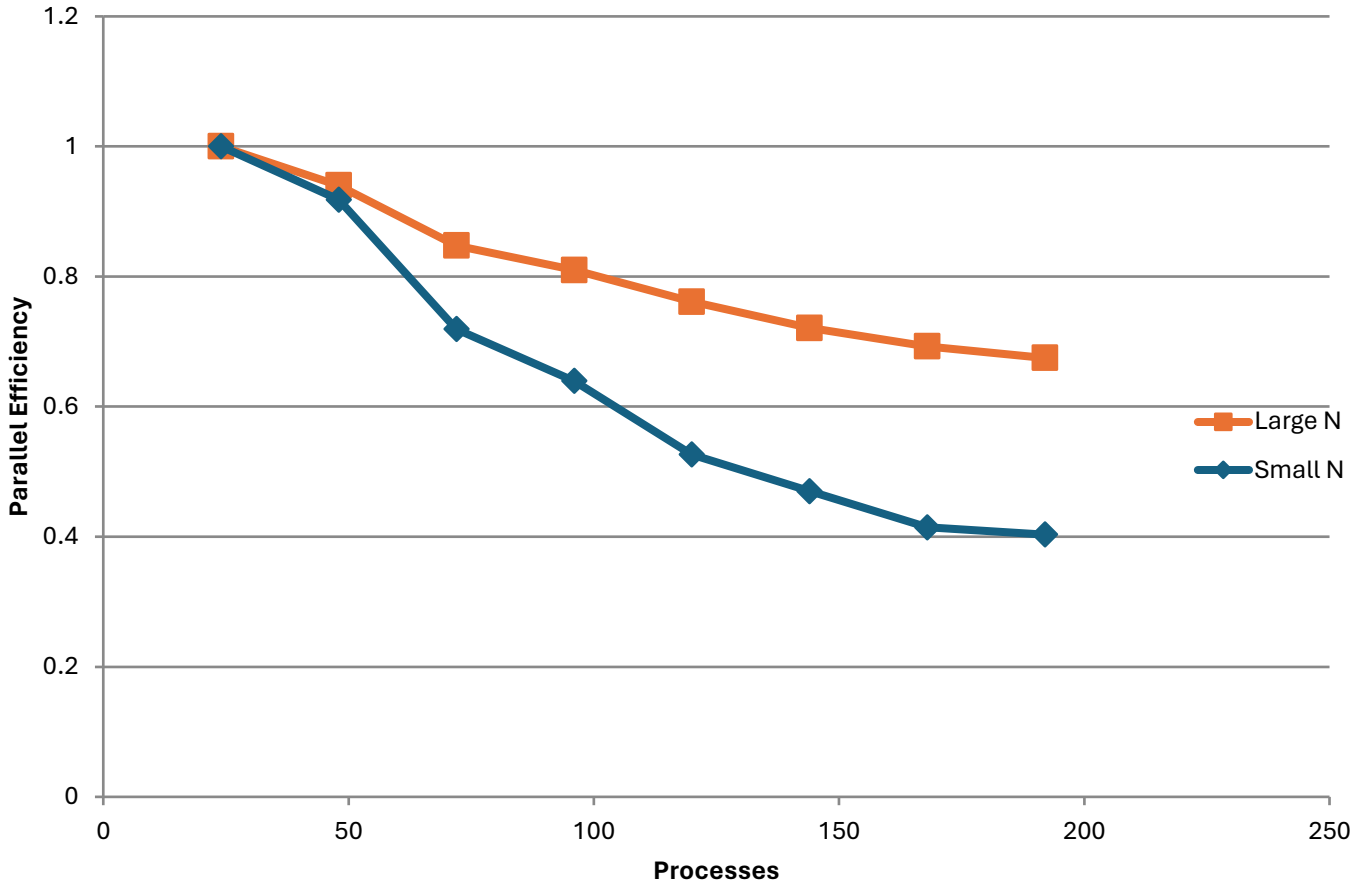


Normalised data easier to compare

- use single-node (24-core) performance as baseline here



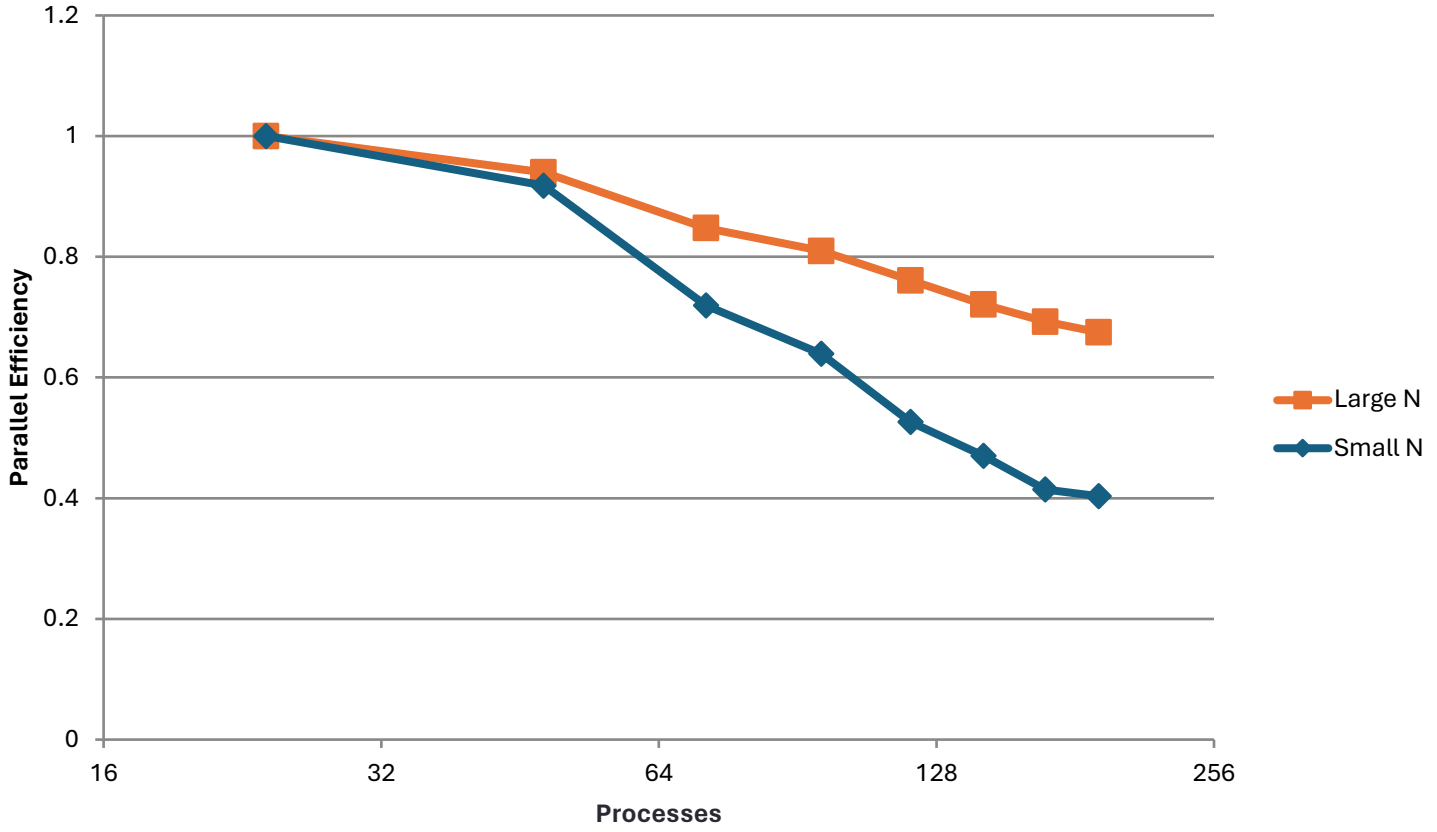
Efficiency plots can be useful too



Performance metrics

DiRAC

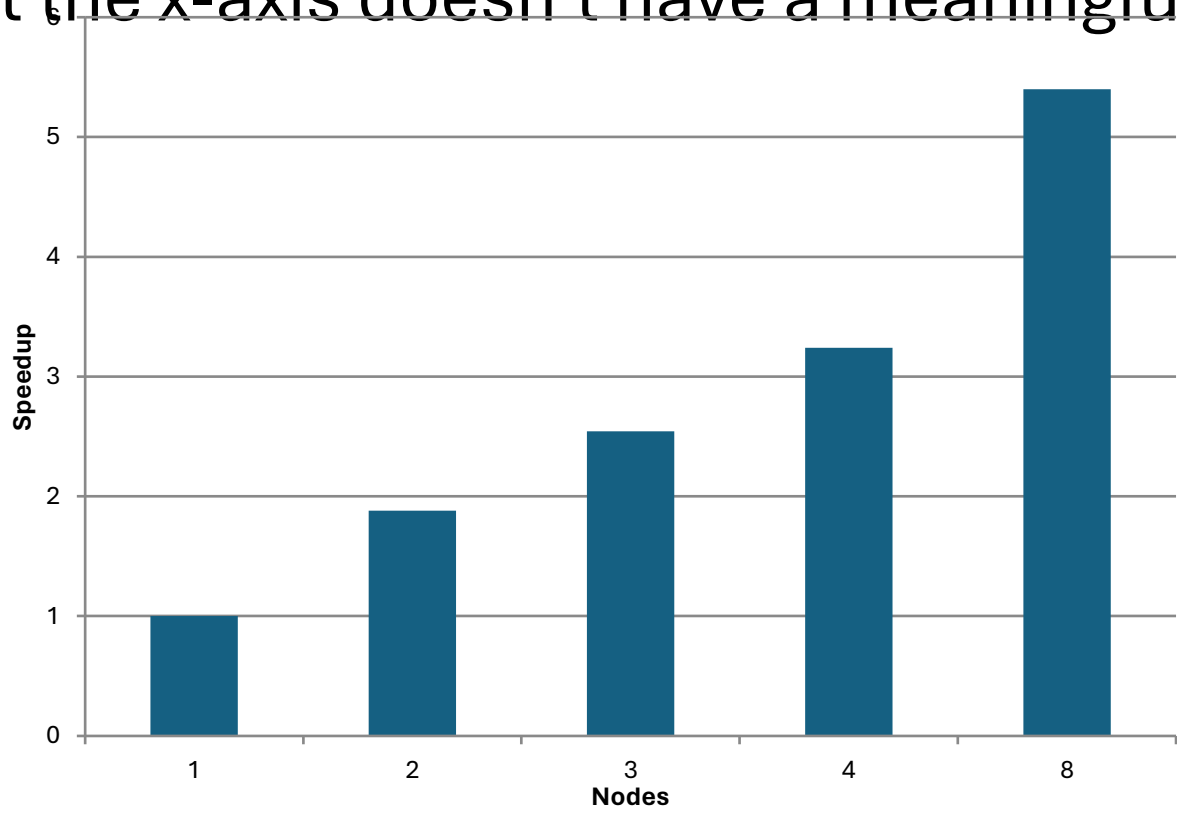
log/lin efficiency if many points at small *P*



DiRAC

Don’t just accept the default options

- In this chart the x-axis doesn’t have a meaningful scale



Proposals – what to ask for?

- In most cases (e.g. RAC) proposals candidates are asked for “how much” of each resource is needed
- Requests usually given in core or CPU hours or GPU hours
  - For an individual job this is the number of cores (or GPUs) used multiplied by the number of hours the job runs for
  - Should then be summed over the entire project
- Typically CPU hours really mean core hours – but you need to check
- Sometimes node hours – but make sure you know which - or your request may be out by factors of 10s or 100s (Cirrus has 288 cores for example)
- There are some moves towards using, eg, kwhrs
  - Early days and may not have relevant data
- Note also sometimes nodes can only be used exclusively
  - If you are only using half the cores of a node (e.g. the code doesn’t scale well within a node, but you need all the memory) then you may be charged for the whole node. You need to check this!
- For DiRAC you need to give a yearly breakdown per machine
  - And a quarterly breakdown for year 1 if you don’t want the time evenly (this cant always be satisfied)

Computational Fluid Dynamics

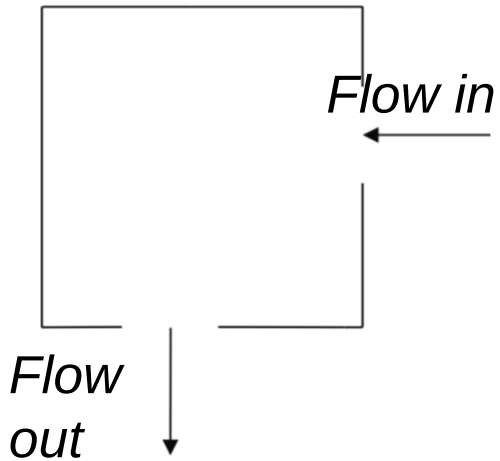
Algorithm, implementation and the problem

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

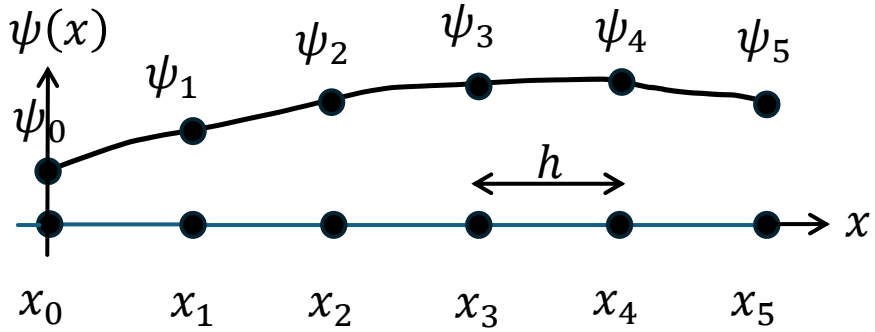
The Problem

- 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

Derivatives and discretisation



x and $\psi(x)$ continuous

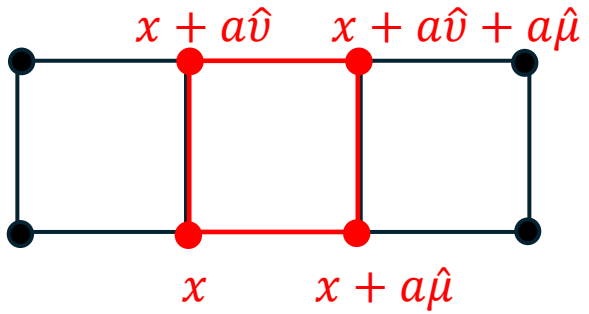
x_i and ψ_i discrete

- Forward derivative: $\frac{d}{dx}\psi(x) \approx \psi_i' = \frac{\psi_{i+1}-\psi_i}{h}$
- Backward derivative : $\frac{d}{dx}\psi(x) = \frac{\psi_i-\psi_{i-1}}{h}$
  - c.f. Taylor series - shows error is $\mathcal{O}(h)$ in both cases
- For second derivative apply forward then backward
  - $\frac{d^2}{dx^2}\psi(x) = \frac{\psi_{i-1}-2\psi_i-\psi_{i+1}}{h^2}$ (i.e. average of nearest neighbours)
- Symmetry about x_i is useful here → symmetric matrices!
  - but symmetry may break when more terms introduced though
- Differentiating at point x_i requires knowledge of value of ψ at points either side of x_i (i.e. ψ_{i+1} and ψ_{i-1})
  - Likewise in more dimensions (i.e. need $\psi_{i+1}, \psi_{i-1}, \psi_{i+1}$ and ψ_{j+1}, etc.) – up down, left, right, in, out, etc.

QCD connection

DiRAC

- QCD, for example, involves covariant derivatives
- Lattice QCD requires these to be discretised
- Forward derivative: $\nabla_\mu \psi(x) := \frac{1}{a}\left(U_\mu(x)\psi(x + a\hat{\mu}) - \psi(x)\right)$
- Backward derivative: $\nabla^*_\mu \psi(x) := \frac{1}{a}\left(\psi(x) - U_\mu(x - a\hat{\mu})^{-1}\psi(x - a\hat{\mu})\right)$
- Still the same idea as Jacobi (nearest neighbours in 4D – look up, down, left right, etc.)



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 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

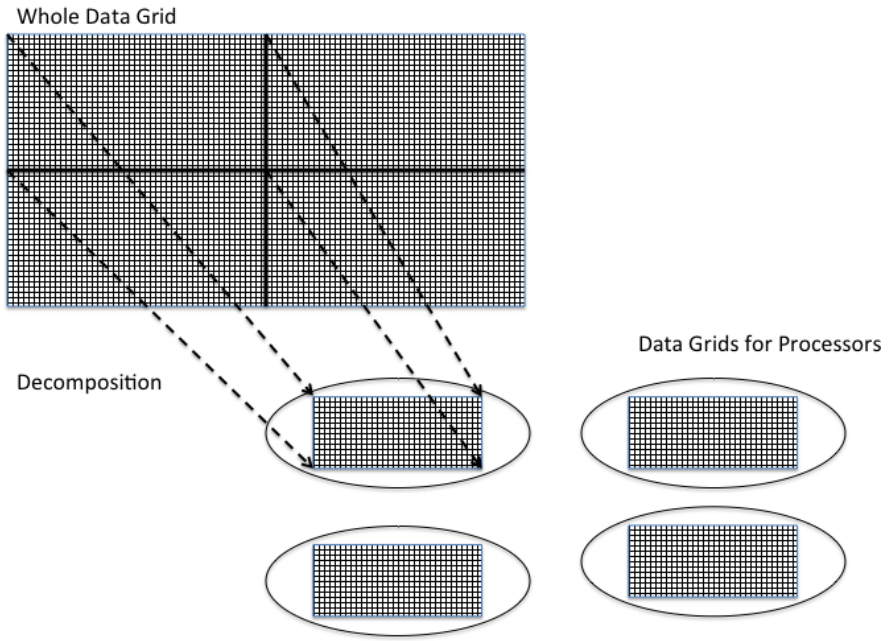
- Finite viscosity gives more realistic flows
  - introduces a new field zeta related to the vorticity
  - equations a bit more complicated but same basic approach
- Terminating the process
  - larger problems require more iterations
  - fixed number of iterations OK for performance measurement but not 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

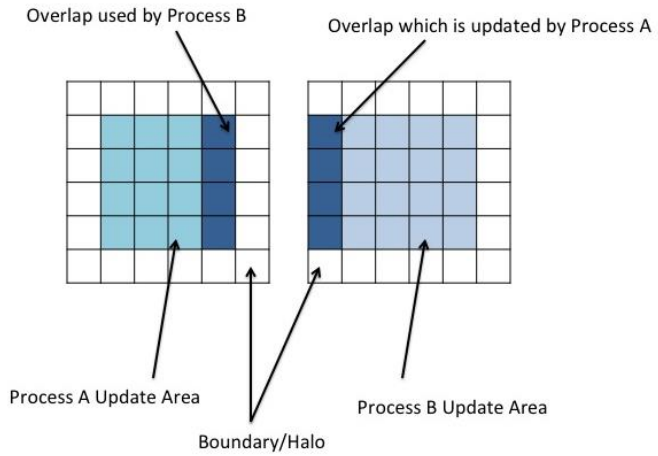
- The algorithm involves calculating the value at each grid point by combining it with the value of its neighbours.
- Same amount of work needed to calculate each grid point – ideal for the geometric decomposition approach.
- Grid is broken up into smaller grids and one is allocated to each process.



Parallel Programming – Halo Swapping

DiRAC

- 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*.



MPI communication

- Data is exchanged via calls to MPI library

Explore the code!

e.g. rank 0 sends “right”

```
MPI_Send(&x[m][1],n,MPI_DOUBLE,rank+1,tag,comm)
```

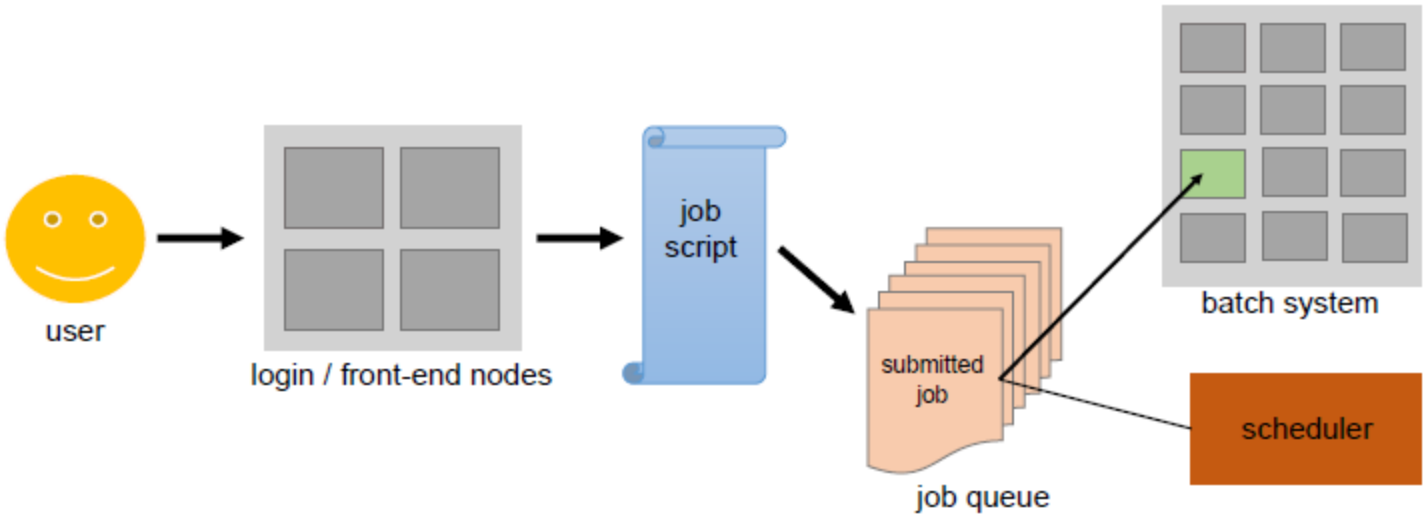
Last rank receives from the left

```
MPI_Recv(&x[0][1],n,MPI_DOUBLE,rank-1,tag,comm,&status);
```

Running a parallel code: Batch systems

- Most HPC systems use some form of “queuing”
- User logs in to login node
- Job then runs on compute nodes, via scheduler
- Batch script required (parallel codes can’t usually be run directly). E.g.

```
>./a.out
Error
>
> cat myscript.slurm
#!/bin/bash
#SBATCH -p scarf
#SBATCH -n 32
#SBATCH -t 30
#SBATCH -o %J.log
#SBATCH -e %J.err
mpirun ./a.out
> sbatch myscript.slurm
Submitted batch job 3133179
```



Source: https://hpc-wiki.info/hpc/Scheduling_Basics

Compiling and running the code

DiRAC

ssh into scarf, e.g.

```
> ssh -i .ssh/privatekey.pem tmp01@ui1.scarf.rl.ac.uk
```

Copy the tar file

```
> cp /work4/training/ccptepp/cfd-par-scarf.tar .
```

Expand tar file:

```
> tar -xvf cfd-par-scarf.tar
```

This should give you the files you need for the exercise

Load the relevant mode to give you a compiler, mpi env to run e.g.

```
> module load intel
```

Change directory to where the code is, e.g.

```
> cd cfd-par/C-MPI/
```

Compile the code using "make" (can ignore warnings for Fortran)

```
> make
```

Edit files as required, e.g.

```
> vi cfd.slurm
```

Submit the job (using the reservation)

```
> sbatch --reservation=ccp-tepp cfd.slurm
```

Check on progress, e.g.

```
> squeue -u $USER
```

Look at output, should contain timing info ☺, e.g.

```
> cat 3146976.log
```

...or error ☹ (some info goes to std error anyway)

```
> cat 3146976.err
```

Practical

DiRAC

- Compile and run the code on Scarf
  - try different numbers of cores - change number of cores using –np flag
    - `mpirun -np 16 ./cfd 32 5000`
  - try to use multiple nodes (i.e. more than 32 cores)
  - for different problem sizes (change scale factor (the 1st arg to cfd)
    - 1 gives 36 x 36
    - Probably need scale factor > 32 to see benefit of multiple cores
  - Can run more than one parallel job in a script. E.g.

```
mpirun -np 16 ./cfd 32 5000
mpirun -np 32 ./cfd 32 5000
```

  - or in a loop

```
for scale_factor in 1 4 8
do
    mpirun ./cfd $scale_factor 5000
done
```

- All this slide (and previous one) available in

`/work4/training/ccptepp/cfd.README`

Details of the exercise

Over to you...