Advanced Parallel Programming

Networks and All-to-All communication

David Henty, Joachim Hein
EPCC
The University of Edinburgh

Overview of this Lecture

- All-to-All communications
  - MPI_Alltoall
  - MPI_Alltoallv

- Bi-sectional bandwidth, insertion bandwidth and performance of Alltoall communications

Rationale

- Alll-to-all comms is central to several key parallel algorithms
  - e.g. parallel Fast Fourier Transform
- Need to understand what limits performance in practice
- Try to optimise for particular architectures (see later)

MPI_Alltoall

- The simple command MPI_Alltoall offers a convenient way to initiate an All-to-All communication
- In C:

```
int MPI_Alltoall(
    void* sendbuf, int sendcount,
    MPI_Datatype sendtype,
    void* recvbuf, int recvcount,
    MPI_Datatype recvtype, MPI_Comm comm)
```

MPI_Alltoall (continued)

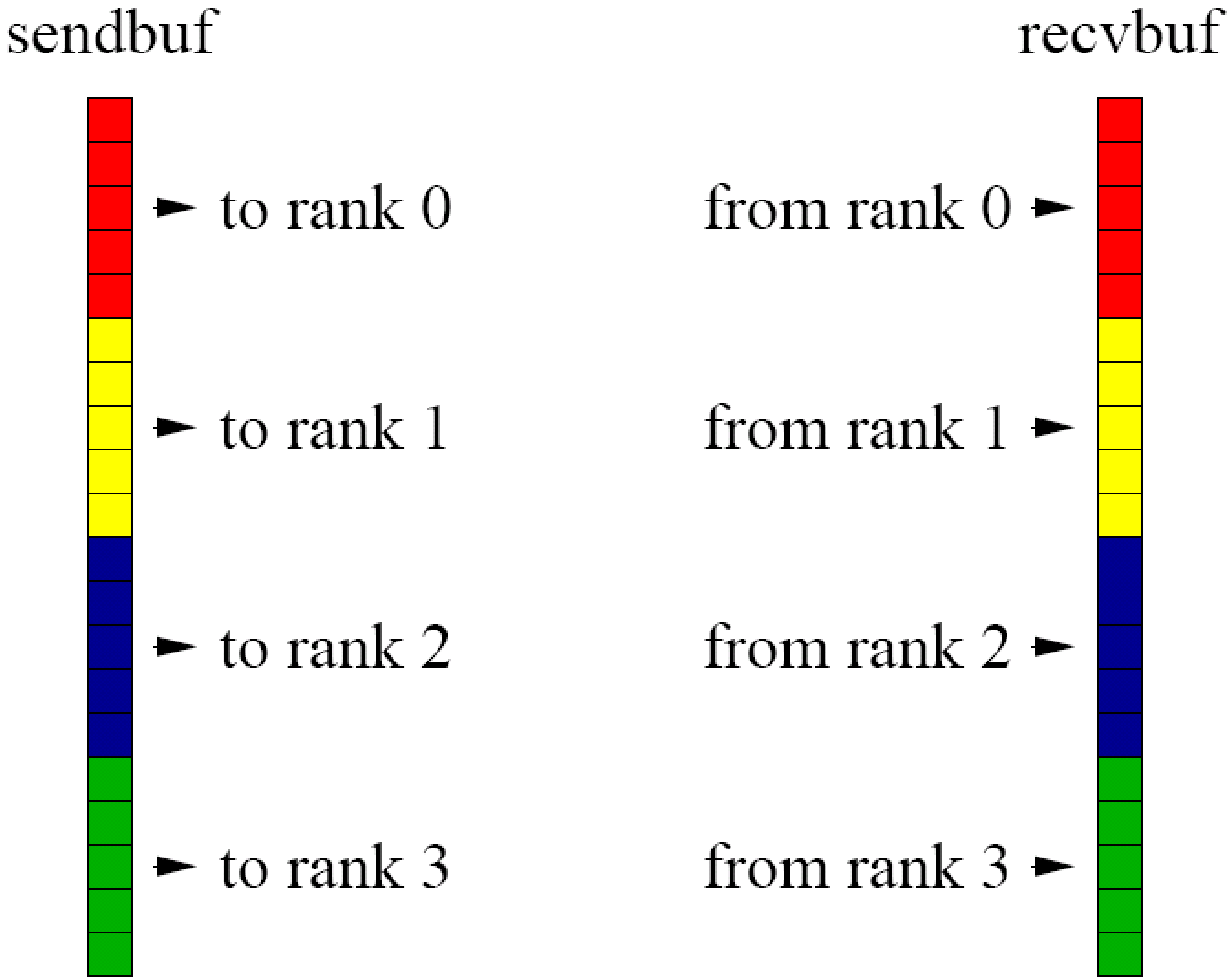
- In Fortran:

```
MPI_ALLTOALL(<type> SENDBUF,
    INTEGER SENDCOUNT,INTEGER SENDTYPE,
    <type> RECVBUF,INTEGER RECVCOUNT,
    INTEGER RECVTYPE,
    INTEGER COMM,INTEGER IERROR)
```

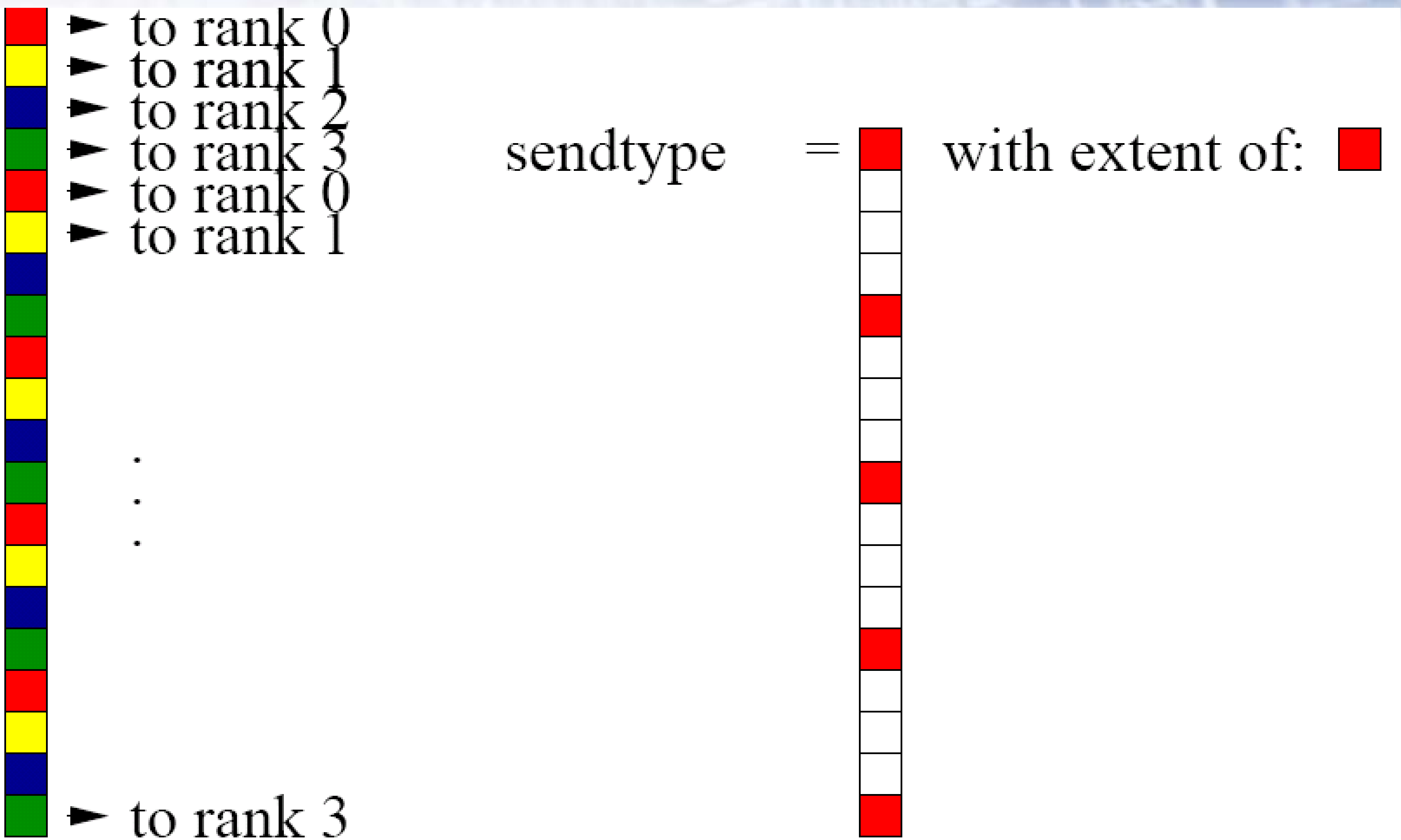
- Each processor has one send buffer and one receive buffer
  - exchange same number of elements, same datatype on each process
  - consecutive data elements separated by their extent

Example for buffer use

`sendcount = 5`, communicator with 4 processors



Strided data layout



- Typical case: Rows/columns of Fortran/C matrix
- Construction of such a data-type discussed later this lecture
  - alternatively copy into contiguous buffer

MPI_Alltoallv

- **MPI_Alltoallv** offers more flexibility
- The “v” stands for vector
- In C:

```
int MPI_Alltoallv(
    void* sendbuf, int *sendcounts,
    int *sdispls, MPI_Datatype sendtype,
    void *recvbuf, int *recvcounts,
    int *rdispls, MPI_Datatype recvtype,
    MPI_Comm comm);
```

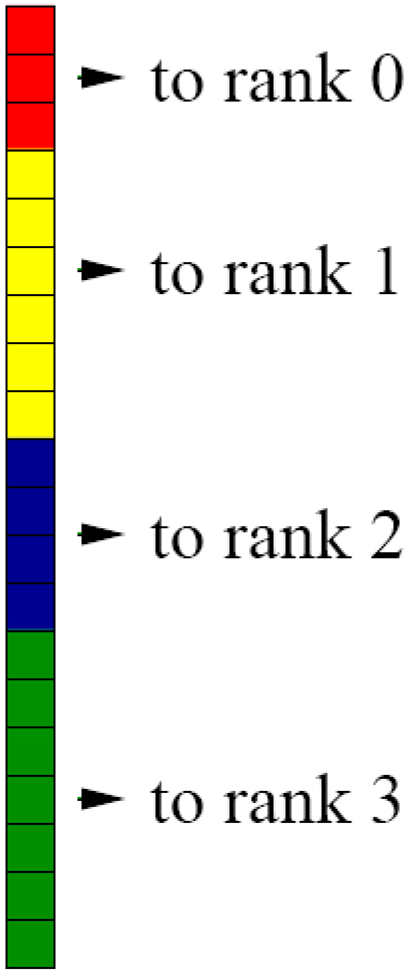
MPI_Alltoallv (continued)

- In Fortran:

```
MPI_ALLTOALLV(
    <type> SENDBUF, INTEGER SENDCOUNTS(*),
    INTEGER SDISPLS(*), INTEGER SENDTYPE,
    <type> RECVBUF, INTEGER RECVCOUNTS(*),
    INTEGER RDISPLS(*), INTEGER RECVTYPE,
    INTEGER COMM, INTEGER IERROR)
```

- Now: Array of **SENDCOUNTS** & **RECVCOUNTS**
- New: Array of displacements, specifying starting position
- Still: Single **SENDTYPE** & **RECVTYPE**

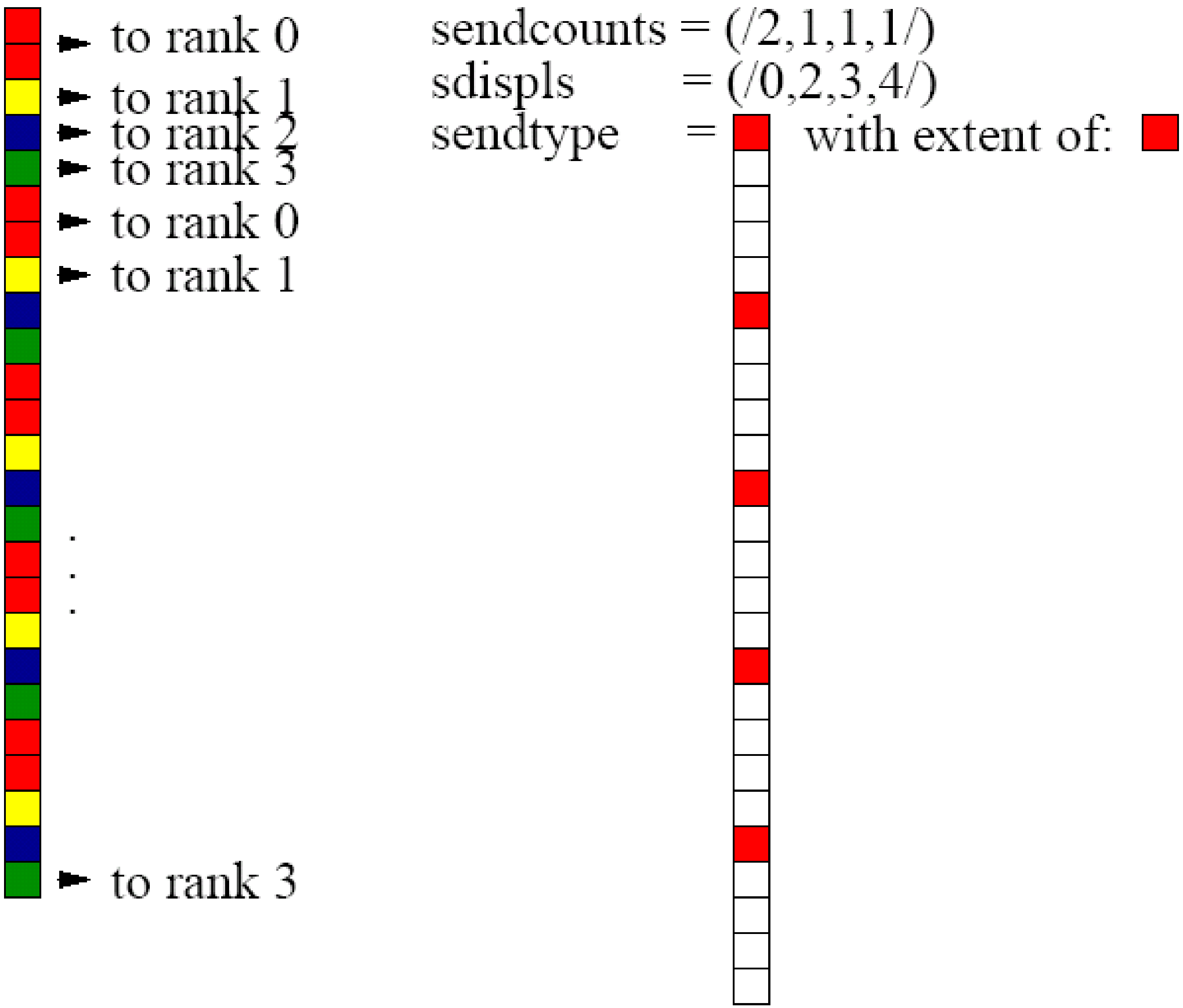
Sending different sized messages



sendcounts = (/3,6,4,7/)

sdispls = (/0,3,9,13/)

sendtype =


to rank 0
to rank 1
to rank 2
to rank 3
to rank 0
to rank 1
to rank 3
sendcounts = (/2,1,1,1/)
sdispls = (/0,2,3,4/)
sendtype =
with extent of:

Aside: making the derived data type (old!)

```
call mpi_type_vector(n_elem, 1, n_proc, &
  mpi_integer, vector, ierror)
call mpi_type_commit(vector, ierror)

bl_array = 1
disp_array(1) = 0
call mpi_type_size(mpi_integer, disp_array(2), ierror)
type_array(1) = vector
type_array(2) = MPI_UB

call mpi_type_struct(2, bl_array, disp_array, &
  type_array, a2a_stride_type, ierror)
call mpi_type commit(a2a_stride_type, ierror)
```

Create a vector for the strided data entries

Place **`MPI_UB`** as second block behind the 1st integer of the vector in the struct

Aside: making the derived data type (new!)

```
call mpi_type_vector(n_elem, 1, n_proc, &
  mpi_integer, vector, m_error)
call mpi_type_commit(vector, ierror)
```

Create a vector for the strided data entries

Explicitly resize the datatype to start at displacement 0 and end at a single integer

```
call mpi_type_size(MPI_INTEGER, intsize, ierror)
call mpi_type_create_resized(vector, 0, intsize,
  a2a_stride_type, ierror)
call mpi_type commit(a2a_stride_type, ierror)
```

Performance of All-to-All

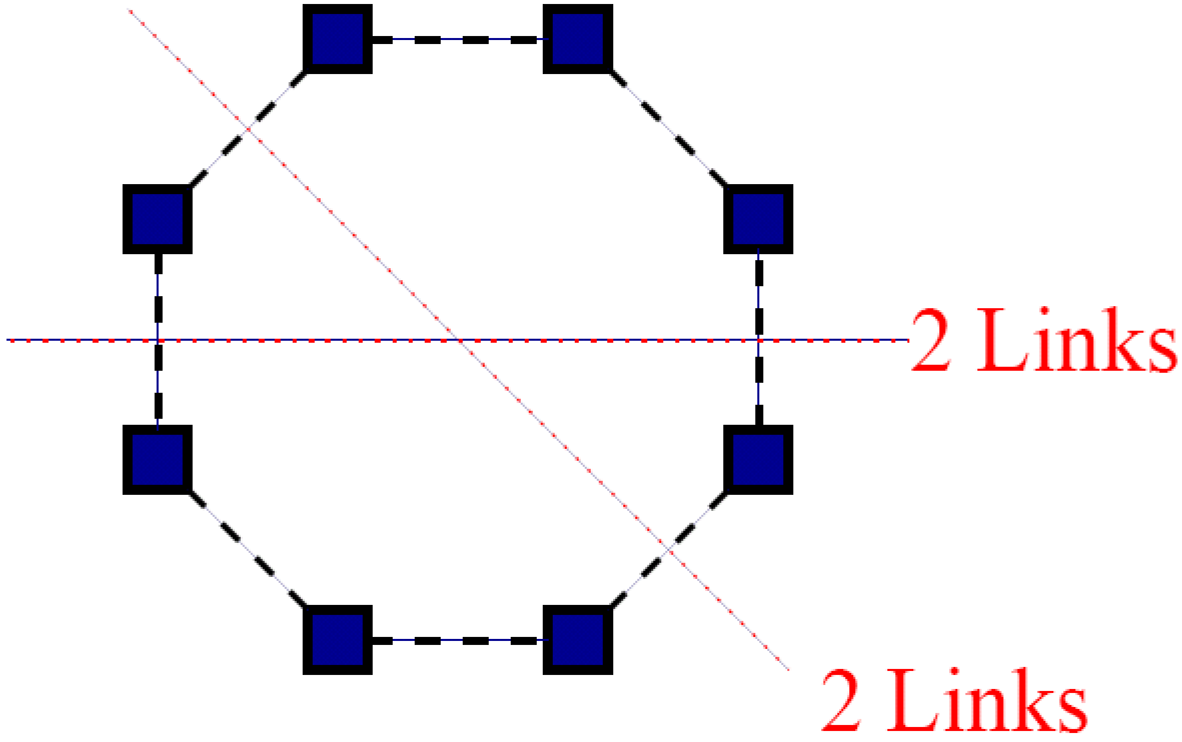
- Key bottlenecks for performance of the All-to-All communication include
  - Bi-sectional bandwidth
  - Insertion Bandwidth into the network

Reminder: Bi-sectional bandwidth

- Divide the processors into two equal sized groups
- Consider interconnect bandwidth between the two groups
- Redo for all possibilities to divide the machine
- The **minimum** observed interconnect bandwidth is called **bi-sectional** bandwidth

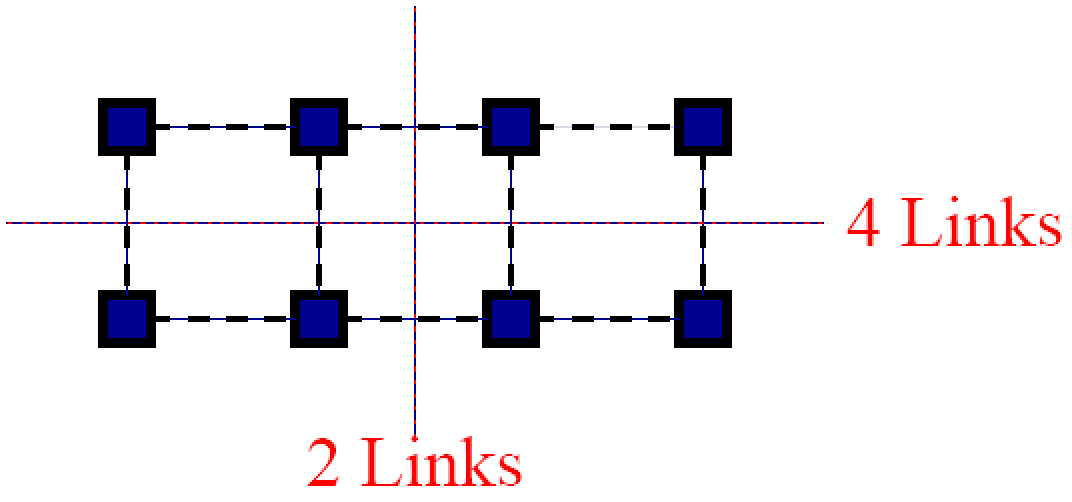
1st Example: Processors on a ring

- Independent of cut: 2× link speed for bi-sectional bandwidth



2nd Example: Open processor mesh

- Number of links depends on the cut
- Relevant cut: 2× link speed for bi-sectional bandwidth



All-to-all and bi-sectional bandwidth

- Assume a “normal” all-to-all: each processor sends the same amount of data to each other processor – no MPI_Alltoallv

- Cut the processor group into two halves
  - Each group has: half of the total data
  - Each group must transfer half of its data (quarter of total) to other side
  - This data needs to go through the “bi-section”

- All-to-all communication cannot complete faster than

  $$t_b = \tfrac{1}{4} D/s = \tfrac{1}{4} pd/s = \tfrac{1}{4} p^2 x/s$$

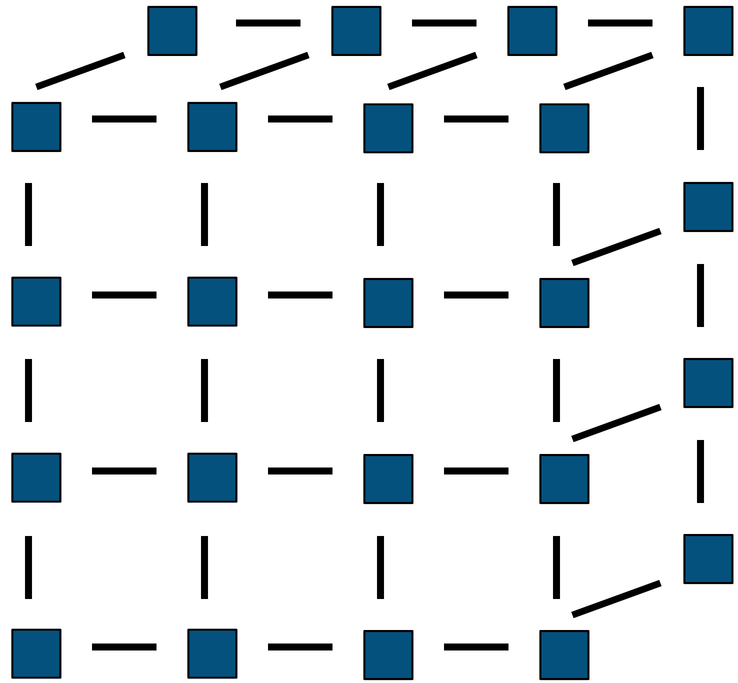
  s: bi-sectional bandwidth, D: total data of the problem

  d: data per process, x: message size, p: number of processes

BlueGene/L

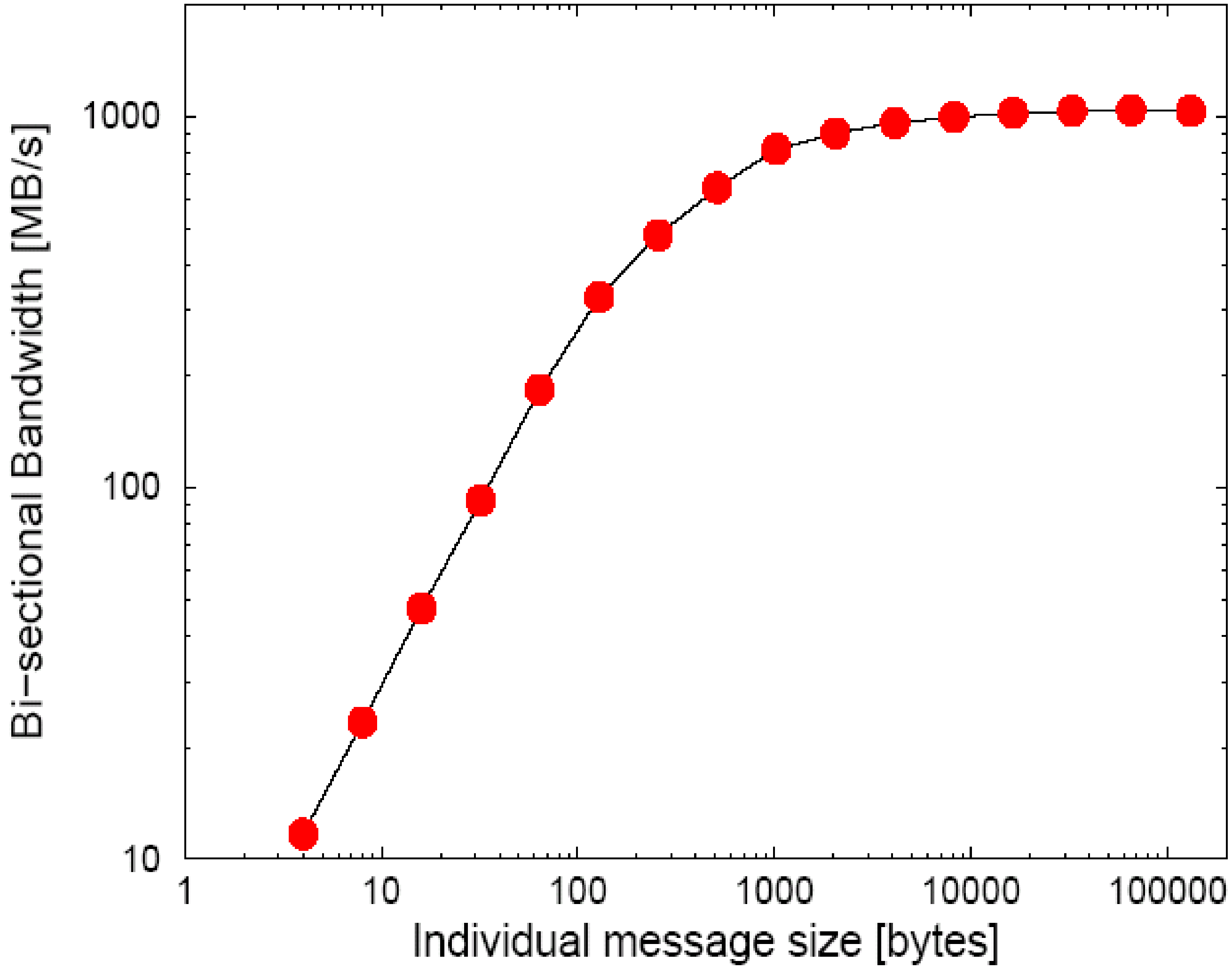
- Simple case
  - single processor per node
  - 3D mesh network

- What is bi-sectional bandwidth for 32 processor partition?



Example for 32 proc partition of BG/L

- Latency dominates for small messages
- Bi-sectional Bandwidth saturates at 1GB/s
- Agrees with 8× linkspeed of 147 MB/s

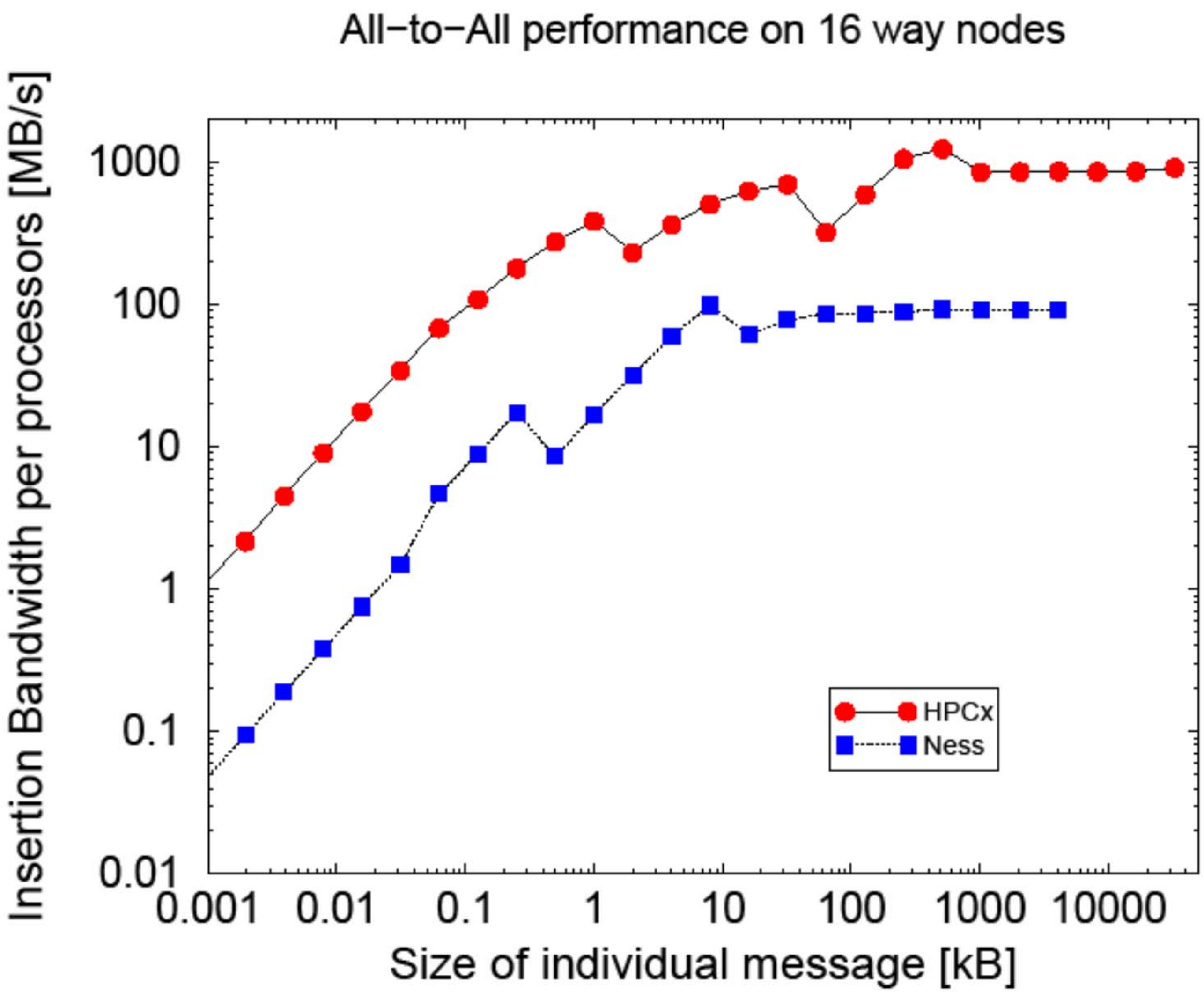


All-to-all and insertion bandwidth

- Insertion bandwidth can be another key bottle neck
  - how fast can data be sent to the network?
- Typically increases with the number of processors
  - each process can send data to the network separately
- Can get complicated for SMP clusters
  - some data is kept local to a node
  - some goes over the network

Example: All-to-All on a single SMP

- Insertion bandwidth related to memory bandwidth
  - HPCx 16 task: Copy 2GB/s per processor
  - Ness 16 task: Copy 0.37GB/s per processor



Limited in the bi-section or at the insertion?

- Networked machine: Look at the scaling with processor number
- Insertion limitation:
  - Insertion bandwidth per task independent of task count
  - Total insertion bandwidth proportional to task count
    - e.g. bandwidth doubles when doubling the task count
  - Compare insertion bandwidth per task to Ping-Ping results on (almost) empty machine
- Bi-section limitation:
  - Bi-sectional bandwidth typically not proportional to the task count, e.g. on a 3D meshed network (BlueGene), bi-sectional bandwidth increases by a factor of 4 when using 8 times the task count
  - Need to understand (a bit about) the network to be fully certain
  - Can depend on the location of your tasks on the physical machine
  - If insertion bandwidth per task decreases when increasing task count, we have an indication for limitation in the communication network

Summary

- All-to-all communications can be initiated conveniently by using **`MPI_Alltoall`** and **`MPI_Alltoallv`**
  - **`MPI_Alltoall`**: very simple to use
  - **`MPI_Alltoallv`**: allows more flexibility

- The performance of the operation is typically limited by
  - The bi-sectional bandwidth of the (partition of the) machine
  - The insertion bandwidth between the processor and the network

What can we do?

- Keep comms local to a node
  - placement of processes to nodes is an issue
  - e.g. use communicator management routines

- Avoid some of the MPI calls
  - hybrid MPI/OpenMP
  - no need for explicit data redistribution on a node