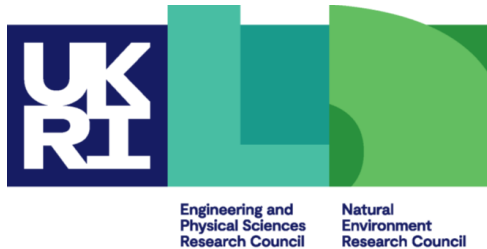


MPI Shared Memory Model

MPI processes behaving as threads



Reusing this material



This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

<https://creativecommons.org/licenses/by-nc-sa/4.0/>

This means you are free to copy and redistribute the material and adapt and build on the material under the following terms: You must give appropriate credit, provide a link to the license and indicate if changes were made. If you adapt or build on the material you must distribute your work under the same license as the original.

Acknowledge EPCC as follows: “© EPCC, The University of Edinburgh, www.epcc.ed.ac.uk”

Note that this presentation contains images owned by others. Please seek their permission before reusing these images.

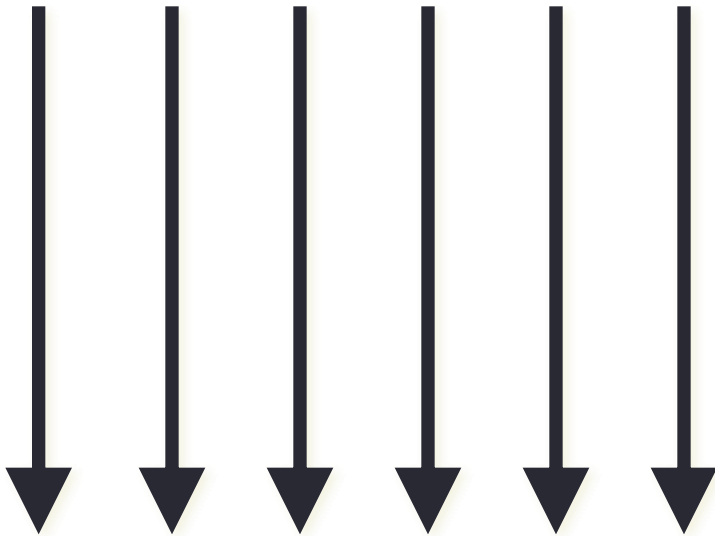
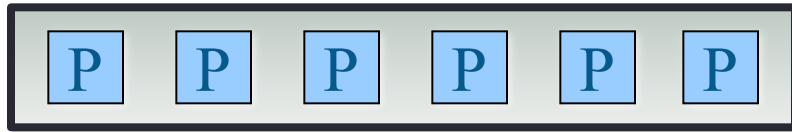
Overview

- Motivation
- Node-local communicators
- Shared window allocation
- Synchronisation

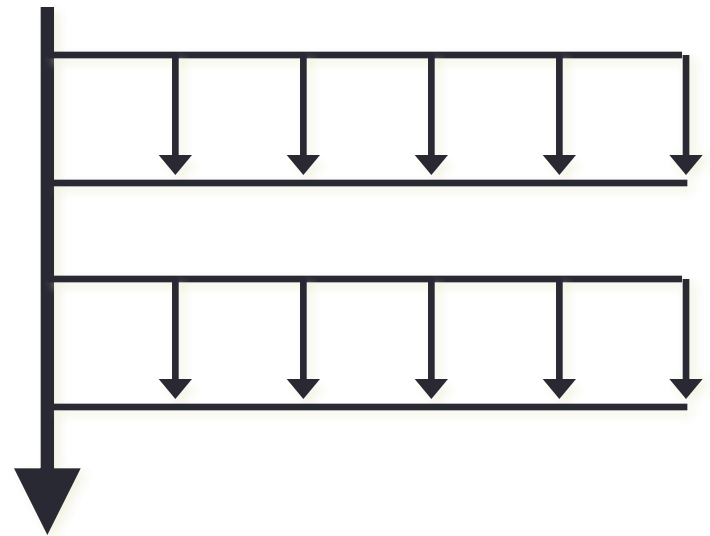
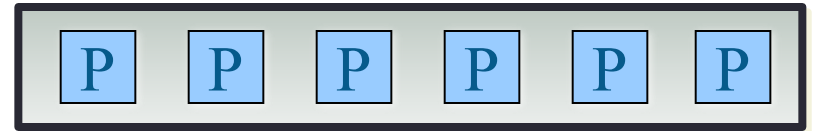
MPI + OpenMP

- In OMP parallel regions, all threads access shared arrays
 - why can't we do this with MPI processes?

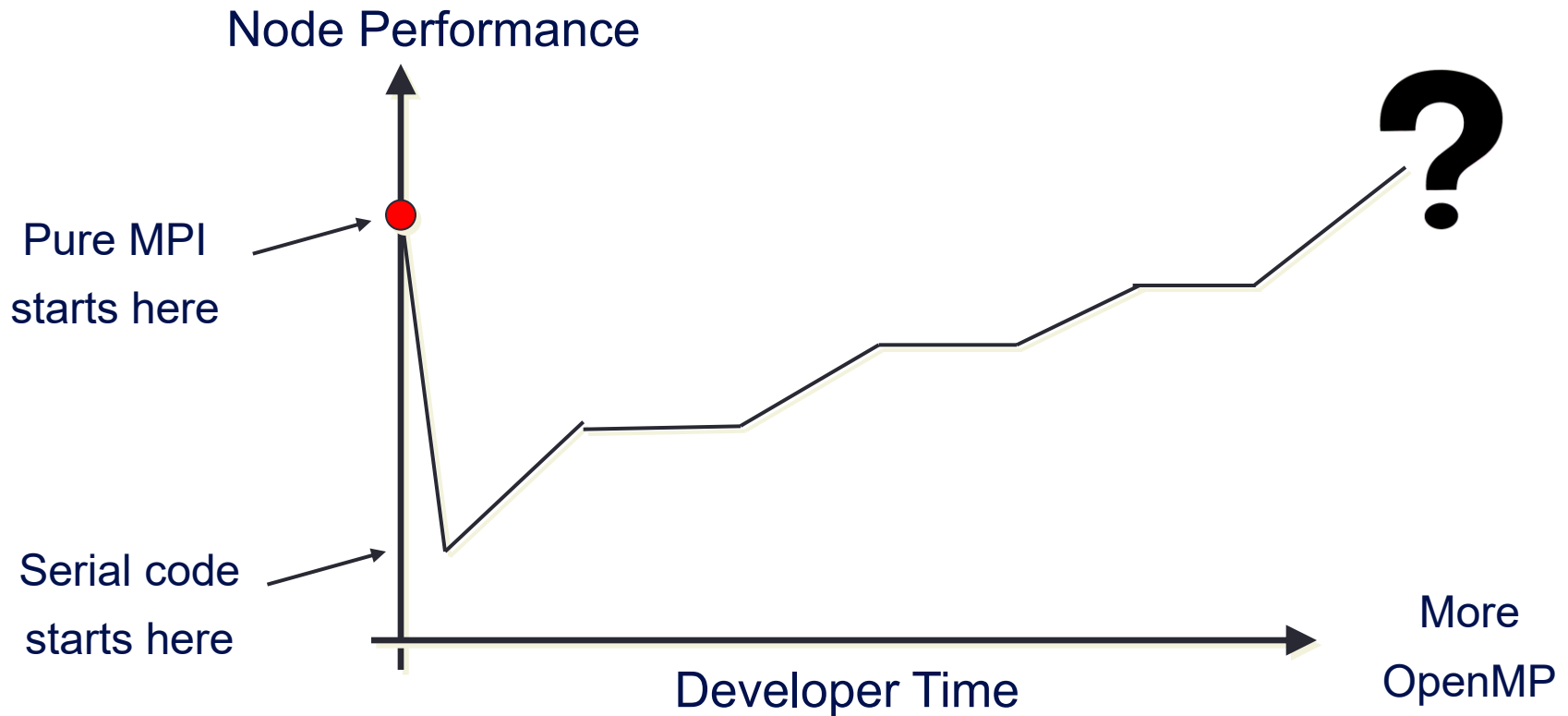
MPI



MPI + OpenMP



Consequences of MPI + OpenMP



- Some successes reported - often due to “threshold” effects
 - not enough memory to use all cores with MPI
 - fixed scalability limit of MPI parallelisation (e.g. slab-based FFTs)

Other use cases

- Saving memory
 - some programs have large, static lookup tables
 - in MPI, need one copy per process
 - on ARCHER2, this means 128 copies on each node!
 - much more efficient to have one copy per node
- NUMA-friendly memory allocation
 - sometimes, rank 0 on a node allocates lots of memory at startup
 - possible to use all the memory on the local NUMA region
 - ARCHER2 has 8 NUMA regions each with 32 MiB
 - subsequent allocations on ranks 0-15 now in different NUMA region
 - can be a disaster for performance
 - MPI shared memory allocation can be collective

Exploiting Shared Memory

- With standard RMA
 - publish local memory in a collective shared window
 - can do read and write with `MPI_Get` / `MPI_Put`
 - plus appropriate synchronisation, e.g. `MPI_Win_fence()`
- Seems wasteful on a node
 - why can't all processes just read and write directly as in OpenMP?
- Requirement
 - technically requires the Unified model
 - where there is no distinction between RMA and local memory
 - can check this calling `MPI_Win_get_attr` with `MPI_WIN_MODEL`
 - model should be `MPI_WIN_UNIFIED`
 - this is not a restriction in practice for standard CPU architectures

Procedure

- Processes join separate communicators for each node
- Shared array allocation across all processes on a node
 - each process receives a local array
 - OS can arrange for local arrays to be part of a single global array
- Remote access by indexing outside limits of local array
 - e.g. `localarray[-1]` will be last entry on the previous process
- Need appropriate synchronisation for remote accesses
- Still need MPI calls for inter-node communication
 - e.g. standard send and receive

Splitting the communicator

```
int MPI_Comm_split_type(MPI_Comm comm, int split_type,  
    int key, MPI_Info  info, MPI_Comm *newcomm)
```

```
MPI_COMM_SPLIT_TYPE(COMM, SPLIT_TYPE, KEY, INFO,  
    NEWCOMM, IERROR)
```

```
INTEGER COMM, SPLIT_TYPE, KEY, INFO, NEWCOMM, IERROR
```

- comm: parent communicator, e.g. MPI_COMM_WORLD
- split_type: MPI_COMM_TYPE_SHARED
- key: controls rank ordering within sub-communicator
- info: can just use default: MPI_INFO_NULL

Example

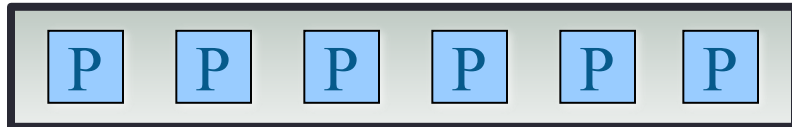
```
MPI_Comm_split_type(MPI_COMM_WORLD, MPI_COMM_TYPE_SHARED,  
rank, MPI_INFO_NULL, &nodecomm);
```

COMM_WORLD

size = 12

rank

0 1 2 3 4 5



6 7 8 9 10 11



0 1 2 3 4 5

rank
size = 6
nodecomm

0 1 2 3 4 5

rank
size = 6
nodecomm

Allocating the array

```
int MPI_Win_allocate_shared (MPI_Aint size, int disp_unit,  
    MPI_Info info, MPI_Comm comm, void *baseptr, MPI_Win *win)
```

```
MPI_WIN_ALLOCATE_SHARED(SIZE, DISP_UNIT, INFO, COMM, BASEPTR,  
    WIN, IERROR)
```

```
INTEGER(KIND=MPI_ADDRESS_KIND) SIZE, BASEPTR
```

```
INTEGER DISP_UNIT, INFO, COMM, WIN, IERROR
```

- size: window size in bytes
- disp_unit: basic counting unit in bytes, e.g. sizeof(int)
- info: can just use default: MPI_INFO_NULL
- comm: parent comm (must be within a single node)
- baseptr: allocated storage
- win: allocated window

Traffic Model Example

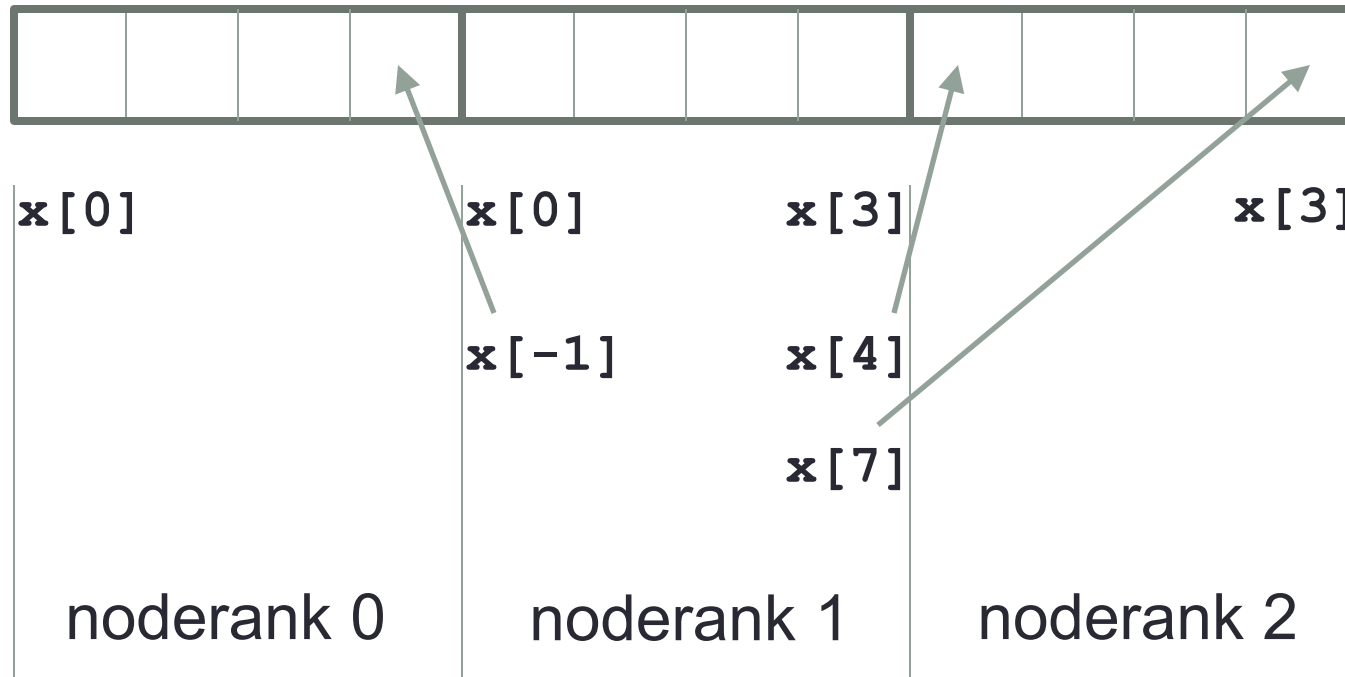
```
MPI_Comm nodecomm;
int *oldroad;
MPI_Win nodewin;
MPI_Aint winsize;
int displ_unit;

winsize = (nlocal+2)*sizeof(int);

// displacements counted in units of integers
disp_unit = sizeof(int);

MPI_Win_allocate_shared(winsize, displ_unit,
                        MPI_INFO_NULL, nodecomm, &oldroad, &nodewin);
```

Shared Array with nlocal = 2



- Default is contiguous block of memory across processes
 - use value of info, `alloc_shared_noncontig = true`, to relax this

Accessing another rank's memory

- In previous diagram
 - rank 1 can access rank 0's $x[0]$ by referencing its own $x[-4]$
- Might be more convenient to reference as $x_0[0]$
 - but how do we find out address for x_0 ?
 - or perhaps data is only allocated on rank 0 – what is its address?
- Rank 0 could `MPI_Send` its value of x to rank 1
 - will not work in general!
- Separate processes can have different virtual addresses (i.e. pointer values) for the same physical location
 - OS may do this deliberately to foil buffer overflow hacking attacks
- Must use special call
 - see `MPI_Win_shared_query()`
 - gives us a local pointer which we can use to access remote data

Synchronisation

- Can do halo swapping by direct copies
 - need to ensure data is ready beforehand and available afterwards
 - requires synchronisation, e.g. `MPI_Win_fence()`
 - takes hints – can just set to default of 0
- Entirely analogous to OpenMP
 - bracket remote accesses with `omp_barrier` or `begin / end parallel`

```
MPI_Win_fence(0, nodewin);  
oldroad[nlocal+2] = oldroad[nlocal]  
oldroad[-1]      = oldroad[1];  
MPI_Win_fence(0, nodewin);
```

Off-node comms

- Direct read / write only works within node
- Still need MPI calls for inter-node
 - e.g. `noderank = 0` and `noderank = nodesize-1` call `MPI_Send / Recv`
 - could actually use *any* rank to do this ...
- This must take place in `MPI_COMM_WORLD`

Conclusion

- Relatively simple syntax for shared memory in MPI
 - much better than roll-your-own solutions
- Possible use cases
 - on-node communications without needing MPI
 - one copy of static data per node (not per process)
- Advantages
 - an incremental “plug and play” approach unlike MPI + OpenMP
- Disadvantages
 - no automatic support for splitting up parallel loops
 - global array may have halo data sprinkled inside
 - so may not help in some memory-limited cases