PARALLELISATION & SCALING OF NAMD

Iain Bethune (ibethune@epcc.ed.ac.uk)

with thanks to David Henty and Toni Collis

Outline

- Parallel Programming Models
  - Distributed Memory
  - Shared Memory
- Parallel Decomposition Strategies for Molecular Dynamics
- Parallelisation in NAMD
  - Dynamic Load Balancing
- Measuring performance

Parallel Programming Models

- Why do we need parallelism at all?
- Parallel programming is (even) harder than sequential programming
- Single processors are reaching limitations
  - Clock rate stalled at ~2.5 GHz (due to heat)
  - Full benefits of vectorisation (SIMD) can be hard to realise
  - Chip vendors focused on low-power (for mobile devices)

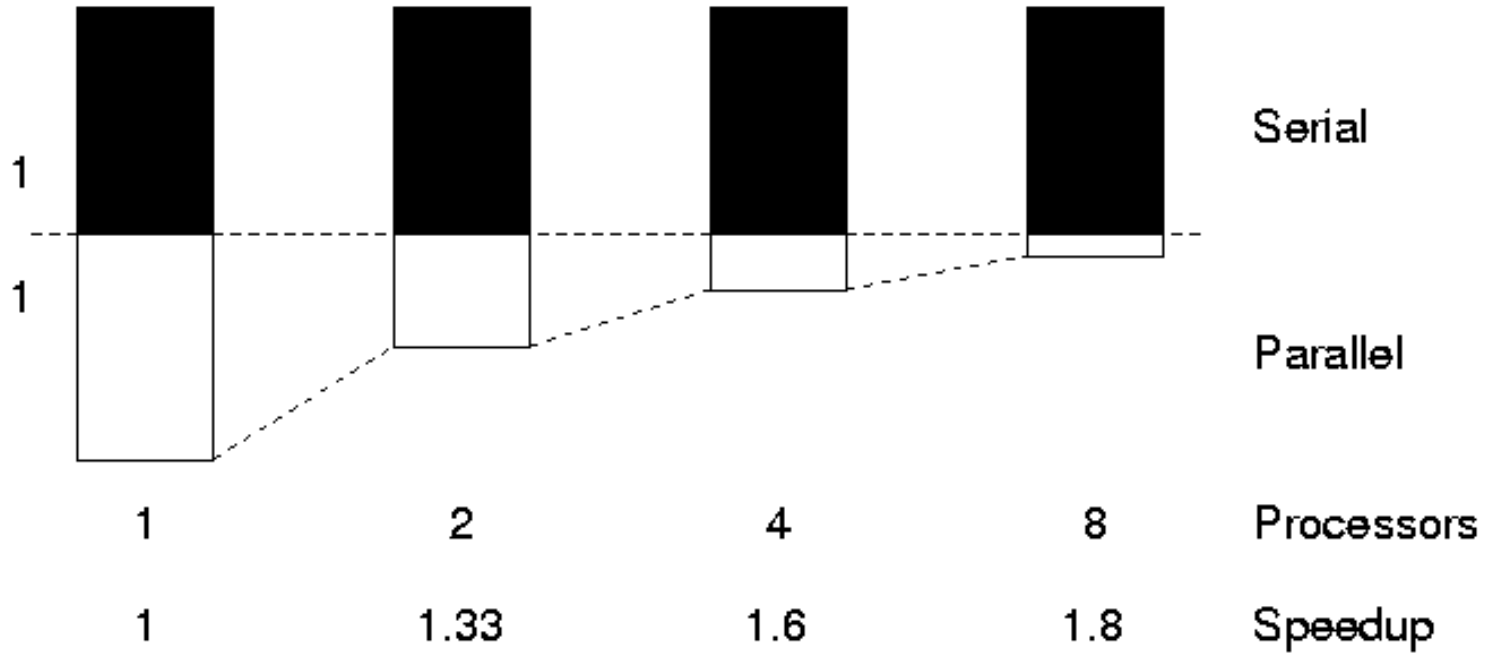
Parallel Programming Models

- But we need *more* speed!
  - Solve problems faster (strong scaling)
  - Solve bigger problems in same time (weak scaling)
  - Tackle new science that emerges at long runtimes / large system size
- Need strategies to split up our computation between different processors
- Ideally our program should run P times faster on P processors - but not in practice!
  - Some parts may be inherently serial (Amdahl's Law)
  - Parallelisation may introduce overheads e.g. communication

Parallel Programming Models

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

Gene Amdahl, 1967

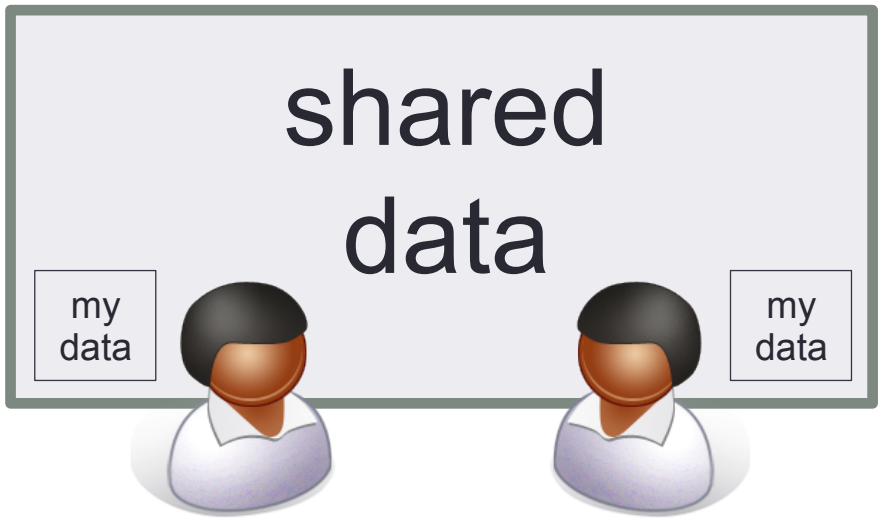


Parallel Programming Models

- Almost all modern CPUs are multi-core
  - 2,4,6… CPU cores, sharing access to a common memory
- This is Shared Memory Parallelism
  - Several processors executing the same program
  - Sharing the same address space i.e. the same variables
  - Each processor runs a single ‘thread’
  - Threads communicate by reading/writing to shared data
- Example programming models include:
  - OpenMP, POSIX threads (pthreads)

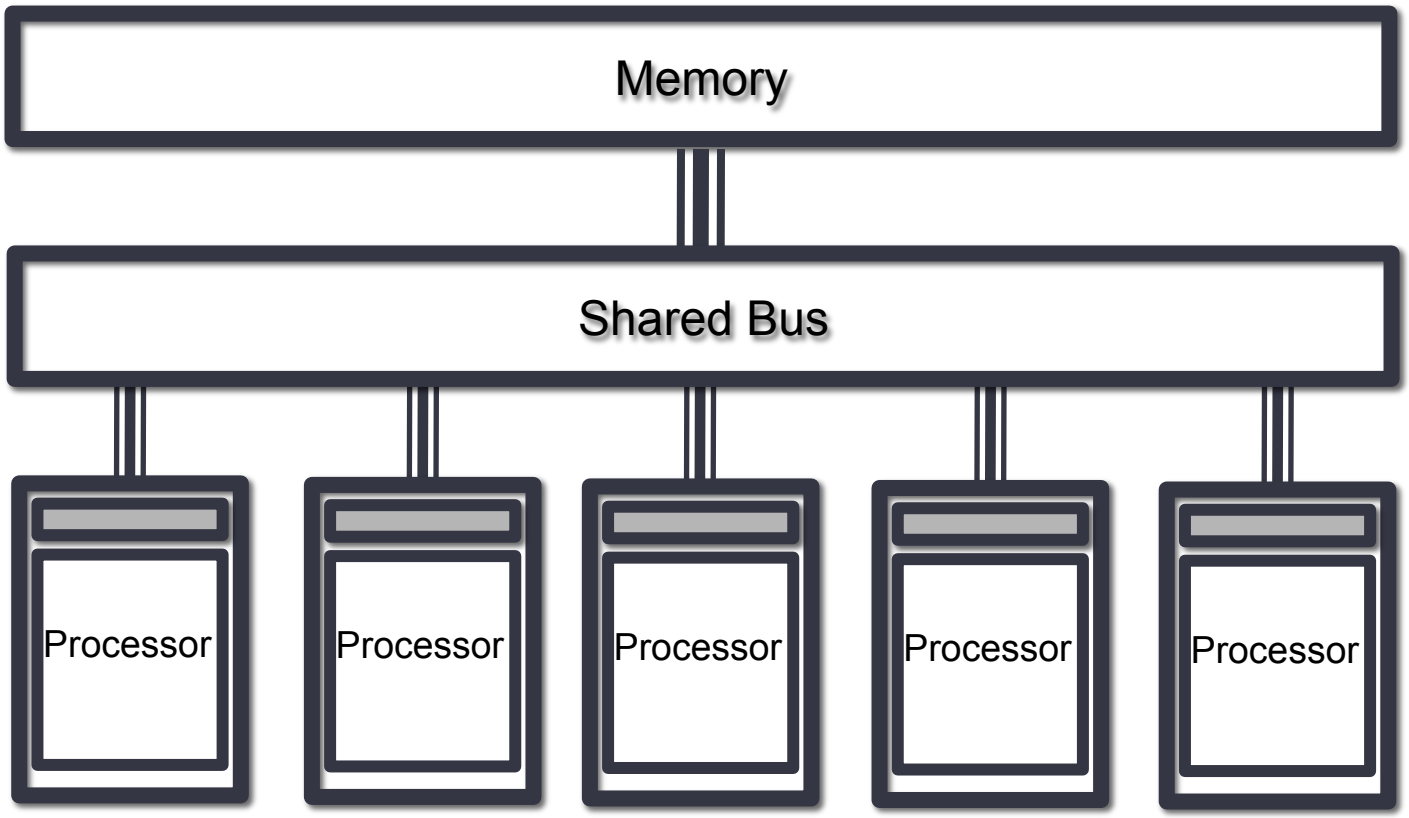
Analogy

- One very large whiteboard in a two-person office
  - the shared memory
- Two people working on the same problem
  - the threads running on different cores attached to the memory
- How do they collaborate?
  - working together
  - but not interfering
- Also need *private* data



Hardware

- Needs support of a shared-memory architecture



Parallel Programming Models

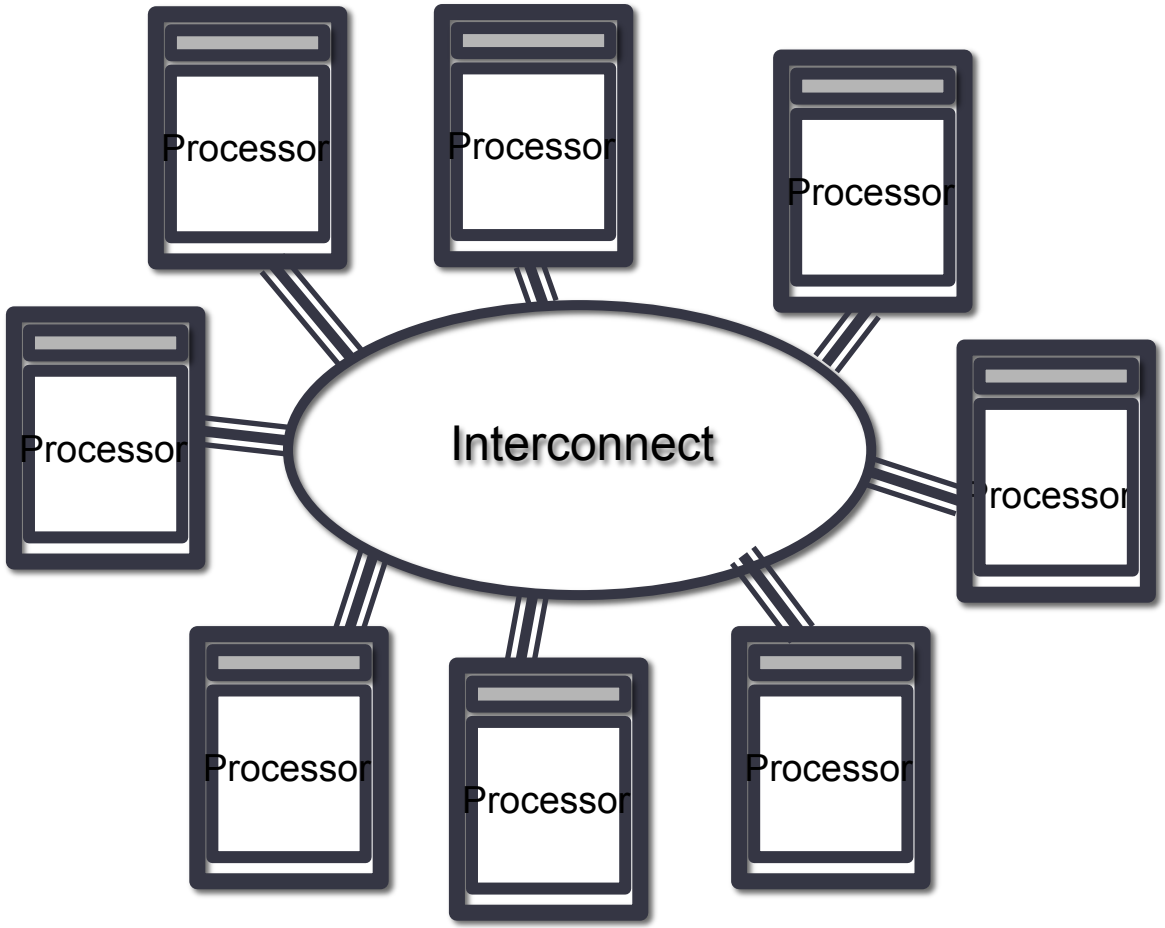
- Most supercomputers are build from 1000s of nodes
  - Each node consists of some CPUs and memory
  - Connected together via a network

- This is Distributed Memory Parallelism
  - Several processors executing (usually) the same program
  - Each processor has it's own address space
  - Each processor runs a single 'process'
  - Threads communicate by passing messages
- Example programming models include:
  - MPI, SHMEM

Analogy

- Two whiteboards in different single-person offices
  - the distributed memory
- Two people working on the same problem
  - the processes on different nodes attached to the interconnect
- How do they collaborate?
  - to work on single problem
- Explicit communication
  - e.g. by telephone
  - no shared data

Hardware



- Natural map to distributed-memory
  - one process per processor-core
  - messages go over the interconnect, between nodes/OS's

Parallel Programming Models

- Some codes support both OpenMP and MPI
  - Use OpenMP for desktop PCs with multi-cores *or*
  - MPI for supercomputers
  - Maybe also support for Accelerators (GPUs)

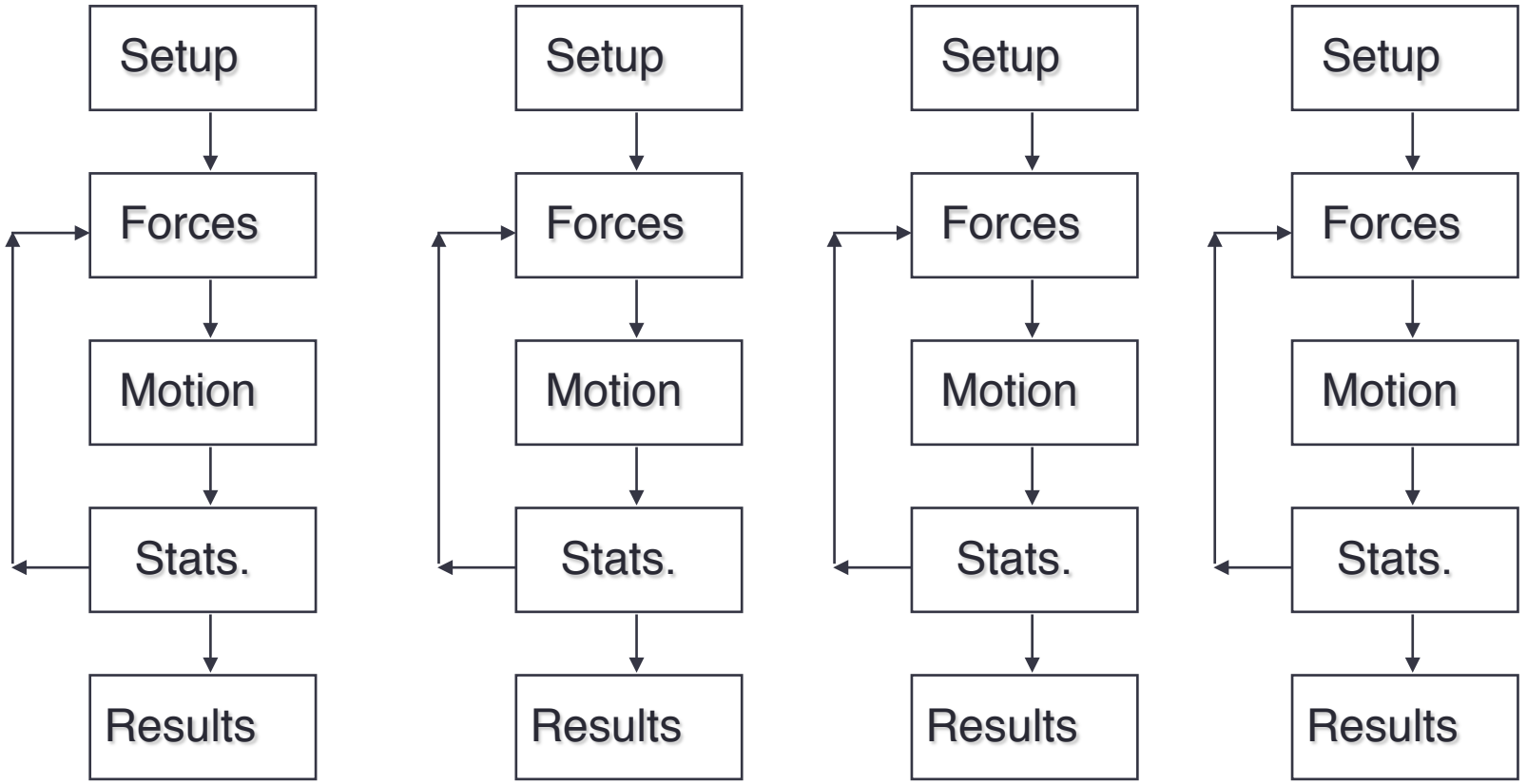
- May also combine MPI *and* OpenMP
  - Called hybrid or mixed-mode parallelism
  - Use shared memory within a node (with several processors)
  - Use message passing between nodes
  - Usually only useful for scaling to 10,000s of cores!

Parallel Decompositions for MD

- Given P processes, how to we split up the work?

- Goals:
  - Achieve good load balance
    - Each processor takes an equal share of the work / time
    - Poor load balance limits scaling (similar to Amdahl's Law)
  - Reduce communication
    - Especially global communication e.g. Broadcast, gather
  - Asynchronous communication
    - If possible, do communication while other work is going on

Parallel MD - Task farm



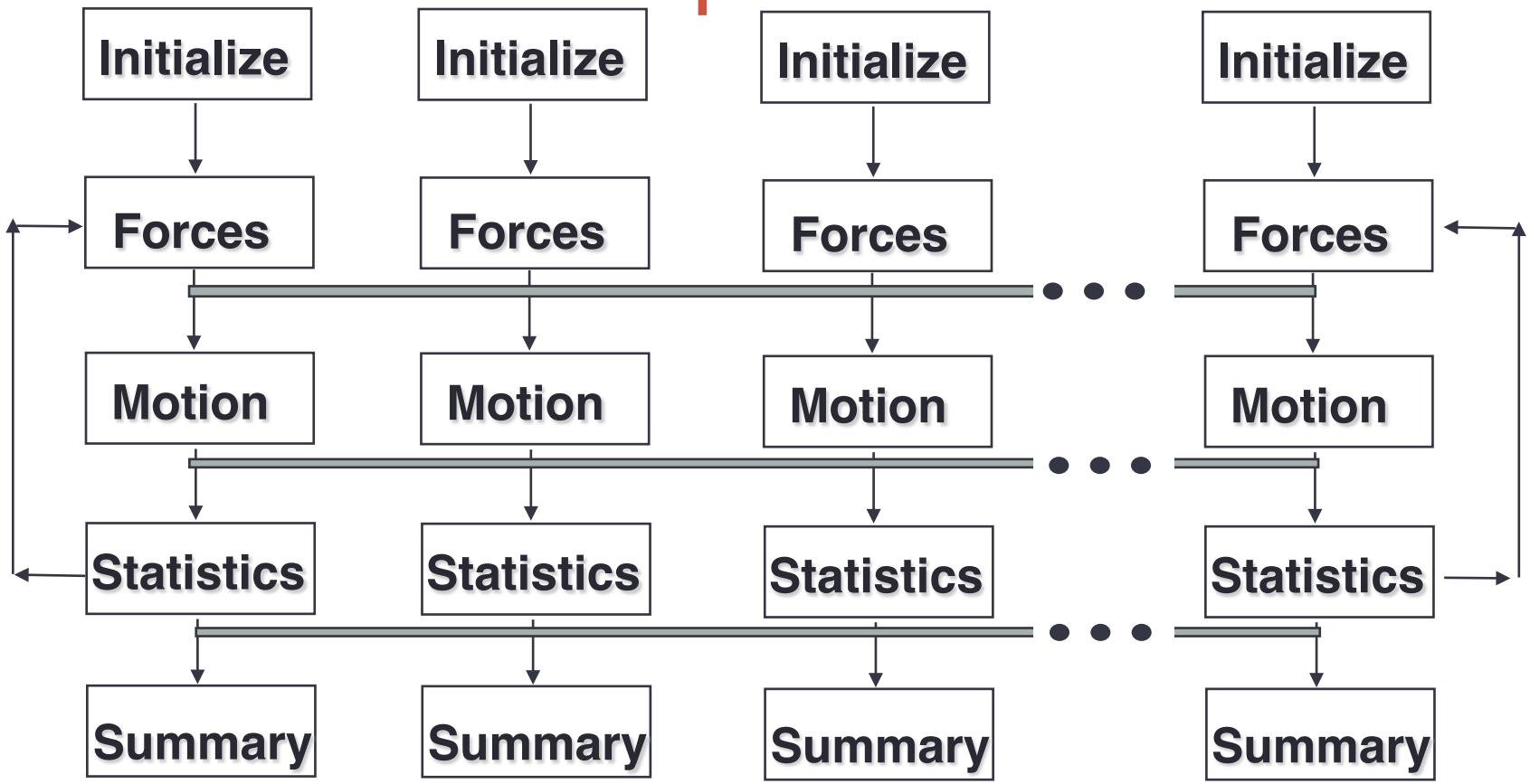
Proc 0 **Proc 1** **Proc 2** **Proc 3**

Parallel MD - Task farm

- Advantages:
  - Simple to implement – no communications
  - Excellent load balance (assuming all systems are the same size)
  - ‘Embarassingly parallel’ – perfect scaling

- Disadvantages:
  - Only for replica / multiple walker sampling
  - Cannot reduce runtime per MD step – limit to short MD timescales

Parallel MD – Replicated Data



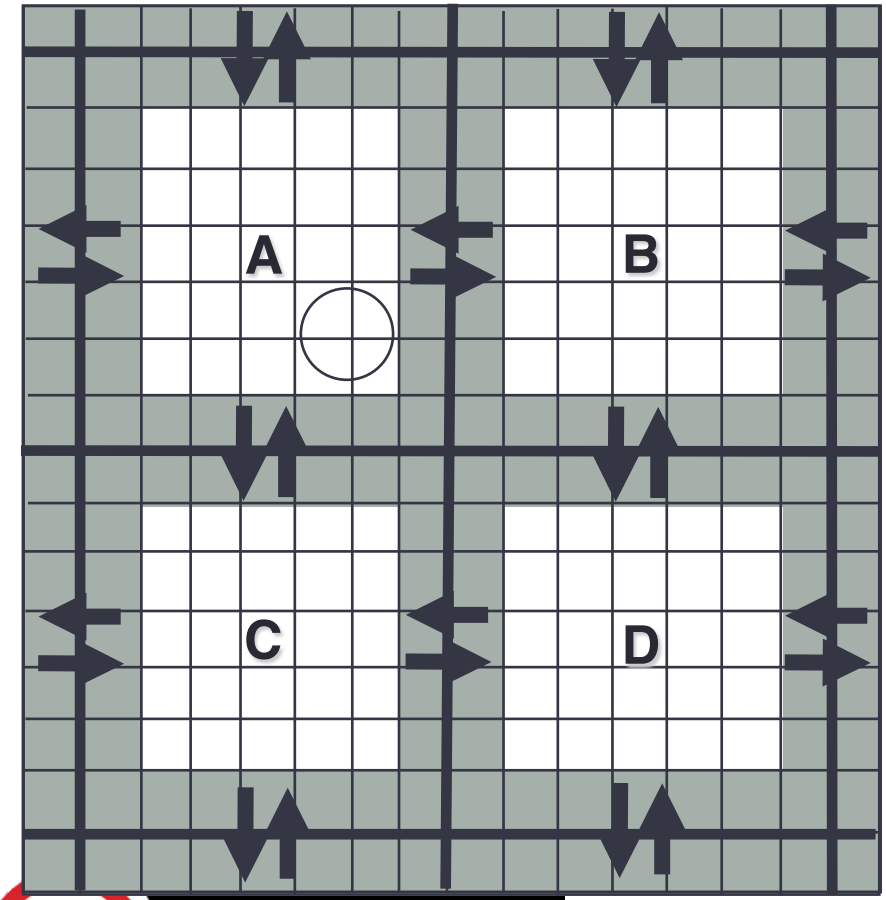
Parallel MD – Replicated Data

- Advantages:
  - Relatively simple to implement
  - Possible to achieve good load balance
    - Can decompose over particles, terms in the force field …
  - Works well with complex force-fields

- Disadvantages:
  - Global communication overhead
  - Leads to limited scalability
  - Requires large amount of memory in total
    - Every process stores all the particles

Parallel MD - Domain decomposition

2D Example



- Short range potential cut off ($r_{cut} << L_{cell}$)
- Spatial decomposition of atoms into domains
- Map domains onto processors
- Use *link cells* in each domain
- Pass border link cells to adjacent processors
- Calculate forces, solve equations of motion
- Re-allocate atoms leaving domains

archer

epcc

THE UNIVERSITY OF EDINBURGH

Parallel MD – Domain Decomposition

- Advantages:
  - Communication is mainly local (between neighbouring processes)
  - Possible to achieve good load balance
    - If system is isotropic
  - Memory is distributed over all processes
    - Allows large scaling
  - Enables bigger systems than can be handled by a single CPU (millions of atoms)

- Disadvantages:
  - Larger cut-offs lead to more communication
  - Implementation is more complex

Parallelisation in NAMD

- Modified version of domain decomposition
- Split up space into ‘patches’
  - nPatches >> nCPUs
- Initial static load balance
  - Assign patches to CPUs so each has roughly same number of atoms
  - Keep neighbouring patches on nearby CPUs (minimise communication

Parallelisation in NAMD

- Workload is modelled as follows:
  - Local force computation ~ Na_p^2
    - All pairs of local atoms
  - Force computations between neighbouring patches ~ $w * Na_a * Na_b$
    - Weighting *w* depends on if patches share a corner, edge or face
  - Forces between patches are assigned to ‘compute objects’
    - May be migrated freely between processors later

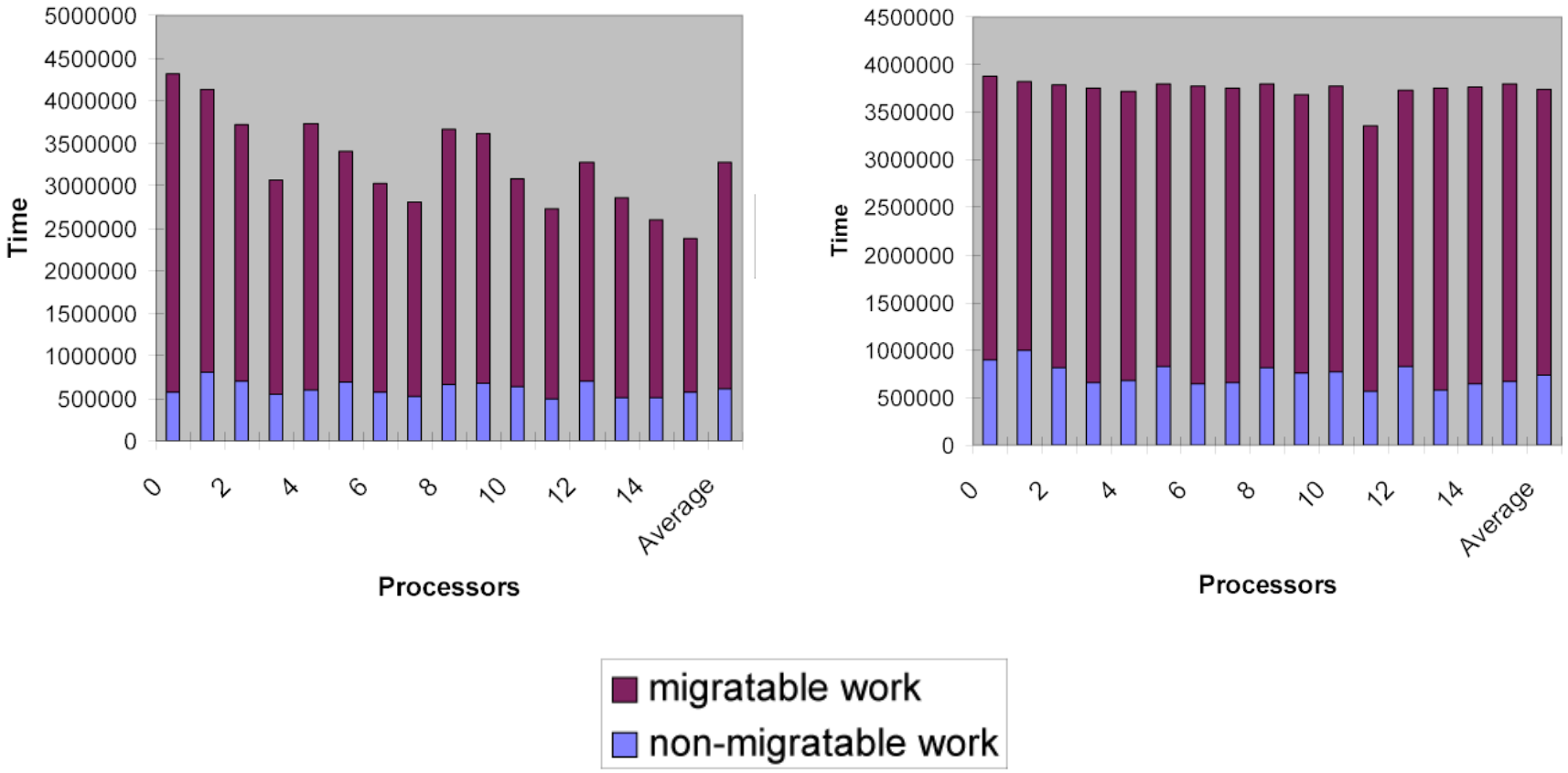
- Then at runtime, use dynamic load balancing to optimise the domain decomposition
  - Accounts for costs not covered by the model
  - Cope with changing system geometry during MD

Parallelisation in NAMD

- Workload metrics are recorded as follows:
  - Background load (non migratable work)
  - Idle time
  - Migratable compute objects and their associated compute load
  - The patches that compute objects depend upon
  - The home processor of each patch
  - The proxy patches required by each processor

- Load balancing heuristic
  - Move most expensive migratable object (compute objects) to least loaded processor, taking into account possible communication increases
  - Details in Kalé *et al*, LNCS 1457, 1998

Parallelisation in NAMD



Measuring Performance

- Basic measure – wallclock time
  - How long did my calculation take from start to finish?
  - Depends on the number of processors!
  - Lower is better

- Application-specific measures
  - For MD, simulation time per wallclock time
  - e.g. ns / day
  - Using how many processors?
  - Higher is better

Measuring Performance

- Speed up
  - typically S(N,P) < P

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

- Parallel efficiency
  - typically E(N,P) < 1

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

Where N is the size of the problem and P the number of processors

- Usually, consider E > 70% to be ‘good’ scaling

Measuring Performance

- How to choose the number of CPUs for your simulation
- Rely on *relevant* benchmark data
  - How many atoms, what force-field (cut-off, PME) ?
  - Some examples provided at http://www.ks.uiuc.edu/Research/namd/performance.html
- No substitute for testing with your own system

Measuring Performance

- Important factors for a benchmark calculation
- Use ‘production settings’
  - I/O turned on, chosen forcefield settings
  - Benchmark should closely reflect performance of real simulation
- Reduce the number of MD steps
  - Long enough to ignore the effects of startup overheads
    - In NAMD after a few 100 steps the dynamic load balancer starts working
  - Short enough to not waste CPU time
    - Aim for a few minutes

Summary

- Modern HPC systems support both shared and distributed memory parallelism
  - Codes have adapted to exploit this
- Many ways to parallelise MD
  - All are a compromise between complexity and performance
  - ‘Best’ method depends on the system e.g. in vacuo, solvated, solid state
- Always run scaling tests before spending large amounts of CPU time for long MD runs