Mysteries of the Deep (Computing Messaging Framework): What happens inside of MPI on BGP and why it matters

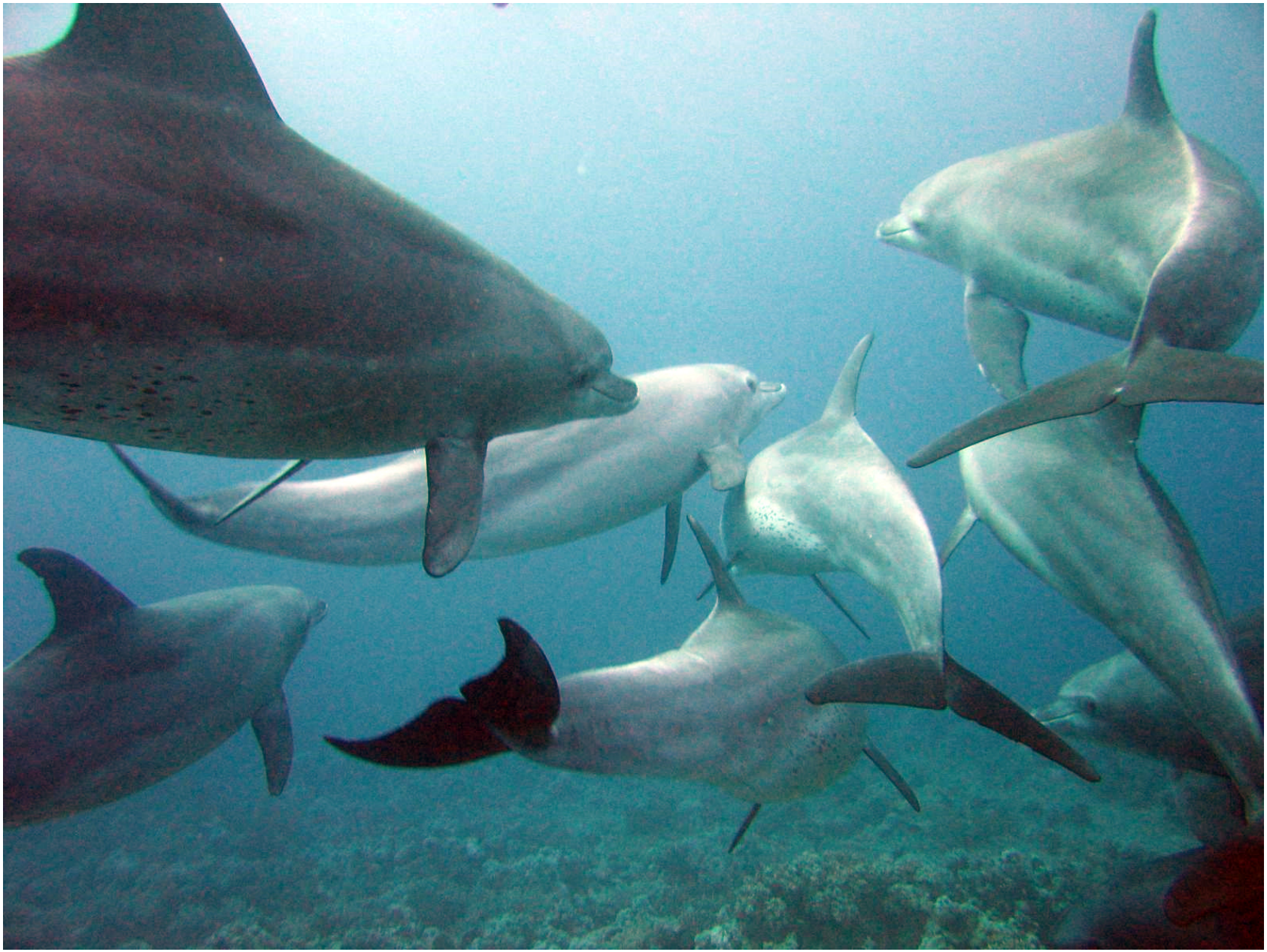
Jeff Hammond

Leadership Computing Facility
Argonne National Laboratory

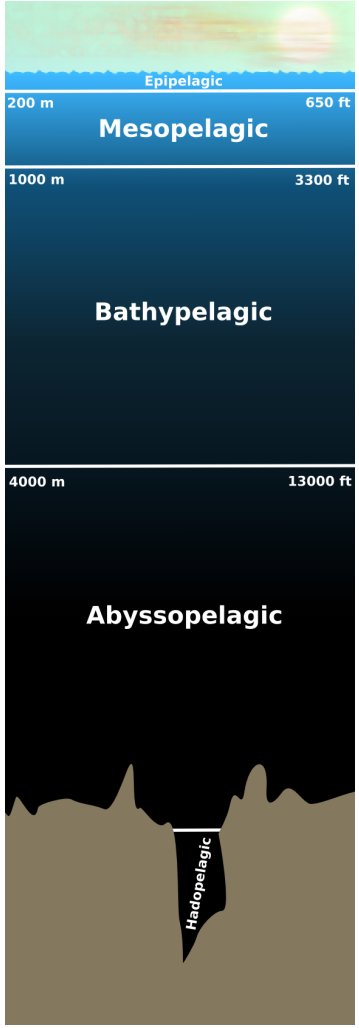
24 January 2011

The view from the boat

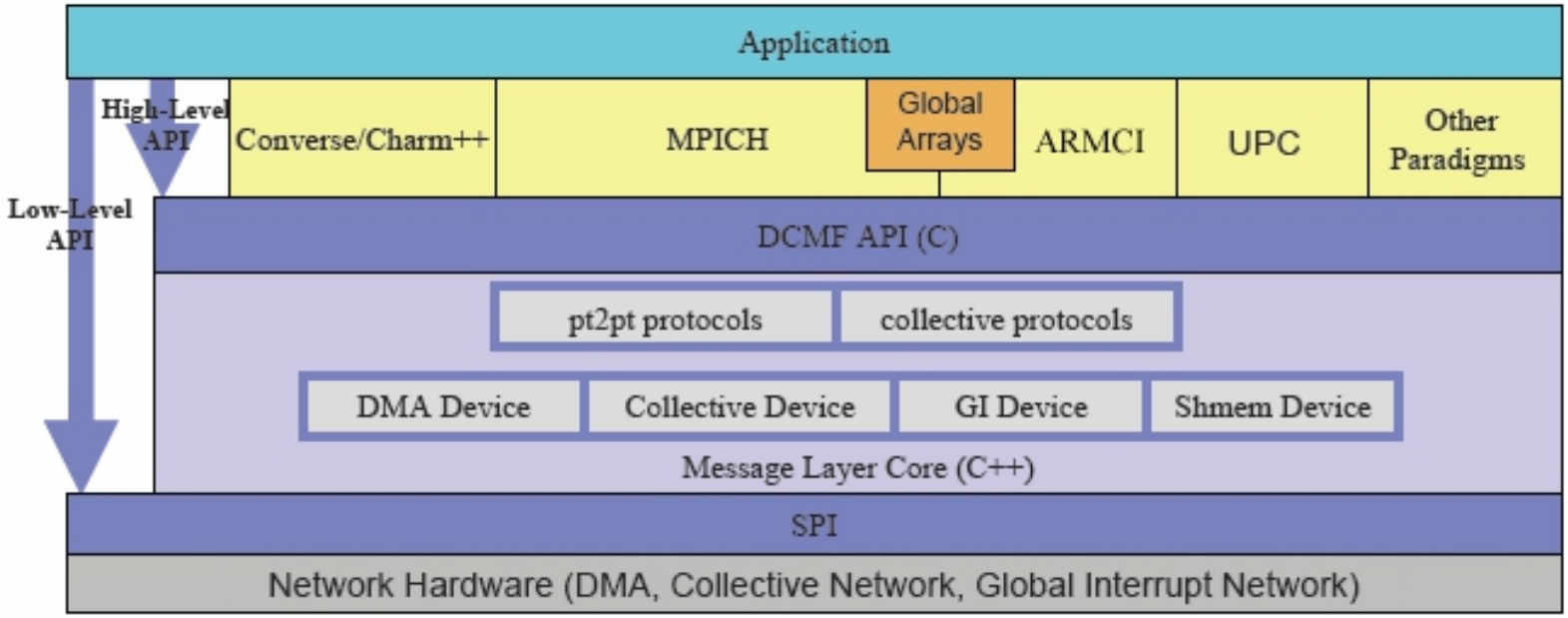
A reason to dive



But not too deep



Blue Gene/P Communication architecture



Source: IBM

DCMF is used to implement MPI(CH2), Charm++, ARMCI, GASNet, etc. It provides active-messages, RDMA and collectives.

All operations are *nonblocking* by default (except when specific hardware is blocking, e.g collective networks).

Performance results

Neighbor exchange

Testing injection (send) bandwidth along 1 to 6 links.

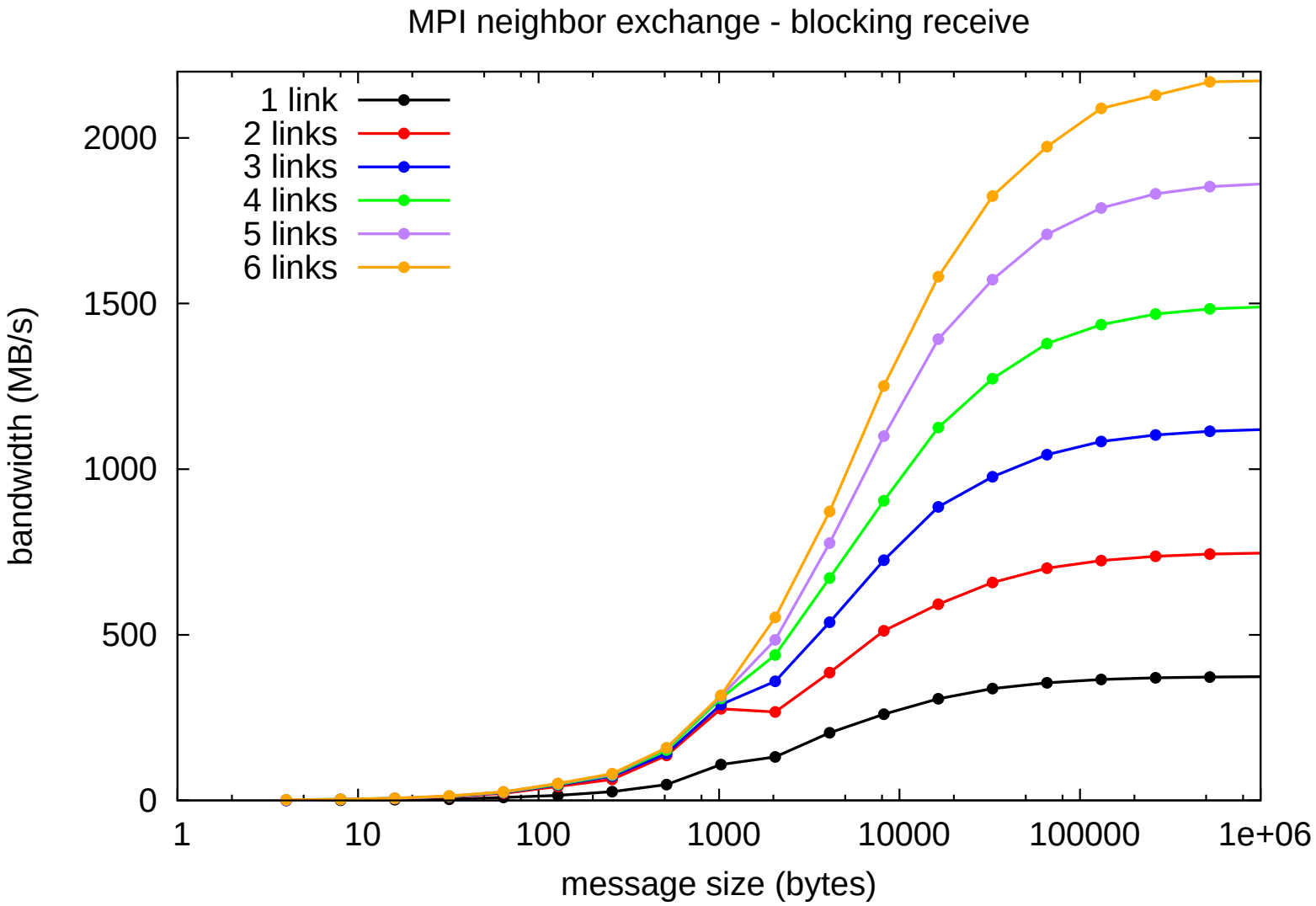
Explicitly mapped to torus using r = MPIX_torus2rank(x,y,z,t)

Pre-post receives (blocking or nonblocking followed by sleep)

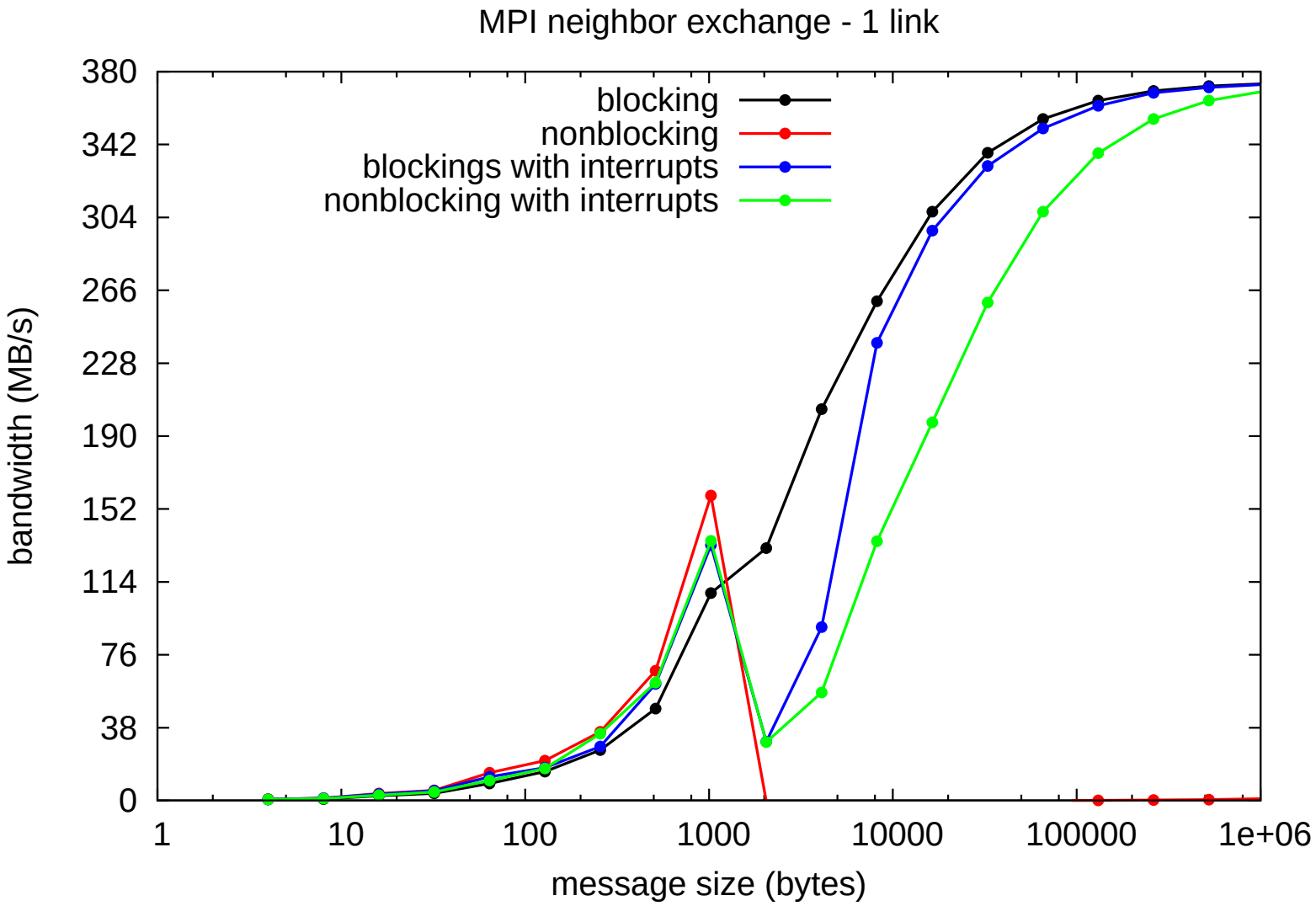
Nonblocking send followed by waitall.

No repetition in test but warmup along all 6 links done first.

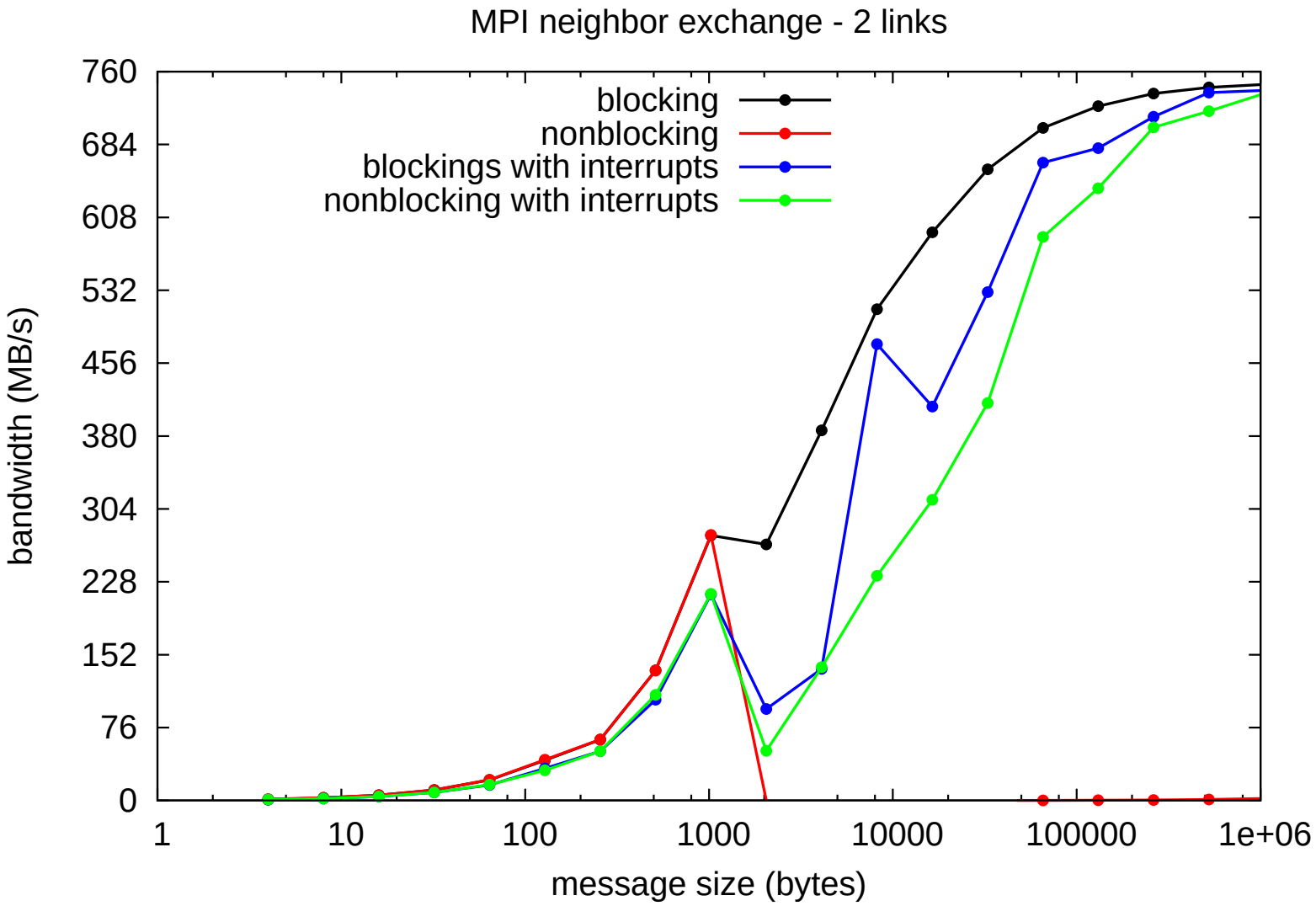
Neighbor exchange



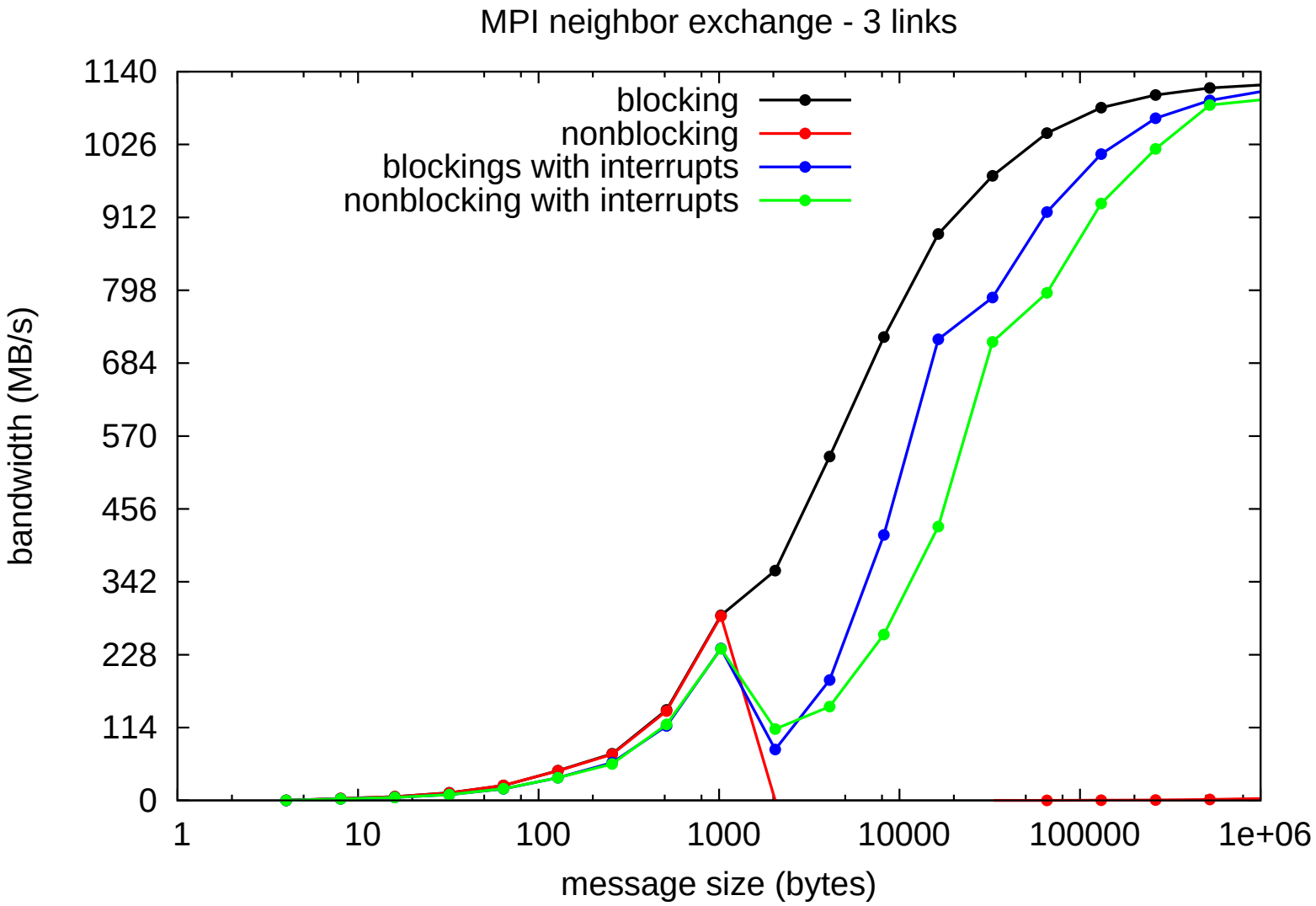
Neighbor exchange



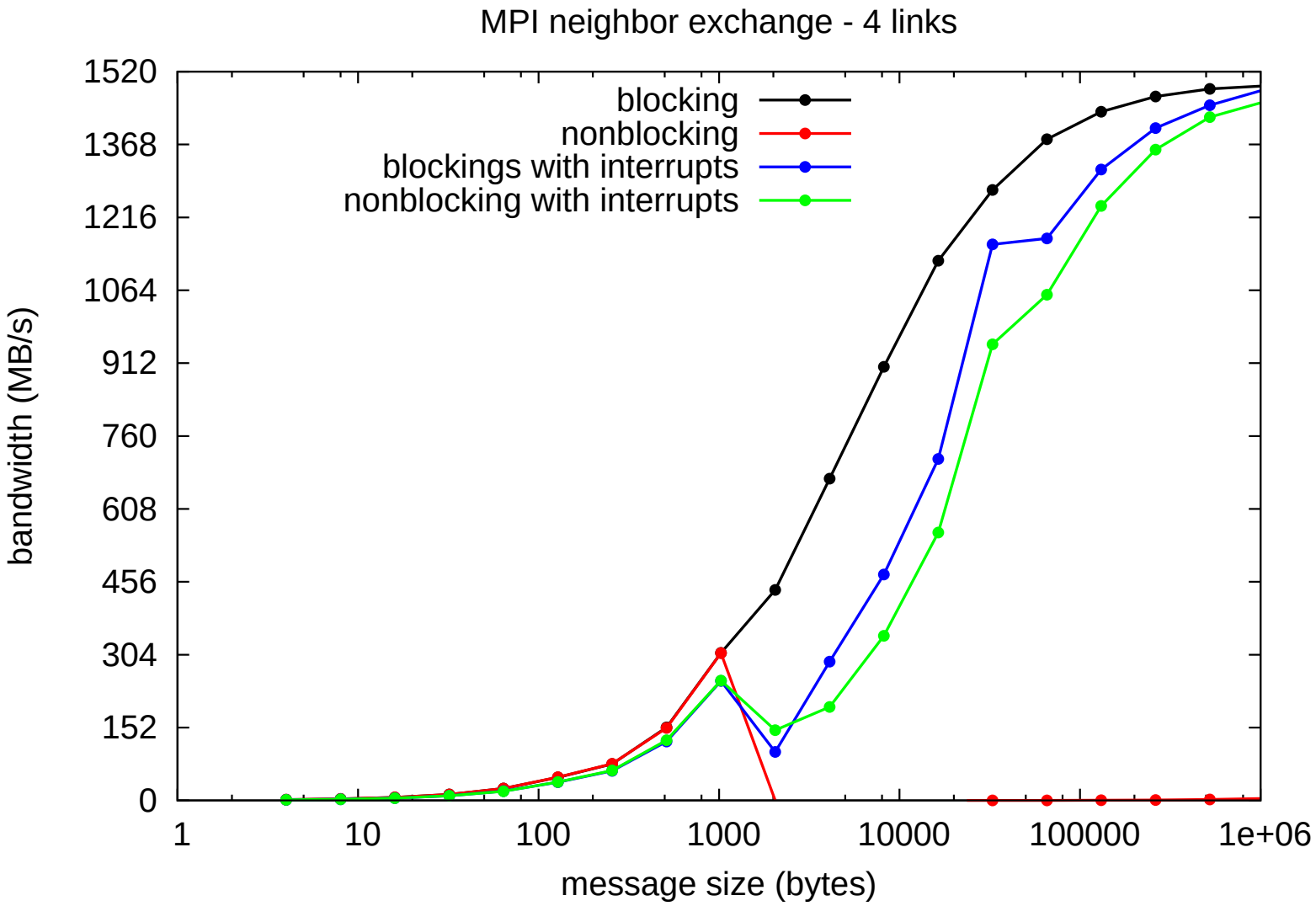
Neighbor exchange



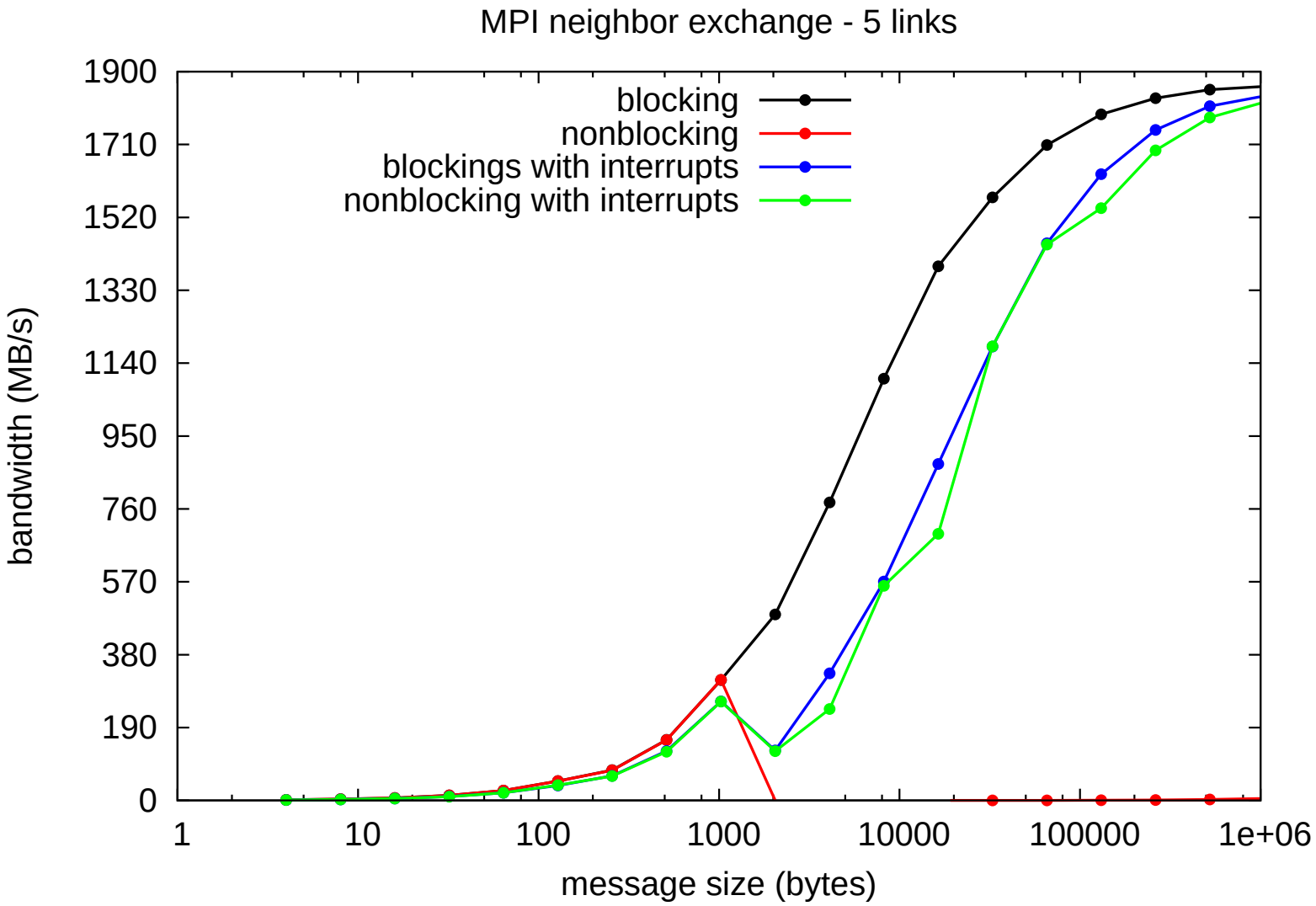
Neighbor exchange



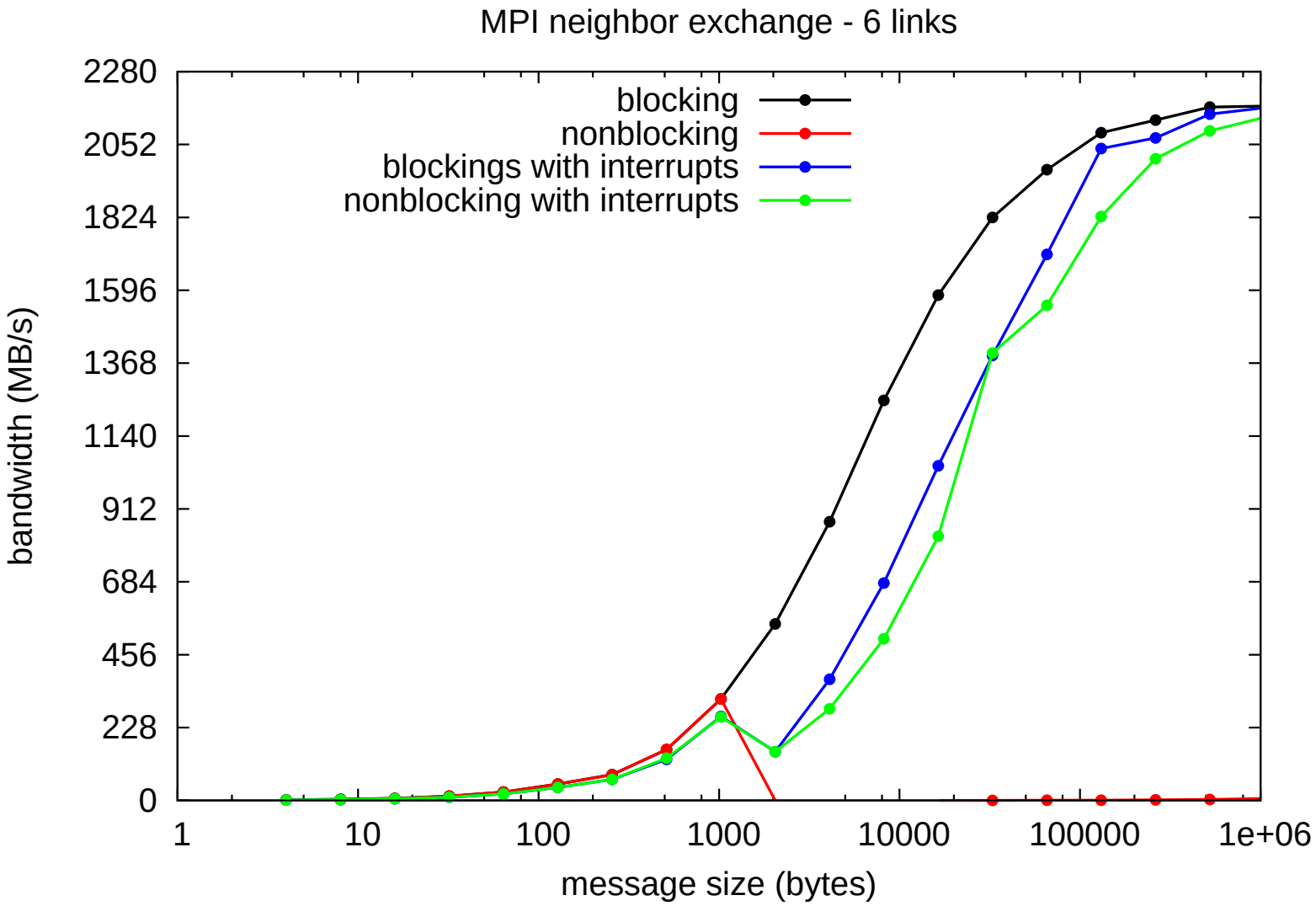
Neighbor exchange



Neighbor exchange



Neighbor exchange



Environment variables

DCMF Verbosity

DCMF_DMA_VERBOSE
Control the output of information associated with the Direct Memory Access messaging device. Specifically, it controls whether informational RAS events are generated when remote get resources become full and are increased in size. *Disabled by default.*

DCMF_STATISTICS
Turns on statistics printing for the message layer such as the maximum receive queue depth. *Disabled by default.*

DCMF_VERBOSE
Increases the amount of information dumped during an MPI_Abort() call. *Disabled by default.*

DCMF High-level tuning options

DCMF_INTERRUPTS

Turns on interrupt driven communications. This can be beneficial to some applications and is required if you are using asynchronous (e.g. one-sided) or irregular codes. *Disabled by default.*

If interrupts improve the performance of your code, you're doing a lot of MPI nonblocking send and/or recv that do not match in time. Interrupts introduce significant overhead in some cases and will *slow down* codes with well-structured communication.

DCMF_EAGER

Sets the cutoff for the switch to the rendezvous protocol. Switches from the eager protocol to the rendezvous protocol for point-to-point messaging. Increasing the limit might help for larger partitions and if most of the communication is nearest neighbor. *Default is 1200 bytes.*

The eager protocol does not require the receiver to post or be in an MPI call. It requires buffering and will cause problems for larger messages.

DCMF High-level tuning options

DCMF_OPTRVZ

Determines the optimized rendezvous limit. For sending, one of three protocols will be used depending on the message size: The eager protocol for small messages, the optimized rendezvous protocol for medium messages, and the default rendezvous protocol for large messages. The optimized rendezvous protocol generally has less latency than the default rendezvous protocol, but does not wait for a receive to be posted first. Therefore, unexpected messages in this size range might be received, consuming storage until the receives are issued. The default rendezvous protocol waits for a receive to be posted first. Therefore, no unexpected messages in this size range will be received. The optimized rendezvous protocol also avoids filling injection fifos which can cause delays while larger fifos are allocated, for example, alltoall on large subcommunicators with thread mode multiple will benefit from optimized rendezvous. *Default is 0 bytes (disabled).*

DCMF Low-level tuning options

DCMF_FIFOMODE

The fifo mode to use. This determines how many injection fifos are used by messaging and what they are used for:

- *DEFAULT:* Uses 22 injection fifos:
  - 6 normal fifos, each mapped to 1 torus fifo.
  - 6 remote get fifos, each mapped to 1 torus fifo.
  - 6 all-to-all fifos. These can inject into any of the torus fifos.
- RZVANY: Optimized for sending messages that use the rendezvous protocol. It has 6 more remote get fifos optimized for sending around corners:
  - 6 normal fifos, each mapped to 1 torus fifo.
  - 6 remote get fifos, each mapped to 1 torus fifo.
  - 6 remote get fifos, each mapped to all of the torus fifos.
  - 6 all-to-all fifos.
- ALLTOALL: Optimized for All-To-All communications. Same as DEFAULT, except there are 16 All-To-All fifos that can inject into any of the torus fifos:

DCMF Low-level tuning options

DCMF_RECFIFO
The size, in bytes, of each DMA reception FIFO. Incoming torus packets are stored in this fifo until DCMF Messaging can process them. Making this larger can reduce torus network congestion. Making this smaller leaves more memory available to the application. DCMF Messaging uses one reception FIFO. The value specified is rounded up to the nearest 32-byte boundary. *Default is 8388608 bytes (8 megabytes).*

If you see RAS events telling you to increase this variable, do not listen! Blowing up the DMA reception FIFOs is a sign of poor communication patters in your code. Using more memory just obscures the problem, but it is not a solution.

The *right way* to solve this problem is to restructure your code to do communication properly — preposted receives are essential, mixing MPI and computation more often is useful — and if that is too much work, enable interrupts.

DCMF Low-level tuning options

```
BE_MPI (ERROR): print_job_errtext() - last event:
APPL_0A2B A DMA unit reception FIFO is full.  Automatic
recovery occurs for this event, but performance might be
improved by increasing the FIFO size (via environment
variable DCMF_RECFIFO=size-in-bytes) or by checking the
DMA FIFOs more often.  Additional details:  1) torus
location is (3,0,1).  2) Packet PID is 0.  3) rFIFO bit
mask is 0b00000001.  4) Current fifo size is 8388608
bytes.
```

Two choices:

1. Increase the FIFO size.
2. Check the DMA FIFOs more often.

The right answer is (2). Besides restructuring communication, you can poke the network with MPI_Iprobe using an unused tag.

DCMF Collective options

DCMF_COLLECTIVES
Controls whether optimized collectives are used.
Options are 0, 1 (default) or NOTREE.

Bbefore disabling the tree altogether, see **DCMF_THREADED_TREE** and **DCMF_TREE_HELPER_THRESH**.

DCMF_BARRIER,BCAST,ALLTOALL,REDUCE,...
Controls whether optimized collectives are used.
Options are MPICH and many optimized protocols.

Setting DCMF_COLLECTIVES=0 or any specific collective to MPICH is a great way to slow your code down! There are more useful options in some cases...

DCMF Collective options

DCMF_ALLGATHER Controls the protocol used for allgather.

- **MPICH** Use the unoptimized MPICH point-to-point protocol.
- **ALLREDUCE** Use collective network allreduce. Default on MCW for smaller messages.
- **ALLTOALL** Use all-to-all. Default on irregular communicators. It works very well for larger messages.
- **BCAST** Use bcast. Default for larger messages on MCW. This can work well on rectangular subcommunicators for smaller messages.
- **ASYNC** Use async bcast. This will use asynchronous broadcasts to do the allgather. This is a good option for small messages on rectangular or irregular subcommunicators.

It is not clear what the best algorithm is, but IBM has done *some* work to figure it out. MPI_Allgather is a case where the defaults are not always optimal and one has to switch the defaults (Nick Romero can show you how) or by tricking MPI (by changing datatype).

DCMF Collective options

DCMF_SAFEALLGATHER,SAFEALLGATHERV
DCMF_SAFEBCAST,SAFESCATTERV,SAFEALLREDUCE
The direct put bandwidth optimization protocols require the send/recv buffers to be 16-byte aligned on all nodes. Unfortunately, you can have root's buffer be misaligned from the rest of the nodes. Therefore, by default we must do an allreduce before the dput protocol to ensure all nodes have the same alignment. If you know all of your buffers are 16 byte aligned, turning on this option will skip the allreduce step and improve performance. If the application uses well-behaved datatypes, you can set this option to skip over the allreduce. This is most useful in irregular subcommunicators where the allreduce can be expensive.

Enabled (N) by default. Use Y to bypass the allreduce, but mismatched alignment will lead to undefined and/or erroneous results.

Optimizing collectives through thinking

MPI_Reduce_scatter: reduce a buffer to root, then scatter from root. This MPI-2.1 function requires vector arguments, so this is really reduce, then scatterv. MPI_Scatterv is not optimized. In addition, the arguments to scatterv must be allocated internally. At scale, this consumes a lot of memory (perhaps as much as 8x).

MPI_Reduce_scatter_block: reduce a buffer to root, then scatter from root. This MPI-2.2 function takes scalar arguments and therefore uses less memory at scale. Blue Gene/P only provides MPI 2.1.

MPI_Allreduce: reduce a buffer everywhere, then copy out my portion. This MPI-1 function requires scalar arguments and is highly optimized on Blue Gene/P. Using MPI_IN_PLACE means no extra memory is used (although the input buffer is modified).

At least on regular communicators on BGP, allreduce+copy is much faster than reduce+scatter. On BGQ, allreduce will be faster than reduce...

Time for an oxygen tank

DCMF Sample program

Both Initialize MPI (DCMF internally) and setup message protocols.

This is what happens when the message is small enough to be buffered at the receiver (EAGER protocol).

Sender	**Receiver**
DCMF_Send to **R**	—
(Wait on local completion)	—
Local completion callback	—
—	Short message callback
—	Process buffer
—	DCMF_Control to **S**
(Wait on remote completion)	
Remote completion callback	

DCMF Sample program

Larger messages require buffering (RNDZV protocol).

Sender	**Receiver**
DCMF_Send to **R**	—
(Wait on local completion)	—
Local completion callback	—
—	Long message callback
—	Allocate buffer for transfer
—	⟨DMA⟩ (`rget`)
—	Cleanup callback
—	Process buffer
—	DCMF_Control to **S**
(Wait on remote completion)	
Remove completion callback	

Look at example code

DCMF Sample program

```
0:  sending 1 bytes
0:  before DCMF_Send
0:  local_completion_cb
0:  after DCMF_Send
0:  after local completion
0:  after remote completion
0:  remote_completion_cb peer=1
1:  default_short_cb peer=0 count=0
1:  src[0] = X
```

```
0:  sending 200000 bytes
0:  before DCMF_Send
0:  after DCMF_Send
0:  local_completion_cb
0:  after local completion
0:  after remote completion
0:  remote_completion_cb peer=1
1:  default_long_cb peer=0 count=0
1:  default_long_cleanup_cb peer=0
1:  recv_buffer[0] = X
1:  recv_buffer[1] = X
.............................
1:  recv_buffer[199998] = X
1:  recv_buffer[199999] = X
```

Closing thoughts

"People first, then money, then things" – Suze Orman

"Science first, then algorithms, then communication" – Me

However, you know your algorithms are good and communication is holding you back, understanding the internals of MPI will help you write faster and more scalable code on Blue Gene/P.

P.S. If profiling shows you spend too much time in MPI_Barrier, your *algorithm* is the problem (load imbalance), not communication.