COMP 605: Introduction to Parallel Computing
Topic: MPI: Communication Performance

Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

Posted: 03/06/17
Updated: 03/01/17

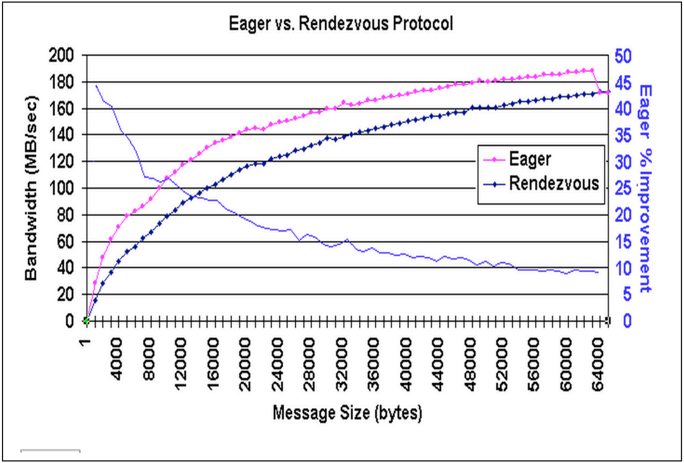
Table of Contents

Factors Affecting MPI Communication Performance

- CPU/Processors:
  - Number of processors involved in the communication
  - Type of processor (speed, memory)
  - Software stack (including OS)
- Cluster Network Architecture:
  - Type/topology: `http://en.wikipedia.org/wiki/Network_topology`
  - Hardware design: Ethernet, Myrinet, WiFi
  - Protocols/Transport layer: TCP/IP, infiniband, `http://en.wikipedia.org/wiki/Lists_of_network_protocols`
- MPI Message Passing Protocols
- MPI Messages

MPI Message Passing Protocols

- MPI Protocol describes the internal methods and policies used to send messages.
- *Eager:* asynchronous protocol that allows a send operation to complete without acknowledgement from a matching receive
  - Sending process assumes receiving process can store message
  - Generally used for smaller message sizes (up to Kbytes).
  - Reduces synch. delays and simplifies programming.
  - not scalable: buffer "wastage"; program crash if data bigger than buffer
- *Rendezvous:* synchronous protocol; requires acknowledgement from a matching receive in order for the send operation to complete.
  - Requires some type of "handshaking" between the sender and the receiver processes
  - More scalable: robustness - prevents memory exhaustion and termination; only buffer small message envelopes; reduces data copy.
  - problem with synchronization delays; more programming complexity



Timings for Eager vs Rendevouz protocols

REF: `https://computing.llnl.gov/tutorials/mpi_performance/`

MPI Messages

- Characteristics
  - Message size (KBytes, MBytes, GBytes,) and buffering (GBytes/sec)
  - Number of other messages being sent
  - Where/how data is stored between the time a send operation begins and when the matching receive operation completes.
  - Larger messages tend to have better performance.
- Performance function of:
  - the number of words being sent
  - machine precision (32, 64 bit)
  - data type (int, long int, float, double)
- Performance measurement:
  - Calculate the time needed for a communication to start and send a message of known size.
  - Perform "warmup" events first: MPI implementation may use "lazy" semantics to setup and maintain streams of communications ⇒ the first few events may take significantly longer than subsequent events.
- Speedup and Efficiency are relevant as well.

Total Parallel Run-Time

- The total parallel program run time is a function of a large number of variables: number of processing elements (PEs); communication; hardware (cpu, memory, software, network), and the program being run (algorithm, problem size, # Tasks, complexity, data distribution); parallel libraries:

$$T = \mathcal{F}\left(PEs, N, \; Tasks, \; I/O, \; Communication, \ldots\right)$$

- The execution time required to run a problem of size N on processor i, is a function of the time spent in different parts of the program (computation, communication, I/O, idle):

$$T^i = T^i_{comp} + T^i_{comm} + T^i_{io} + T^i_{idle}$$

- The total time is the sum of the times over all processes averaged over the number of the processors: $T =$

$$\frac{1}{p}\left(\sum_{i=0}^{p-1} T_{comp} + \sum_{i=0}^{p-1} T_{comm} + \sum_{i=0}^{p-1} T_{io} + \sum_{i=0}^{p-1} T_{idel}\right)$$

The message passing communication time required to send N words (or Bytes):

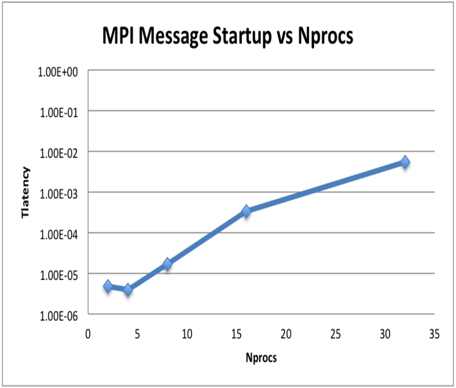
$$T_{comm} = t_{startup} + t_{bw}$$

Where:

- $t_{startup}$ is the message startup time (or latency)
  - Time required to set up communications on the nodes and to prepare them to send a message.
  - Estimated to be *half of the time* of a *ping-pong* operation with a message of size zero.
- t_{bw} is the message passing saturation bandwidth (BW).
  - Peak rate at which data packets can be sent across the network.
- Popular ways to measure:
  - *Ping-Pong*: measures communication between two PEs as function of message size.
  - *Ring*: measures communication between multiple PEs as a function of message size.
  - Can be used to test point-to-point or collective communications.

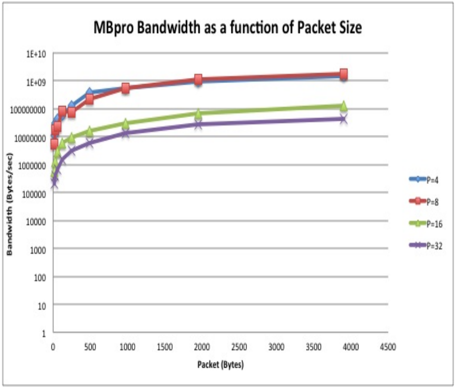
MPI Latency or Startup Time

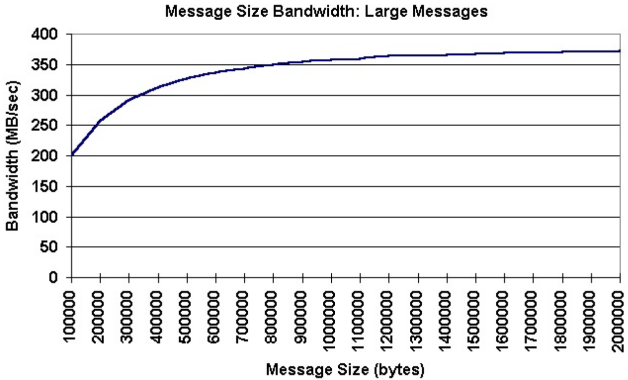
- **Message latency:** the time required to set up communications on the PEs and to prepare them to send a message.
- A function of the number and size of messages that need to be sent, and the number of PEs communicating.
- MPI latency is usually estimated to be 1/2 the time of a *"ping-pong"* operation with a message of size zero.
- In *ping-pong*, packets of information are exchanged between two PEs and the time required to do this is measured.
- Important when working with very fine-grained applications which have more frequent communication requirements.



MPI Message Bandwidth

- **Bandwidth:** Peak rate at which data packets can be sent across the network.
- Bandwidth is relevant for coarse-grained codes that send fewer messages, but typically need to communicate larger amounts of data.
- The bandwidth can be estimated using *ping-pong* and *ring* programs.
- Packets of information consist of an array of dummy integer or floating point numbers that vary in length.
- Code run-time is measured as a function of number of PE's (cores), and message size (number of Bytes).





Source: `https://computing.llnl.gov/tutorials/mpi_performance`

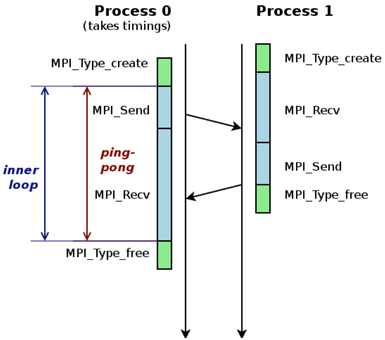
Communication Performance

- PingPong:
  - Two processes send packets of information back and forth a number of times
  - Compute average amount of time per message and transfer rate (bandwidth) as function of message size.
- Ring
  - Processes send packets of information to neighbor
  - Simple ordering: P0 to P1, P1-P2, ... Pn-1 to P0.
  - Measure time required to send message to all PE's as function of message size and the number of PEs.

Timing MPI Messages - Ping-Pong Algorithm

TimeStep	P_0	P_1
t_0	MPI_Send message to P_1	WAITS for message from P_0
t_1	WAITS for message from P_1	MPI_Recv message from P_0
t_2	WAITS for message from P_1	MPI_Send message to P_0
t_3	MPI_Recv message from P_0	

System has $sz = comm\_sz = 2$
Processors numbered $[P_1, P_2]$



Img source: http://htor.inf.ethz.ch/research/datatypes/ddtbench/benchmark_expl.png

MPI Ping-Pong Code

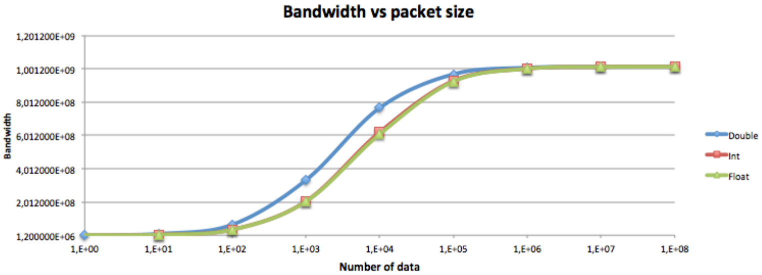
```
/* ping_pong.c -- two-process ping-pong -- send from 0 to 1
 *  and send back   from 1 to 0
 * See Chap 12, pp. 267 & ff. in PPMPI */

#include <stdio.h>
#include "mpi.h"
#define MAX_ORDER 100
#define MAX 2
main(int argc, char* argv[]) {
int   p,my_rank, min_size = 0,max_size = 16;
int   incr = 8, size,pass;
float  x[MAX_ORDER];
int         i;
double      wtime_overhead;
double      start, finish;
double      raw_time;
MPI_Status  status;
MPI_Comm    comm;

/* startup the MPI environment */
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
MPI_Comm_dup(MPI_COMM_WORLD, &comm);

wtime_overhead = 0.0;
for (i = 0; i < 100; i++) {
   start = MPI_Wtime();
   finish = MPI_Wtime();
   wtime_overhead = wtime_overhead + (start - finish);
}
wtime_overhead = wtime_overhead/100.0;
```

```
if (my_rank == 0) {
   for (size=min_size;size<=max_size; size=size+incr {
      for (pass = 0; pass < MAX; pass++) {
         MPI_Barrier(comm);
         start = MPI_Wtime();
         MPI_Send(x, size, MPI_FLOAT,1,0,comm);
         MPI_Recv(x, size, MPI_FLOAT,1,0,comm,&status);
         finish = MPI_Wtime();
         raw_time = finish - start - wtime_overhead;
         printf("%d %f\n", size, raw_time);
      }
   }
} else { /* my_rank == 1 */
   for (size=min_size;size<=max_size; size=size+incr {
      for (pass = 0; pass < MAX; pass++) {
         MPI_Barrier(comm);
         MPI_Recv(x, size, MPI_FLOAT,0,0,comm,&status);
         MPI_Send(x, size, MPI_FLOAT, 0, 0, comm);
      }
   }
}
MPI_Finalize();
}  /* main */
```

Timing MPI Messages: Ping-Pong Output

```
#############
# RUN USING MPICH on OS X
#############
[gidget]% mpirun -np 2 ./ping_pong
MAX_ORDER=100
0 0.000005
0 0.000001
8 0.000009
8 0.000001
16 0.000001
16 0.000005

[gidget]% mpirun -np 2 ./ping_pong
MAX_ORDER=10000
0 0.000007
0 0.000018
8 0.000002
8 0.000007
16 0.000001
16 0.000001

[gidget]% mpirun -np 2 ./ping_pong
MAX_ORDER=1000000
0 0.000005
0 0.000011
8 0.000001
8 0.000001
16 0.000001
16 0.000006
```

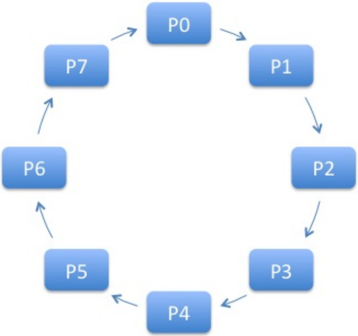
```
#############
# RUN USING %20.16f output
############
[gidget]% mpirun -np 2 ./ping_pong
MAX_ORDER=1000
0    0.0000049583311193
0    0.0000007883342914
8    0.0000138283637352
8    0.0000008103367873
16    0.0000007943296805
16    0.0000009803031571

[gidget]% mpirun -np 2 ./ping_pong
MAX_ORDER=1000000
0    0.0000058855797397
0    0.0000010205834405
8    0.0000014185492182
8    0.0000012685480760
16    0.0000011545774760
16    0.0000009415956447
```



Source: COMP605 Student, J. Ayoub, Spring, 2014

Timing MPI Messages - Ring Algorithm

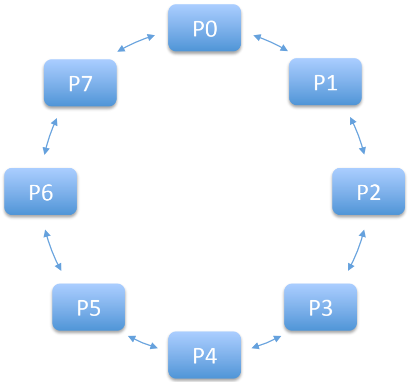
- System has $sz = comm\_sz$ processors numbered:
  $P_0, P_1, , P_{r-1}, P_r, P_{r+1}, ..P_{sz-1}$
- P_0 *sends* msg to P_1
  P_0 *waits* for msg from P_{sz-1}
  ...
  P_r *waits* for msg from P_{r-1}
  P_r *rcvs* msg, *sends* msg to P_{r+1}
  ...
  P_{sz-1} *sends* to P_0
  P_{sz-1} *waits* for msg from P_{sz-2}



8 Processors arranged in a ring

Timing MPI Messages - Ring Exchange

- System has $sz = comm_sz$ processors numbered
- **Step 0**: Each P_i creates unique msg.
- **Step 1**: P_i gets msg from lower nor, P_{i-1}, and sends its msg to upper nbr, P_{i+1}.
- **Step 2**: P_i gets msg from upper nbr, P_{i+1}, and sends its' msg to lower nbr, P_{i-1}.
- Code is done when all messages have been exchanged between each processor and its' neighbor.



Timing MPI Messages: pach_ring.c

```
 /*MPI ring message passing program
 * takes a single command line option: the maximum message
 * size in number of bytes
 * the program converts the number of bytes you specify
 * into numbers of doubles based on the byte size of a
 * double on that system. Then it starts with a message
 * of one double and scales by 2 until it reaches that
 * number, spitting out timing all along the way
 */

#include "stdlib.h"
#include "mpi.h"

/* if you want a larger number of runs to be averaged
#define ITERATIONS 1000
** together, increase INTERATIONS */
#define WARMUP 8

int main(int argc, char **argv)
{

   int i, j, rank, size, tag=96,bytesize, dblsize;
   int max_msg, min_msg, packetsize;
   int iterations;

   double *mess;
   double tend, tstart, tadd, bandwidth;
   MPI_Status status;

   MPI_Init(&argc, &argv);

   MPI_Comm_size(MPI_COMM_WORLD,&size);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```

```
/* get the message size from the command line */
 if(rank == 0)
 {
    printf("argcnt= %d\n",argc);
    dblsize = sizeof(double);

    if( argc >= 2 )
       max_msg = atoi(argv[1]);
    else
       max_msg = 4096;

    if( argc >= 3 )
       min_msg = atoi(argv[2]);
    else
       min_msg = 0;

   if( argc >= 4 )
       iterations = atoi(argv[3]);
    else
       iterations = 10;
```

Timing MPI Messages: pach_ring.c

```
 printf("ring size is %i nodes\n", size);
   printf("max message specified= %i\n", max_msg);
   printf("min message specified= %i\n", min_msg);
   printf("iterations =           %i\n", iterations);
   bytesize = max_msg;
   printf("double size is %i bytes\n", dblsize);
   max_msg = max_msg/dblsize;
   if(max_msg <= 0) max_msg = 1;
   printf("#of doubles being sent is %i\n", max_msg);

   printf("PacketLength\tBandwidth\tPacketTime\n");
   printf(" (MBytes)   \t (B/sec) \t(sec)\n");
   printf("------------ -------------- --------------\n");
}

/* pass out the size to the kids */
MPI_Bcast(&max_msg, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&min_msg, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&iterations, 1, MPI_INT, 0, MPI_COMM_WORLD);

/* make the room for the largest sized message */
mess = (double*)malloc(max_msg * (sizeof(double)));
if(mess == NULL)
{
   printf("malloc prob, exiting\n");
   MPI_Finalize();
}
```

```
/* warmup lap */
 for(packetsize = 0; packetsize < WARMUP; packetsize++)
 {
    /* head node special case */
    if(rank == 0)
    {
       MPI_Send(mess, max_msg, MPI_DOUBLE, 1, tag, MPI_COMM_WORL
       MPI_Recv(mess, max_msg, MPI_DOUBLE, size-1,tag,
       MPI_COMM_WORLD, &status);
    }
    /* general case */
    if((rank != 0) && (rank != (size-1)))
    {
       MPI_Recv(mess, max_msg, MPI_DOUBLE, rank-1,tag,
       MPI_COMM_WORLD, &status);
       MPI_Send(mess, max_msg, MPI_DOUBLE, rank +1,tag,
       MPI_COMM_WORLD);
    }
    /* end node case */
    if(rank == size-1)
    {
       MPI_Recv(mess, max_msg, MPI_DOUBLE, rank-1,tag,
       MPI_COMM_WORLD, &status);
       MPI_Send(mess, max_msg, MPI_DOUBLE, 0,tag, MPI_COMM_WORLD
    }
 }
 /* end warmup lap */
 /*
 if(rank == 0)
 printf("warmup lap done\n");
 */
```

Timing MPI Messages: pach_ring.c

```
/* real timed stuff now */
for(packetsize = min_msg; packetsize <= max_msg; packetsize*=2)
{
   if(rank == 0)
      printf("Starting packetsize: %i\n",packetsize);
   /* init timing variables */
   tadd = 0.0;
   tend = 0.0;
   tstart = 0.0;

   for(j = 0; j < iterations; j++)
   {
      MPI_Barrier(MPI_COMM_WORLD);
      if(rank == 0)
      {
         tstart = MPI_Wtime(); /* timing call */
         MPI_Send(mess, packetsize, MPI_DOUBLE, 1, tag,
         MPI_COMM_WORLD);
         MPI_Recv(mess, packetsize, MPI_DOUBLE, size-1,tag,
         MPI_COMM_WORLD, &status);

         tend = MPI_Wtime();
         tadd += (tend - tstart);
      if( j%20 == 0 )
         printf("deltaT[%i]= %i\n",j,tend-tstart);
      }
               /* general case */
      if((rank != 0) && (rank != (size-1)))
      {
         MPI_Recv(mess, packetsize, MPI_DOUBLE, rank-1,tag,
         MPI_COMM_WORLD, &status);
         MPI_Send(mess, packetsize, MPI_DOUBLE, rank +1,tag,
         MPI_COMM_WORLD);
      }
         /* end node case */
         if(rank == size-1)
         {
            MPI_Recv(mess, packetsize, MPI_DOUBLE, rank-1,tag,
            MPI_COMM_WORLD, &status);
            MPI_Send(mess, packetsize, MPI_DOUBLE, 0,tag,
            MPI_COMM_WORLD);
         }

      }
      /* calc and print out the results */
      if(rank == 0)
      {
         bandwidth = ((size * packetsize *dblsize)/
                                 (tadd/(double)iterations));
         printf("RESULTS: %16.12lf \t%20.8lf \t%16.14lf \n",
                 (double)(packetsize * dblsize)/1048576.0,
                 bandwidth,
                 tadd/(double)iterations);
      }
      /* to make it possible to do a 0 size message */
      if (packetsize == 0) packetsize = 1;
   }
   /* end real timed stuff */

    if( rank == 0 ) printf("\nRing Test Complete\n\n");
   MPI_Finalize();
   exit(1);

} /* end ring.c */
```

Timing MPI Messages: pach_ring.c

```
[mthomas@tuckoo ring]$ mpirun -np 4 ./pach-ring
ring size is 4 nodes
max message specified= 4096,  min message specified= 0
iterations =          10
double size is 8 bytes,   #of doubles being sent is 512
PacketLength Bandwidth PacketTime
 (MBytes)     (B/sec)  (sec)
------------ -------------- --------------
Starting packetsize: 0  deltaT[0]= 0
RESULTS:    0.000000000000            0.00000000  0.00000300407410
Starting packetsize: 2  deltaT[0]= 0
RESULTS:    0.000015258789      13908572.84974093  0.00000460147858
Starting packetsize: 4  deltaT[0]= 0
RESULTS:    0.000030517578      14202934.17989418  0.00000901222229
Starting packetsize: 8  deltaT[0]= 0
RESULTS:    0.000061035156      61709300.22988506  0.00000414848328
Starting packetsize: 16  deltaT[0]= 0
RESULTS:    0.000122070312     138547332.12903225  0.00000369548798
Starting packetsize: 32  deltaT[0]= 0
RESULTS:    0.000244140625     258732969.63855419  0.00000395774841
Starting packetsize: 64  deltaT[0]= 0
RESULTS:    0.000488281250     445074331.19170982  0.00000460147858
Starting packetsize: 128  deltaT[0]= 0
RESULTS:    0.000976562500     885560267.21649492  0.00000462532043
Starting packetsize: 256  deltaT[0]= 0
RESULTS:    0.001953125000    1347440720.31372547  0.00000607967377
Starting packetsize: 512  deltaT[0]= 0
RESULTS:    0.003906250000    1391082525.02024293  0.00001177787781
Ring Test Complete
```

Comments: Calculating BW

Calculating BW:

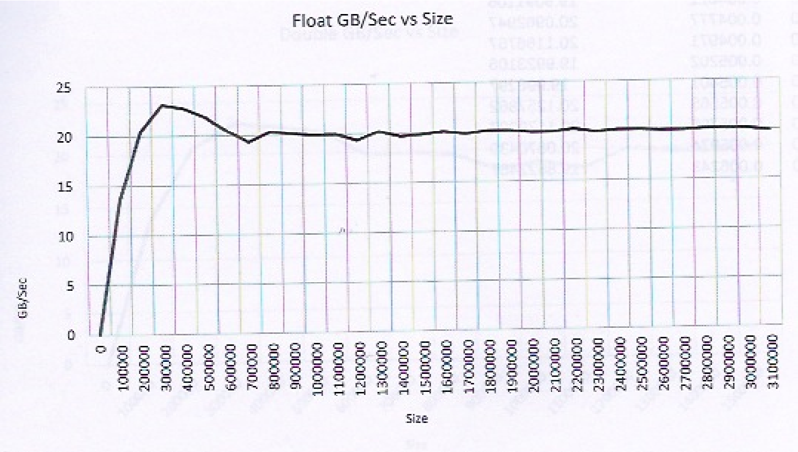
- BW units typically Mega or Giga Bytes per second, e.g., GByte/sec
- Estimate packet size per send or recv
- Count the number of sends or recvs you are using
- are you calculating BITS/sec, or BYTES/second? Convert packet size accordingly
- Example estimation: Ping-pong:

$$BW\left[\frac{a}{b}\right] \cong \frac{(\#exchanges) * packetSize[floats] * size[1float]}{rawTime[\mu sec]}$$

$$\cong \frac{[2] * 10^6[floats] * 32[bits/float]}{3x10^{-3}[seconds]}$$

$$\cong 21x10^9 \frac{bits}{second} * \frac{1Byte}{8bits}$$

$$\cong 2.67x10^9 \frac{GBytes}{second}$$



Source: COMP605 Student, D. Biscane, Spring, 2014