



Development of Mixed Mode MPI / OpenMP Applications

Lorna Smith and Mark Bull

EPCC, The University of Edinburgh
l.smith@epcc.ed.ac.uk or m.bull@epcc.ed.ac.uk

<http://www.epcc.ed.ac.uk/>

- ▶ Introduction
 - ▶ Related work
 - ▶ Programming model characteristics
 - ▶ Implementation
 - ▶ Benefits
 - ▶ Case study
 - ▶ Real application
 - ▶ Conclusions
-

▶ Funding

- UK High End Computing



▶ Motivation

- performance on SMPs
 - performance on clustered SMPs
 - exploit the characteristics of both MPI and OpenMP
-

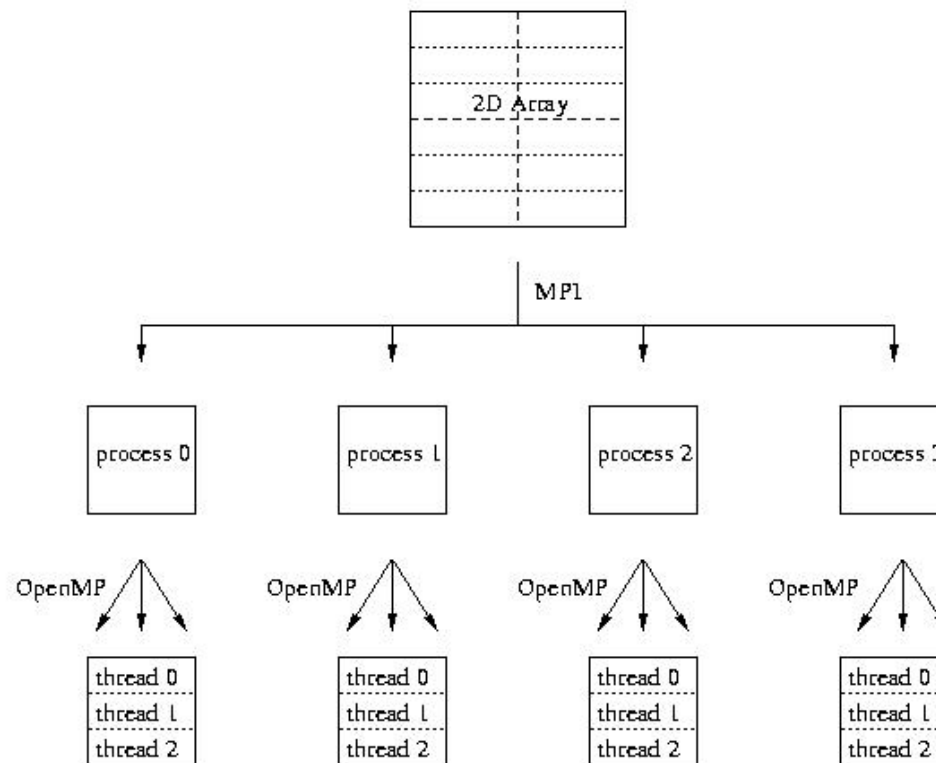
▶ MPI

- characteristics
 - distributed memory model
 - explicit control parallelism
- advantages:
 - data placement problems rarely observed
 - implicit synchronisation with subroutine calls
 - portable to distributed and shared memory machines
- disadvantages:
 - development and debugging: time consuming
 - communications overhead
 - large code granularity often required to minimise latency
 - global operations can be expensive

▶ OpenMP

- characteristics:
 - shared memory model
 - implicit communication
 - advantages
 - relatively easy to implement
 - better use of the shared memory environment (in theory)
 - both course and fine grain parallelism are effective
 - disadvantages
 - limited to shared memory machines
 - data placement may cause problems
-

- ▶ No guarantee of thread safe MPI implementation
 - MPI calls within threads sequential code regions
- ▶ Hierarchical model



- ▶ Poor scaling MPI codes
 - Load balance problems
 - Fine grain parallelism problems
 - ▶ Replicated data
 - ▶ Ease of implementation
 - ▶ Restricted MPI processes applications
 - ▶ Poorly optimised intra-node MPI
 - ▶ Poor scaling of the MPI implementation
 - ▶ Computation power balancing
 - Tafti et al
-

▶ Aim

- to gain performance improvement over a pure MPI application
- allow for the most efficient use of a clustered SMP

▶ The code

- Game of Life code
 - grid-based cellular automaton
 - 2D grid of cells, two states: alive / dead
 - Code structure
 - 1 initialise the 2D cell
 - 2 carry out boundary swaps (for periodic boundary conditions)
 - 3 loop over the 2D grid, to determine the no of alive neighbours
 - 4 up-date the 2D grid, based on the no of alive neighbours
 - 5 iterate steps 3-4 for the required no of iterations
 - 6 write out the final 2D grid
-

▶ MPI

- geometric domain decomposition of 2D grid
 - each process responsible for up-dating the elements of its sub-array
 - halo swaps carried out for each iteration
 - on completion, all data sent back to the master process
 - numerous MPI calls, considerable code modification (~100 extra lines of code)
-

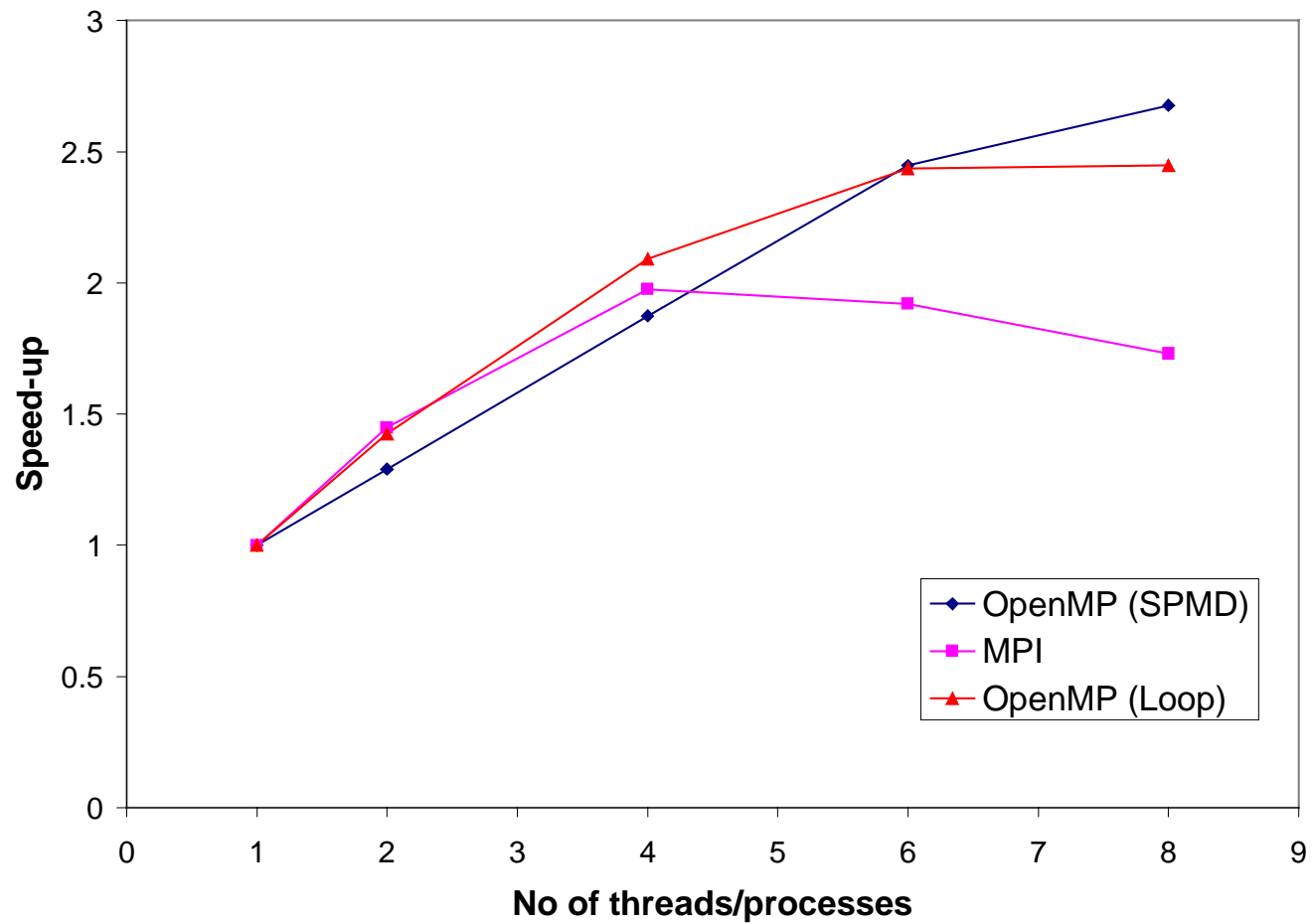
▶ OpenMP

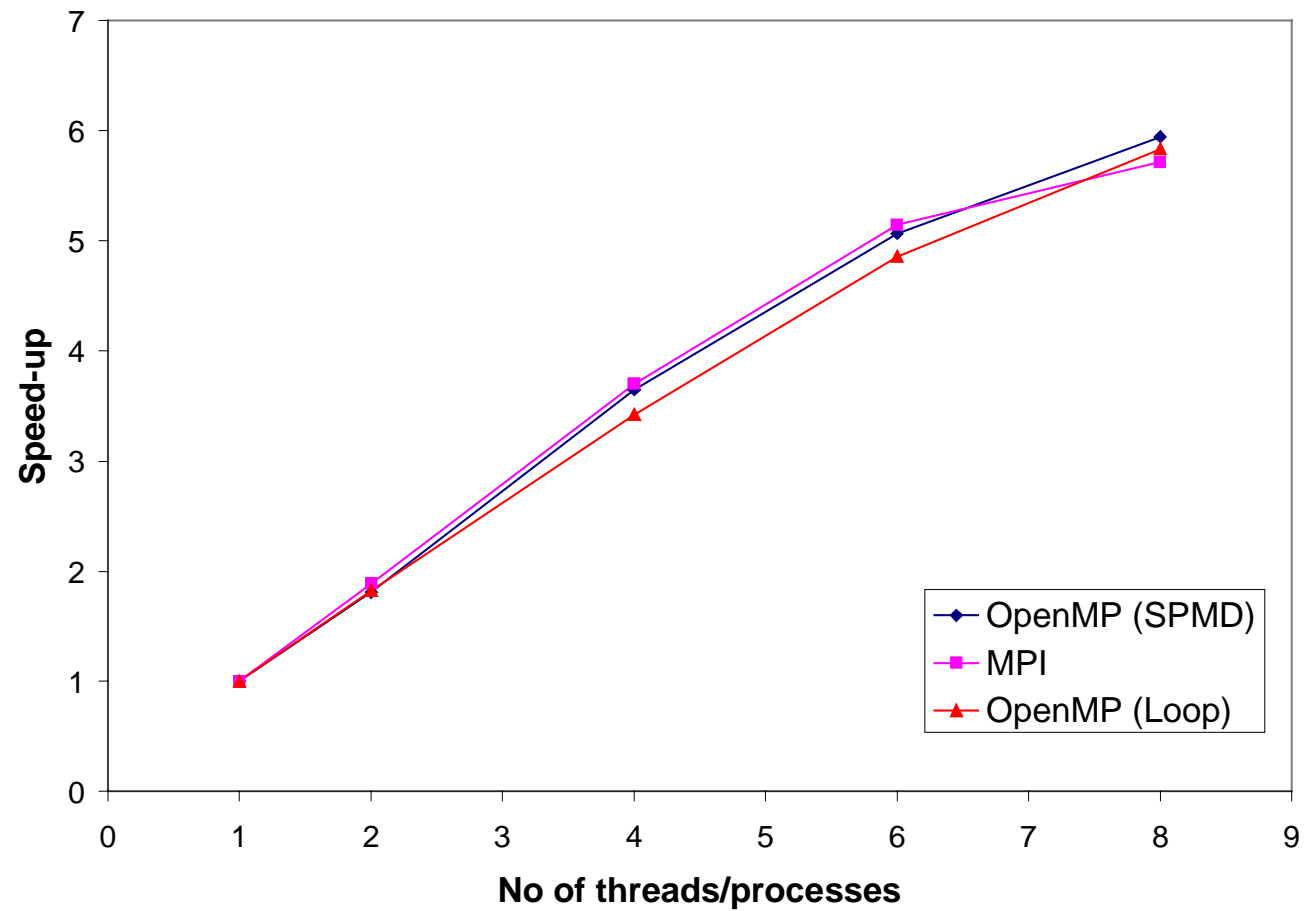
– Loop based

- PARALLEL DO directives around the 3 computationally intense loops
- minimal code changes (~15 lines of code)

– SPMD

- domain decomposition strategy
 - PARALLEL region
 - extra index to the main array of cells, based on thread number
 - each thread responsible for its own array section based on thread index
 - halo swaps carried out between different array sections
 - only nearest neighbour synchronisation required
 - halo swaps involve direct read/writes
-





▶ Performance

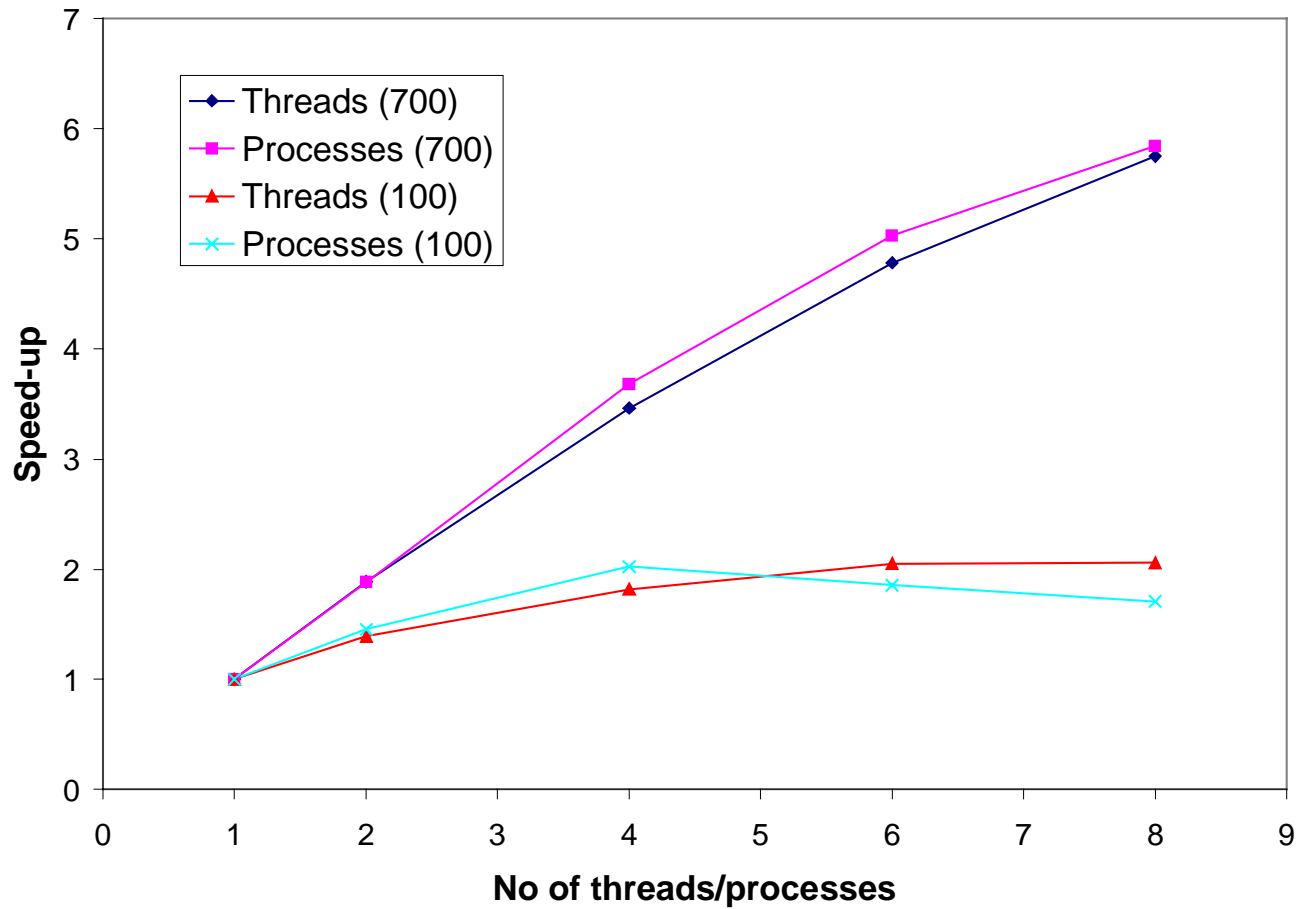
- Better performance for both OpenMP codes on both problem sizes
- Performance difference more extreme on the finer grain problem
- SPMD OpenMP code gives best performance

▶ Scaling

- MPI: communication involved in halo swaps
- Loop based OpenMP: all threads synchronise for each iteration
- SPMD OpenMP: benefits from minimum, nearest neighbour synchronisation (like MPI)

▶ Mixed mode loop based code

- MPI domain decomposition as before
 - OpenMP PARALLEL DO directives placed around relevant loops
 - OpenMP parallelisation occurs beneath MPI parallelisation
 - Work distributed geometrically between MPI processes, further parallelised using OpenMP loop based parallelisation
-

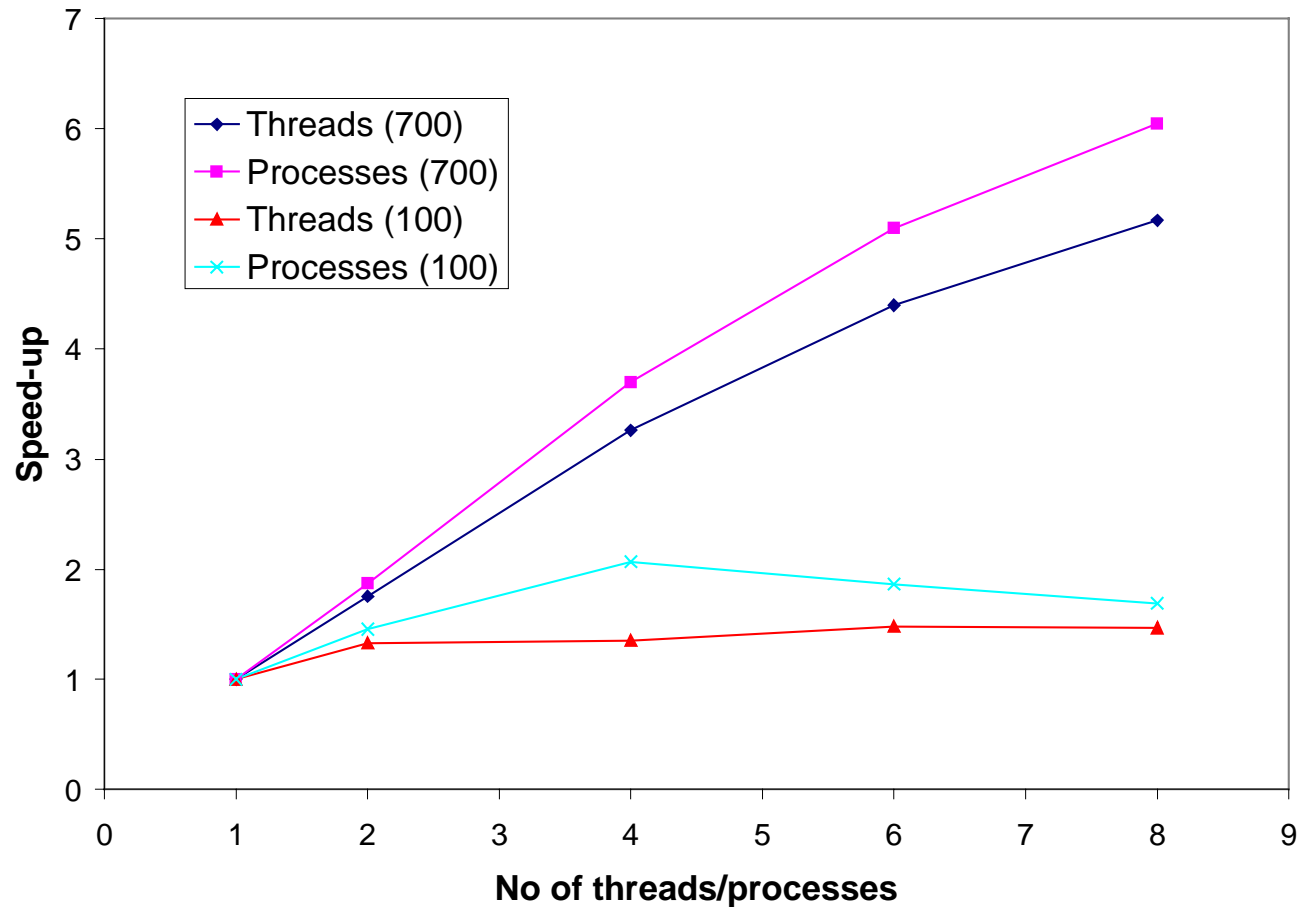


▶ Loop based

- Similar scaling for MPI processes + OpenMP threads
 - OpenMP scaling slightly better for the smaller problem size
 - MPI scaling slightly better for the larger problem size
 - MPI scaling: extra communication due to halo swaps
 - OpenMP scaling: additional synchronisation
- No significant benefit over pure MPI code

▶ 2D code

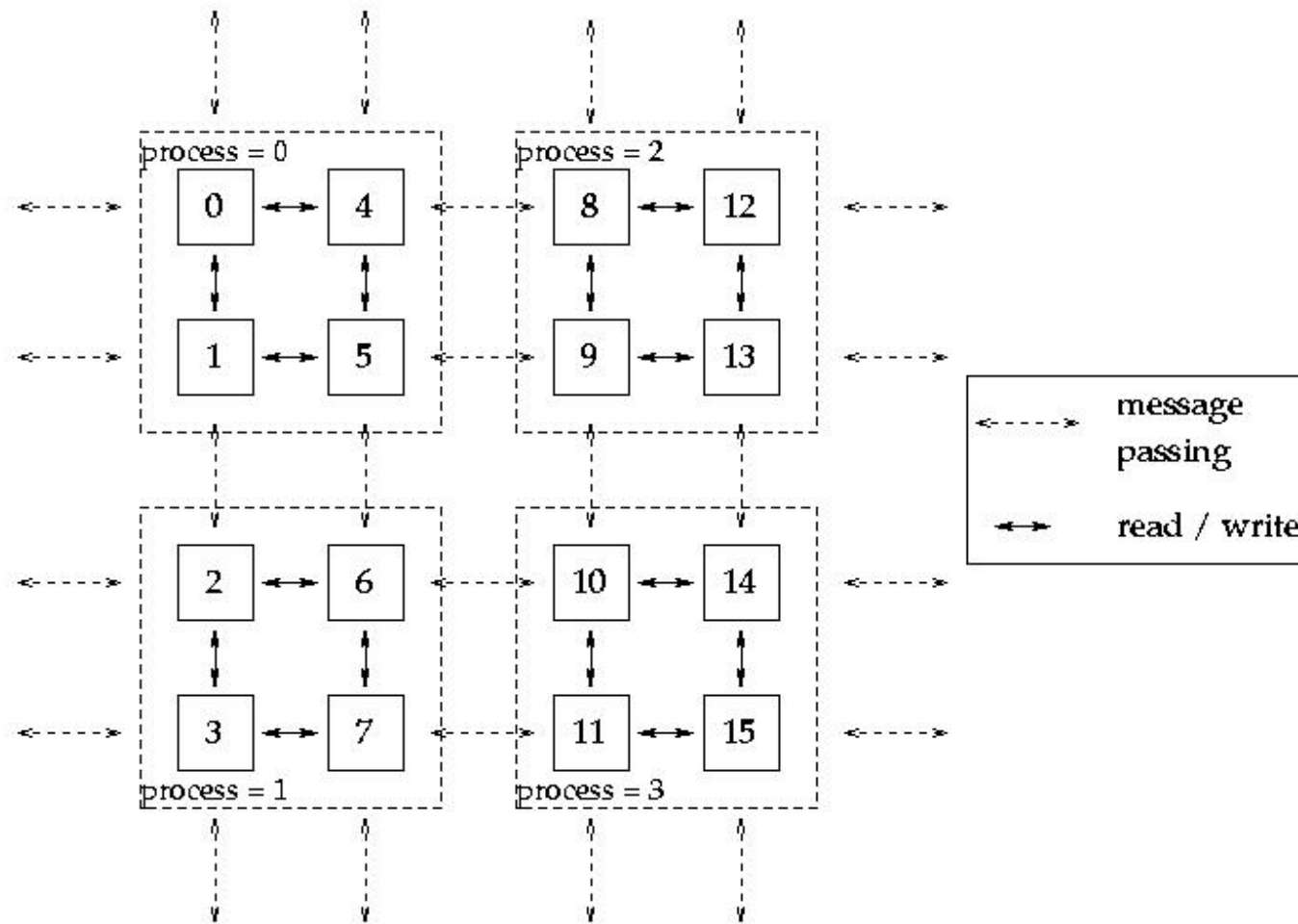
- MPI domain decomposition as before
 - PARALLEL region around principal OpenMP loops
 - work divided between threads geometrically
 - each thread responsible for its own section of the 2D grid
 - amount of synchronisation reduced
 - OpenMP uses 2D decomposition rather than 1D
-

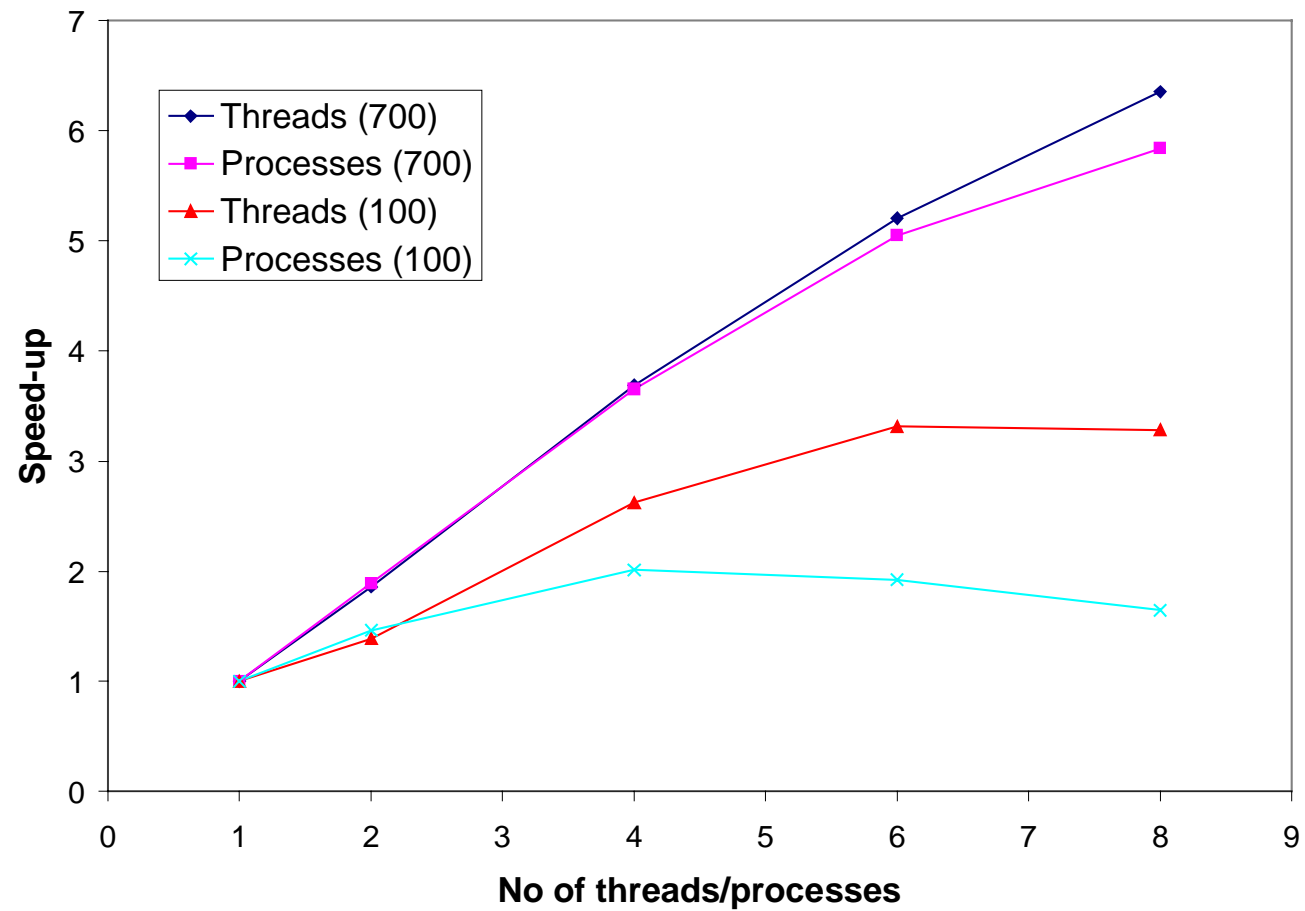


- scaling with OpenMP threads has decreased
- extra synchronisation still an issue

▶ SPMD code

- OpenMP parallelisation no longer occurs beneath the MPI parallelisation
 - global topology created: threads + processes have a global id
 - nearest neighbour threads + processes determined
 - 2D grid divided geometrically between threads + processes
 - halo swaps carried for each iteration of the code
 - if sending to a neighbour located on the same process
 - read/write carried out
 - if sending to a neighbour located on a different process
 - MPI send/receives used
 - only requires nearest neighbour synchronisation
 - thread-safe MPI
-





- ▶ OpenMP scaling better than MPI scaling
 - due to halo swaps
 - OpenMP: read/writes
 - MPI: explicit message passing
 - ▶ Combinations of threads /processes
 - no of threads increases - times decreases
 - ▶ Conclusions
 - Fine grain problem sizes
 - Synchronisation
 - non-hierarchical model required
-

▶ Aim

- to develop a real mixed-mode application
- to assess the performance benefits
- to exploit SMP clusters effectively

▶ Quantum Monte Carlo code

- Electronic structures HPCI consortium
- Computationally intense: successful MPI version already exists
- Predicting the electronic structure and properties of real materials
- f77 / f90 code

▶ Quantum Monte Carlo techniques

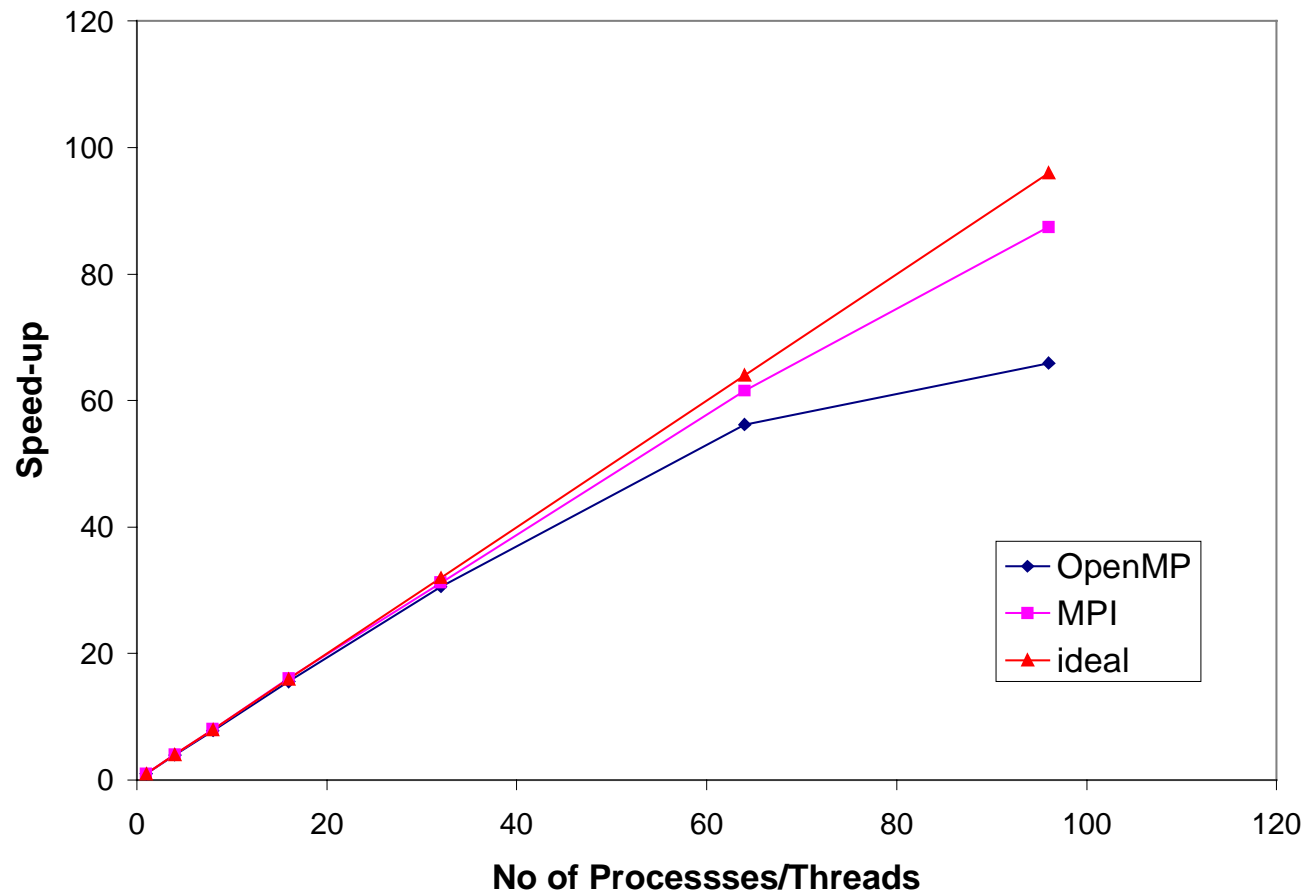
- Variational Monte Carlo (VMC)
 - Diffusion Monte Carlo (DMC)
-

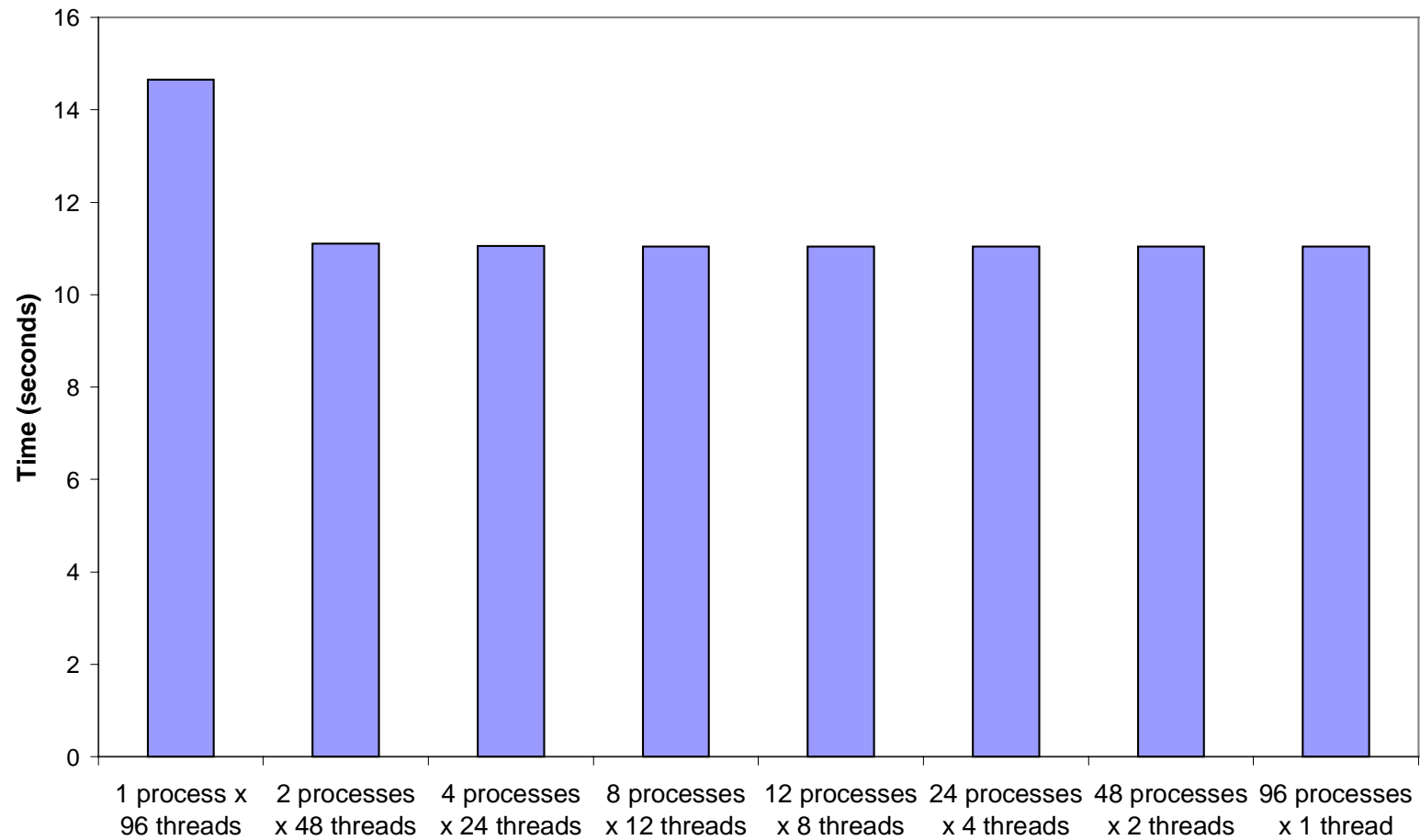
- ▶ 1) Initialise the ensemble of 'walkers'
 - ▶ 2) Update each walker in the ensemble. For each walker:
 - Move the electron
 - calculate the local energy and other variables of interest
 - calculate the new weight
 - accumulate the local energy contributions for this walker
 - breed or kill new walkers based on the energy
 - ▶ 3) Evaluate new generation averages
 - ▶ 4) After N generations calculate new 'block' averages
 - ▶ 5) Iterate steps 2-4 until equilibrium is reached
-

- ▶ Master-slave model
 - Electron configurations divided between slave processes
 - Master process broadcasts parameters to slave processes
 - Slave process evaluate properties dependant on subset of electron configurations: return properties to master process
 - Repeat until convergence achieved
 - ▶ Variational Monte Carlo
 - each process assigned a fixed no of electron configurations
 - inter-process communication only required for the final result
 - ▶ Diffusion Monte Carlo
 - electron configurations created/ annihilated: no on each process may change after each 'block': poor load-balancing
 - electron configurations redistributed between the processes after each block
-

- ▶ Majority of time spent in the loop over electron configurations
 - Compiler directives placed around this loop
 - Course grain parallelism
 - comparable level to MPI code: allows direct comparison
 - 150 variables, numerous modules and subroutine calls
 - Two principle shared arrays
 - start of loop: copied to temporary private arrays
 - end of loop: copied back to shared arrays
 - Electron configurations change: size of temporary arrays change
 - Ensure arrays copied back in the same order as sequential version: ORDERED statement
 - ▶ Mixed code
 - Hierarchical model: in general OpenMP beneath MPI
 - exceptions: MPI_BCASTs during 1st iteration: CRITICAL section
-

- ▶ SGI Origin 2000, 300MHz R12000 processors





- ▶ OpenMP scaling
 - similar to MPI to 32 processors, tails off considerably above 64
 - ▶ Thread / process combinations
 - similar times, exception 96 threads / 1 process
 - ▶ MPI scaling
 - redistribution of electron configurations between processes
 - all-to-one + point-to-point communications
 - ▶ OpenMP scaling
 - cc-NUMA architecture of the Origin 2000
 - data placement policy changes have no effect
 - MPI calls within thread sequential regions
 - only occur during the first iteration
 - ORDERED statement
 - hierarchical model - added synchronisation
-

- ▶ Clearly not the most effective mechanism for all codes
 - SMPs or clustered SMPs
 - ▶ Significant benefit may be obtained in certain situations:
 - poor scaling with MPI processes
 - replicated data codes
 - restricted MPI process codes
 - poorly optimised or limited scaling MPI implementations
 - ▶ Hierarchical models
 - non-thread safe MPI
 - Synchronisation often an issue
-

▶ EPCC

– <http://www.epcc.ed.ac.uk/>

▶ EWOMP2000

– <http://www.epcc.ed.ac.uk/ewomp2000/>

▶ UKHEC

– <http://www.ukhec.ac.uk/>
