Advanced Parallel Programming
Communicator Management

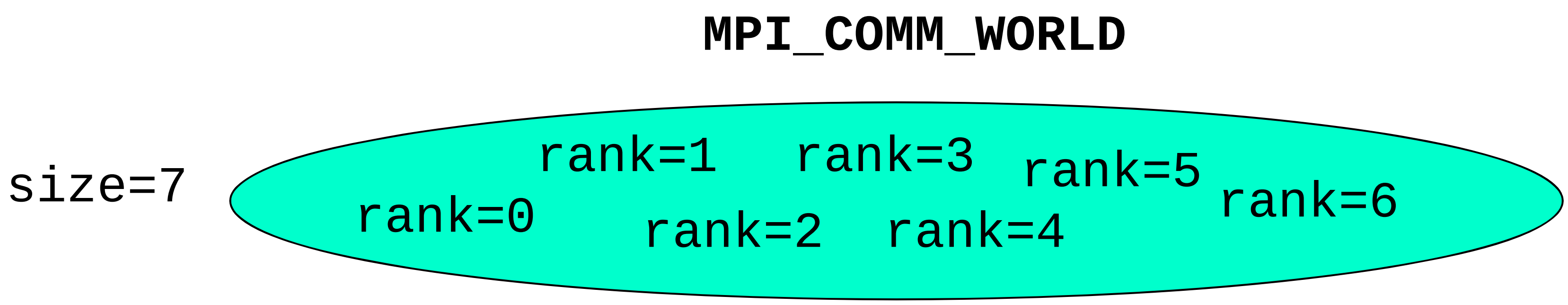
David Henty, Fiona Reid

Overview

- Lecture will cover
  - Communicators in MPI
  - Manipulating communicators
  - Examples of usage:
    - Optimising communications on hierarchical systems
    - Task farms
- Practical
  - Implementing an “Allreduce” over rows and columns

Communicators

- All MPI communications take place within a communicator
  - a group of processes with necessary information for message passing
  - there is one pre-defined communicator: **`MPI_COMM_WORLD`**
  - contains all the available processes
- Messages move within a communicator
  - E.g., point-to-point send/receive must use same communicator
  - Collective communications occur in single communicator
  - unlike tags, it is not possible to use a wildcard



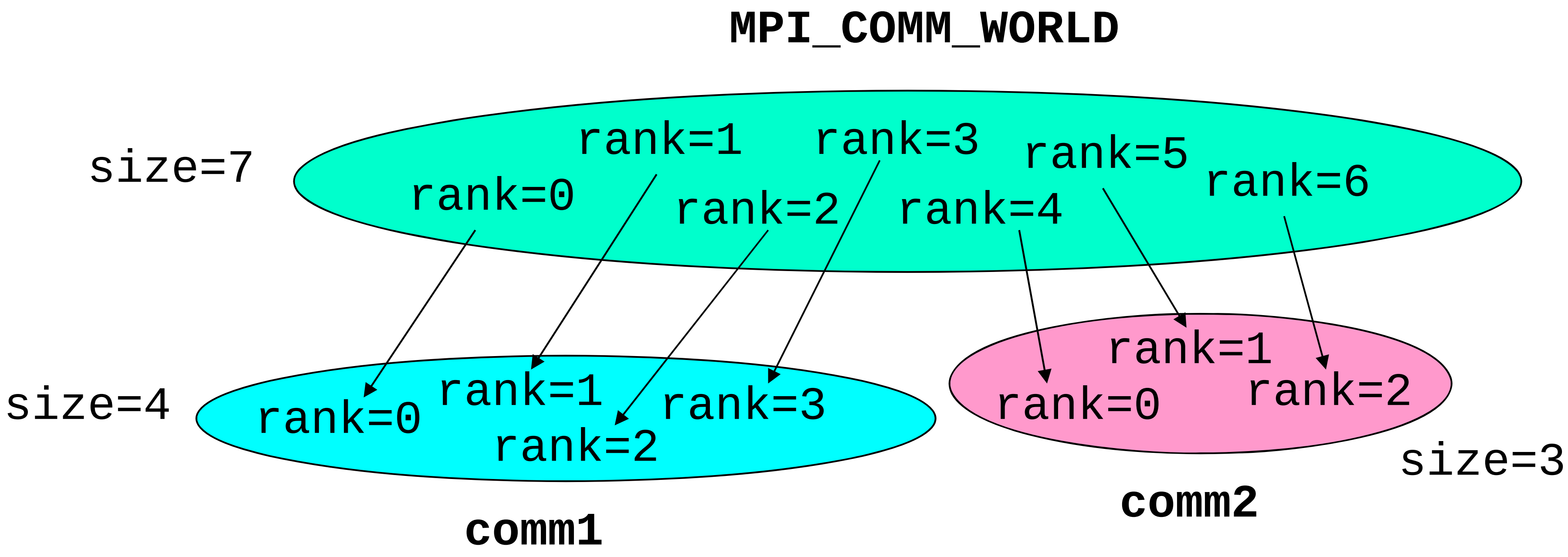
Use of communicators

- Question: Can I just use **`MPI_COMM_WORLD`** for everything?
- Answer: Yes
  – many people use **`MPI_COMM_WORLD`** everywhere in their MPI programs
- Better programming practice suggests
  – abstract the communicator using the MPI handle
  – such usage offers very powerful benefits

```
MPI_Comm comm;        /* or INTEGER for Fortran */
comm = MPI_COMM_WORLD;
...
MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &size);
....
```

Split Communicators

- It is possible to sub-divide communicators
- E.g.,split **MPI_COMM_WORLD**
  - Two sub-communicators can have the same or differing sizes
  - Each process has a new rank within each sub communicator
  - Messages in different communicators guaranteed not to interact



MPI interface

- **`MPI_Comm_split()`**
  - splits an existing communicator into disjoint (i.e. non-overlapping) subgroups
- Syntax, C:

```
int MPI_Comm_split(MPI_Comm comm, int colour, int
                        key, MPI_Comm *newcomm)
```

- Fortran:

```
MPI_COMM_SPLIT(COMM, COLOUR, KEY, NEWCOMM, IERROR)
INTEGER COMM, COLOUR, KEY, NEWCOMM, IERROR
```

- **`colour`** – controls assignment to new communicator
- **`key`** – controls rank assignment within new communicator

What happens…

- **`MPI_Comm_split()`** is collective
  - must be executed by **all** processes in group associated with **`comm`**
- New communicator is created
  - for each unique value of **`colour`**
  - All processes having the same **`colour`** will be in the same sub-communicator
- New ranks 0…size-1
  - determined by the (ascending) value of the key
  - If keys are same, then MPI determines the new rank
  - Processes with the same **`colour`** are ordered according to their **`key`**
- Allows for arbitrary splitting
  - other routines for particular cases, e.g. **`MPI_Cart_sub`**

Split Communicators – C example

```
MPI_Comm comm, newcomm;
int colour, rank, size;
comm = MPI_COMM_WORLD;
MPI_Comm_rank(comm, &rank);
/* Set colour depending on rank: Even numbered ranks have
   colour = 0, odd have colour = 1 */
colour = rank%2;
MPI_Comm_split(comm, colour, rank, &newcomm);
MPI_Comm_size (newcomm, &size);
MPI_Comm_rank (newcomm, &rank);
```

Split Communicators – Fortran example

```
integer :: comm, newcomm
integer :: colour, rank, size, errcode
comm = MPI_COMM_WORLD
call MPI_COMM_RANK(comm, rank, errcode)
```

! Again, set colour according to rank

```
colour = mod(rank,2)
call MPI_COMM_SPLIT(comm, colour, rank, newcomm,&
errcode)
MPI_COMM_SIZE(newcomm, size, errcode)
MPI_COMM_RANK(newcomm, rank, errcode)
```

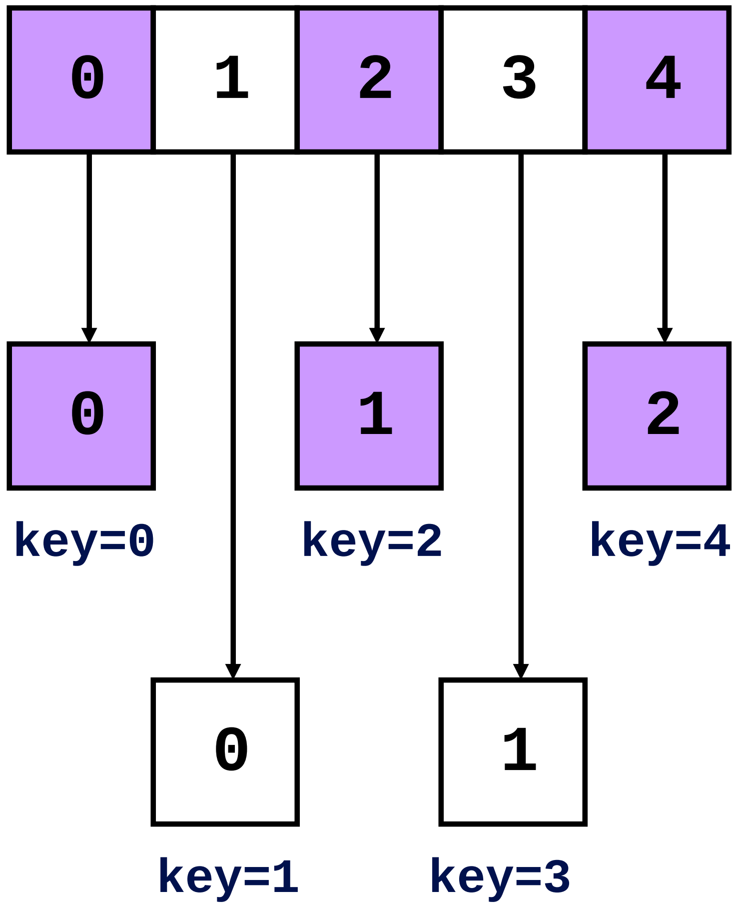
Diagrammatically

- Rank and size of the new communicator

```
MPI_COMM_WORLD, size=5
color = rank%2;
key = rank;
```

```
newcomm, color=0, size=3
```

```
newcomm, color=1, size=2
```



Duplicating Communicators

- **`MPI_Comm_dup()`**
  - creates a new communicator of the same size
  - but a different context
- Syntax, C:

```
int MPI_Comm_dup(MPI_Comm comm,
                 MPI_Comm *newcomm)
```

- Fortran:

```
MPI_COMM_DUP(COMM, NEWCOMM, IERROR)
INTEGER COMM, NEWCOMM, IERROR
```

Using Duplicate Communicators

- An important use is for libraries
  - Library code should not use same communicator(s) as user code
  - Possible to mix up user and library messages
  - Almost certain to be fatal
- Instead, can duplicate the user’s communicator
  - Encapsulated in library (hidden from user)
  - Use new communicator for library messages
  - Messages guaranteed not to interfere with user messages
  - Could *try* to do this by reserving tags in MPI (tricky) but wildcarding of tags can still create problems

Freeing Communicators

- **`MPI_Comm_free()`**
  - a **collective** operation which destroys an unwanted communicator
- Syntax, C:

```
int MPI_Comm_free(MPI_Comm * comm)
```

- Fortran:

```
MPI_COMM_FREE(COMM, IERROR)

INTEGER COMM, IERROR
```

  - Any pending communications which use the communicator will complete normally
  - Deallocation occurs only if there are no more active references to the communication object

Advantages of Communicators

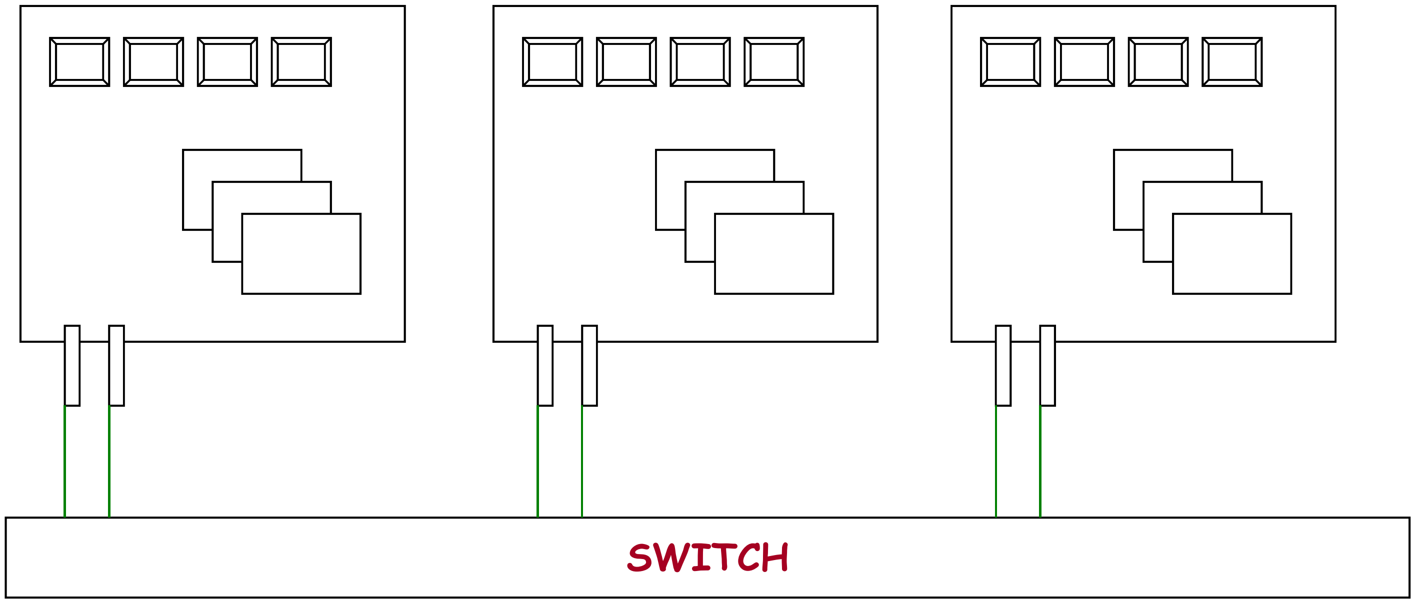
- Many requirements can be met by using communicators
  - Can't I just do this all with tags?
  - Possibly, but difficult, painful and error-prone
- Easier to use collective communications than point-to-point
  - Where subsets of **`MPI_COMM_WORLD`** are required
  - E.g., averages over coordinate directions in Cartesian grids
- In dynamic problems
  - Allows controlled assignment of different groups of processors to different tasks at run time

Applications, for example

- Linear algebra
  - row or column operations or act on specific regions of a matrix (diagonal, upper triangular etc)
- Hierarchical problems
  - Multi-grid problems e.g. overlapping grids or grids within grids
  - Adaptive mesh refinement
    - E.g. complexity may not be known until code runs, can use split comms to assign more processors to a part of the problem
- Taking advantage of locality
  - Especially for communication (e.g. group processors by node)
- Multiple instances of same parallel problem
  - Task farms

Fast and slow communication

- Many systems now hierarchical / heterogeneous
  - Chips with shared memory cores
  - “Nodes” of many chips with shared memory
  - Groups of nodes connected by an interconnect
  - Assume a “node” shares memory and communication hardware

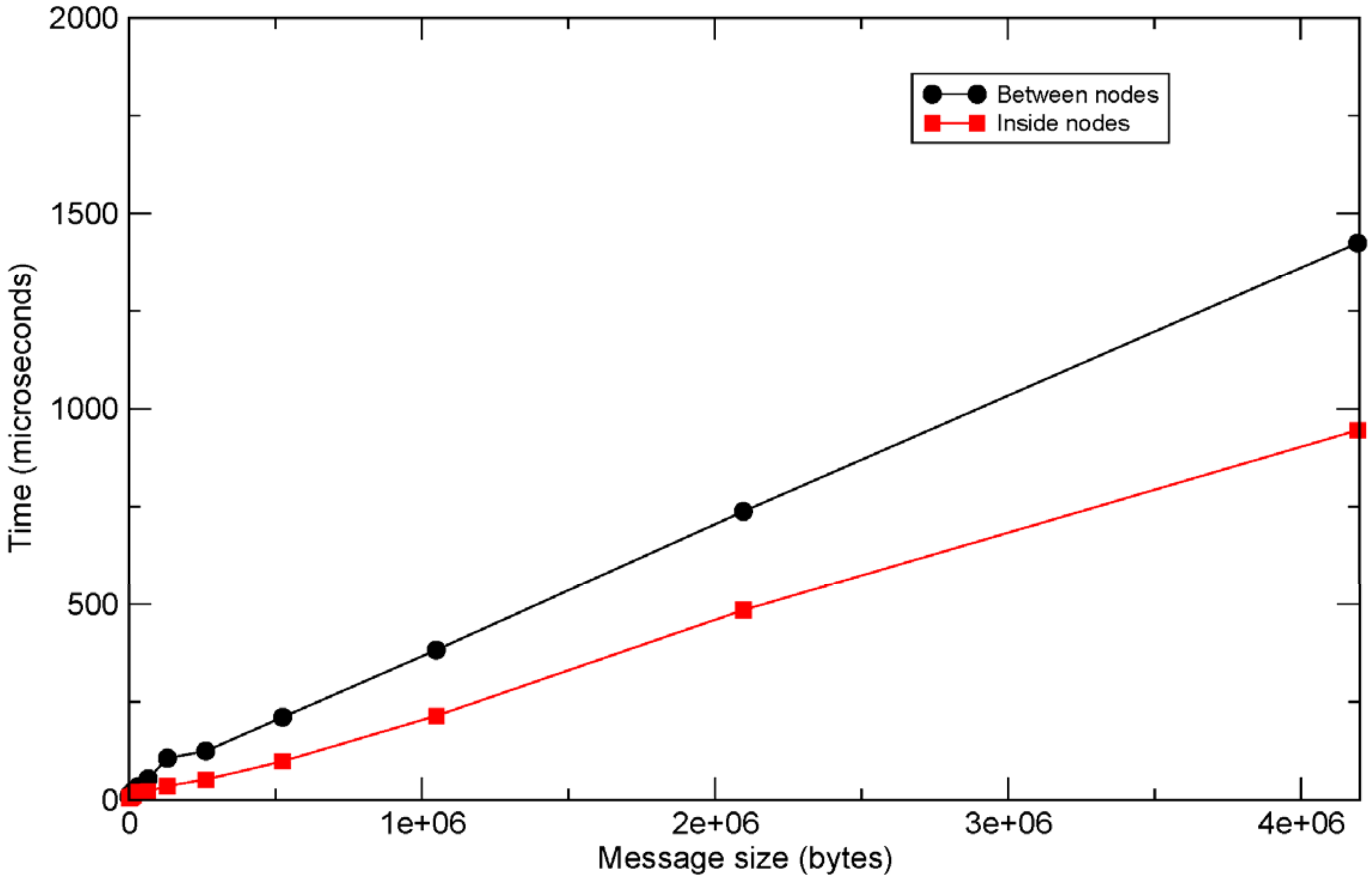


Message passing

- MPI may have two modes of operation
  - One optimised for use within a node (intra-node) via shared memory
  - One for communicating between nodes (inter-node) via network
- Performance may be quite different
  - E.g. for HPCx (previous national supercomputer: IBM)
    - MPI latency within node (shared memory) ~3µs
    - MPI latency between nodes (network) ~6µs
  - HECToR (current national supercomputer: Cray)
    - on-node MPI latency XE6 and XT4 ~0.5µs
    - off-node MPI latency 1.4µs (XE6) and 6.0µs (XT4)
- Do we benefit from this automatically?
  - May depend on the implementation of MPI
  - If MPI doesn't help, can try for ourselves using communicators

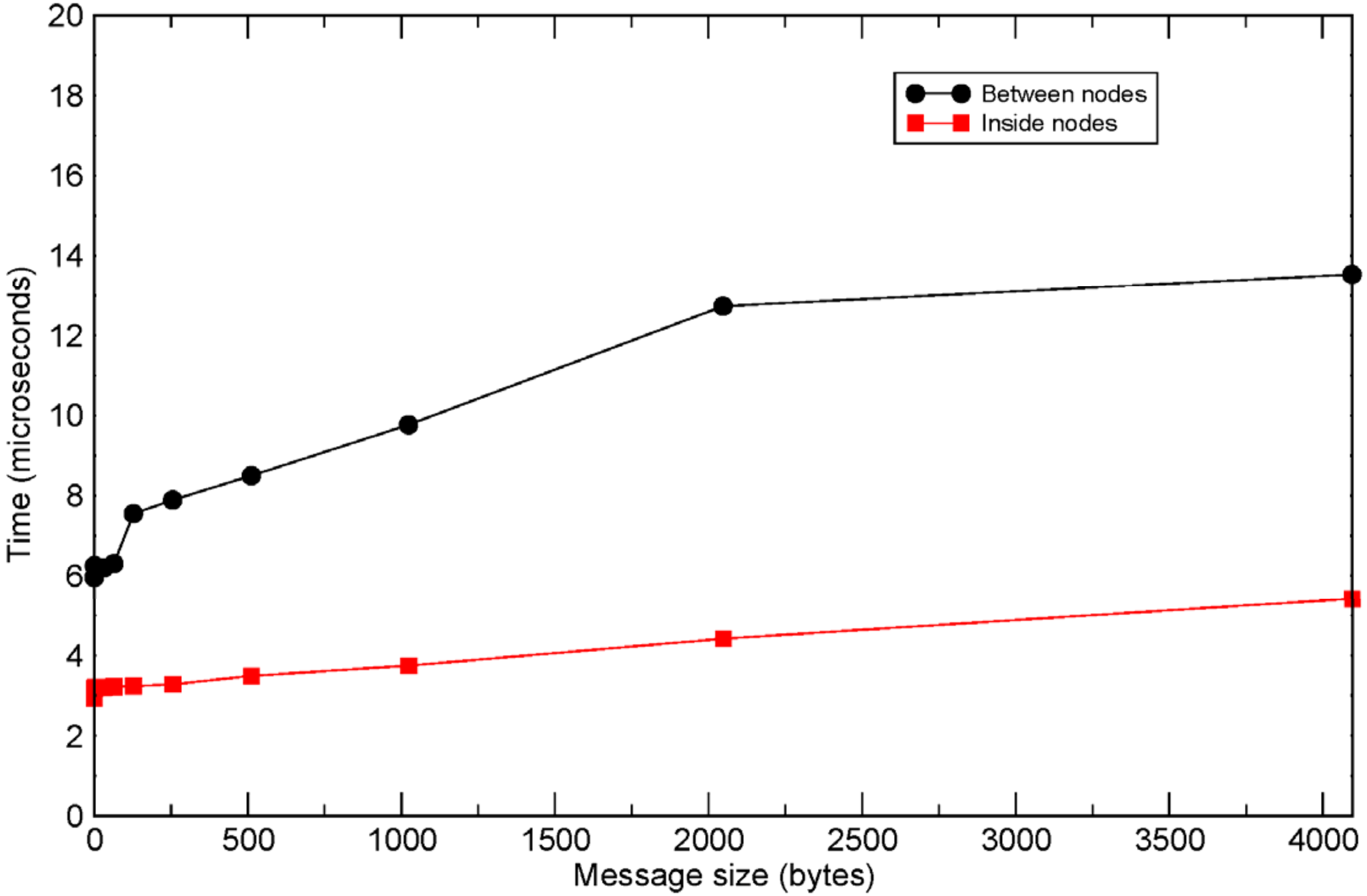
Intra/Inter node communications on HPCx

- Results from Ping Pong Intel MPI benchmark



Intra/Inter node communications on HPCx

- Results from Ping Pong Intel MPI benchmark



Using intra-node and inter-node messages

- Can we take advantage of the difference
  - E.g., to improve the performance of “Allreduce”
- So, want to reduce expensive operations
  - number of inter-node messages (latency)
  - the amount of data sent between nodes (bandwidth)
- Trade off against
  - Additional (cheap) intra-node communication

A Solution

- Split global communicator at node boundaries
- How to do this?
  - Need a way to identify hardware from software
    - i.e. need to know which physical processors reside on which physical nodes
- For example,
  - Use **`MPI_Get_processor_name()`**
  - to give a unique string for different nodes
  - e.g., on HPCx: **`l4f403`**, **`l1f405`**, etc
- Assume we have a function
  - **`int name_to_colour(const char *string)`**
  - Returns a unique integer for any given string

A Solution continued

- Pseudo code for the function might look like

```
hash = 0
 For each byte c in name
    hash -> 131*hash + c
```

  – Creates a unique hash value for each node name
  – 131 is used to avoid collisions
  – On many systems node names only differ by numerical digits.
  – E.g. node names **`l4f403`**, **`l1f405`** equate to 1169064111 and 2052563872 respectively

Intra-node communicator

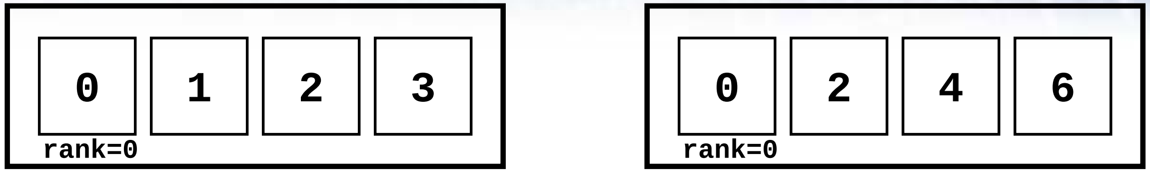
- Use this number to split the **`input`** communicator

```
MPI_Get_processor_name(procname,&len);
node_key = name_to_colour(procname);
MPI_Comm_split(input,node_key,0,&local);
```

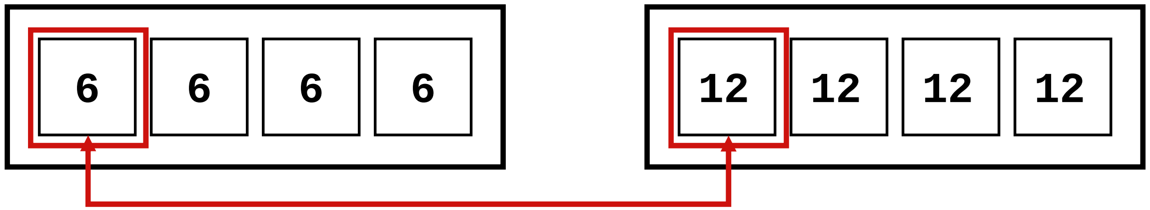
- **`local`** is now a communicator for the local node
- Now we can make communicators across nodes

```
MPI_Comm_rank(local,&lrank);
MPI_Comm_split(input,lrank,0,&cross);
```

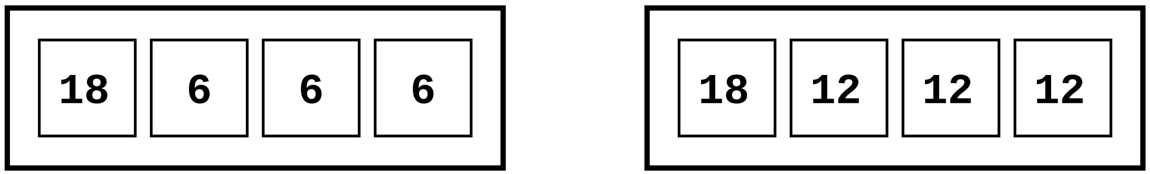
Allreduce with two nodes



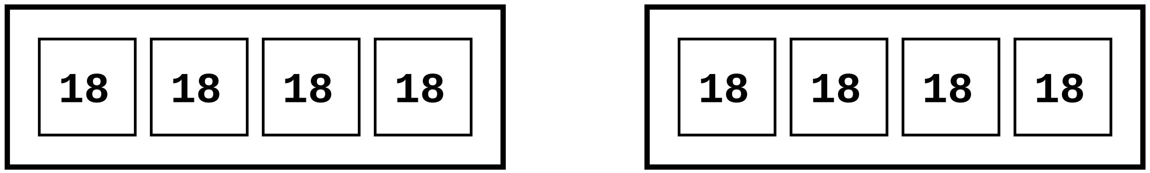
Perform an allreduce (sum) across each node – all comms inside a node



Perform an allreduce (sum) across nodes for rank=0 – comms between nodes



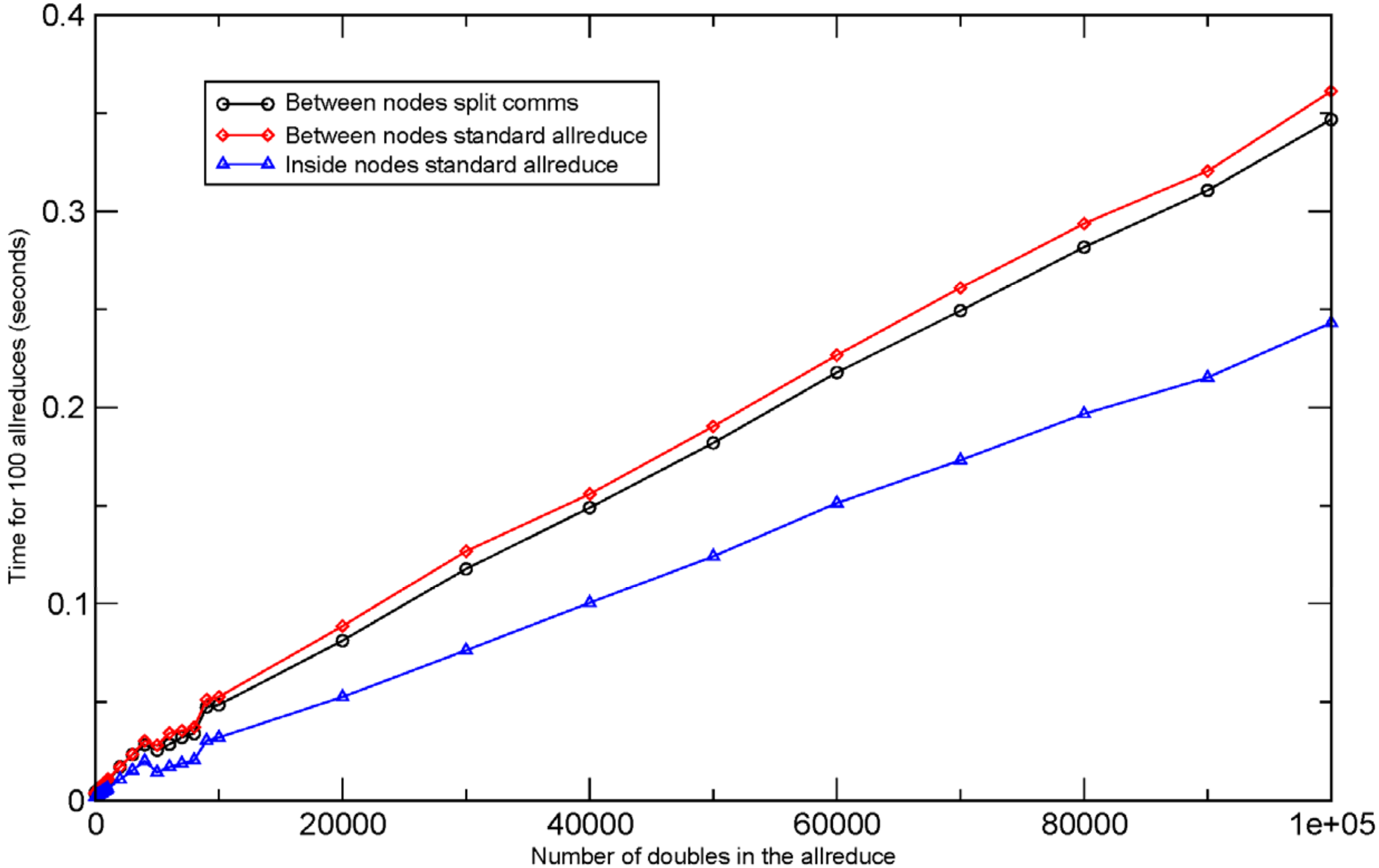
Broadcast result with each node – all comms inside a node



All processors across nodes now have the same value

Sample results

- Results from Allreduce across 2 nodes of HPCx



Summary

- Communicators in MPI
  - Many manipulations possible
  - A powerful mechanism
  - Learn to use!
- Applications of split communicators
  - Increasing locality of communication
- Collectives
  - hope that MPI implementations do this automatically …
  - manual implementation of Allreduce a good test example
  - … is there a benefit on HECToR?