Distributed Programming with MPI

Abhishek Somani, Debdeep Mukhopadhyay

Mentor Graphics, IIT Kharagpur

November 12, 2016

Overview

Outline

Programming Model

- MPI - Message Passing Interface
- Single Program Multiple Data (SPMD)
- Each process has its own (unshared) memory space
- Explicit communication between processes is the only way to exchange data and information
- Contrast with OpenMP

MPI program essentials

```c
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

int main(const int argc, char ** argv)
{
    int myRank, commSize;
    //Initialize MPI runtime environment
    MPI_Init(&argc, &argv);
    //Know the total number of processes in MPI_COMM_WORLD
    MPI_Comm_size(MPI_COMM_WORLD, &commSize);
    //Know the rank of the process
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    ...
    //Clean up and terminate MPI environment
    MPI_Finalize();
    return 0;
}
```

MPI Hello World

```
int main(const int argc, char ** argv)
{
    int myRank, commSize;
    //Initialize MPI runtime environment
    MPI_Init(&argc, &argv);
    //Know the total number of processes in MPI_COMM_WORLD
    MPI_Comm_size(MPI_COMM_WORLD, &commSize);
    //Know the rank of the process
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
    //Say hello
    printf("Hello from process %d out of %d processes\n",
        myRank, commSize);
    //Clean up and terminate MPI environment
    MPI_Finalize();
    return 0;
}
```

Preliminaries for running MPI programs

- MPI cluster has been set up consisting of 4 nodes : 10.5.18.101, 10.5.18.102, 10.5.18.103, 10.5.18.104
- Set up password-free communication between servers
  - RSA key based communication between hosts
  - cd
  - mkdir .ssh
  - ssh-keygen -t rsa -b 4096
  - cd .ssh
  - cp id_rsa.pub authorized_keys

Compiling and running MPI programs

- Create a file containing host names, each host in a different line

```
10.5.18.101
10.5.18.102
10.5.18.103
10.5.18.104
```

- Compiling : Use **mpicc** instead of gcc / cc
  - mpicc is a wrapper script containing details of location of necessary header files and libraries to be linked
  - Part of MPI installation
  - mpicc mpi_helloworld.c -o mpi_helloworld
- Running the program : Use **mpirun**
  - mpirun -hostfile hosts -np 4 ./mpi_helloworld

Outline

Send and Receive

```
int MPI_Send(const void *buf, //initial address of send buffer
             int count, //number of elements in send buffer
             MPI_Datatype datatype, //datatype of each send buffer
                 element
             int dest, //rank of destination
             int tag, //message tag
             MPI_Comm comm); //communicator

int MPI_Recv(void *buf, //initial address of receive buffer
             int count, //maximum number of elements in receive
                 buffer
             MPI_Datatype datatype, //datatype of each receive
                 buffer element
             int source, //rank of source
             int tag, //message tag
             MPI_Comm comm, //communicator
             MPI_Status *status); //status object
```

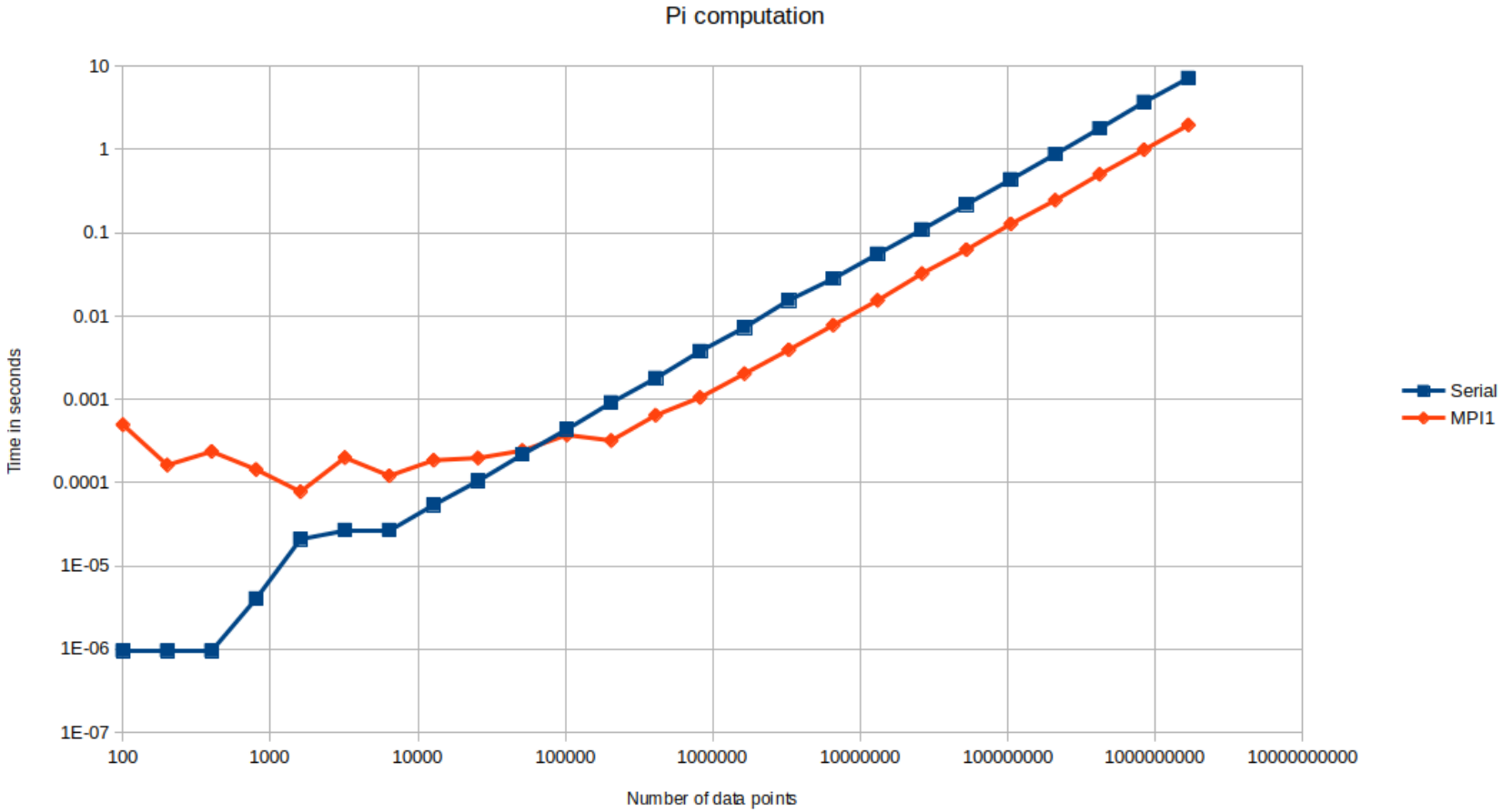
π once again

```
int main(const int argc, char ** argv)
{
    const int numTotalPoints = (argc < 2 ? 1000000 : atoi(argv[1]));
    const double deltaX = 1.0/(double)numTotalPoints;

    const double startTime = getWallTime();
    double pi = 0.0;
    double xi = 0.5 * deltaX;
    for(int i = 0; i < numTotalPoints; ++i)
    {
        pi += 4.0/(1.0 + xi * xi);
        xi += deltaX;
    }
    pi *= deltaX;
    const double stopTime = getWallTime();
    printf("%d\t%g\n", numTotalPoints, (stopTime-startTime));
    //printf("Value of pi : %.10g\n", pi);

    return 0;
}
```

π with MPI

```
double localPi = 0.0;
double xi = (0.5 + numPoints * myRank) * deltaX;
for(int i = 0; i < numPoints; ++i)
{
    localPi += 4.0/(1.0 + xi * xi);
    xi += deltaX;
}
if(myRank != 0)
{
    MPI_Send(&localPi, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
}
else
{
    double pi = localPi;
    for(int neighbor = 1; neighbor < commSize; ++neighbor)
    {
        MPI_Recv(&localPi, 1, MPI_DOUBLE, neighbor, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
        pi += localPi;
    }
    pi *= deltaX;
    //printf("Value of pi : %.10g\n", pi);
}
```

π with MPI : Performance

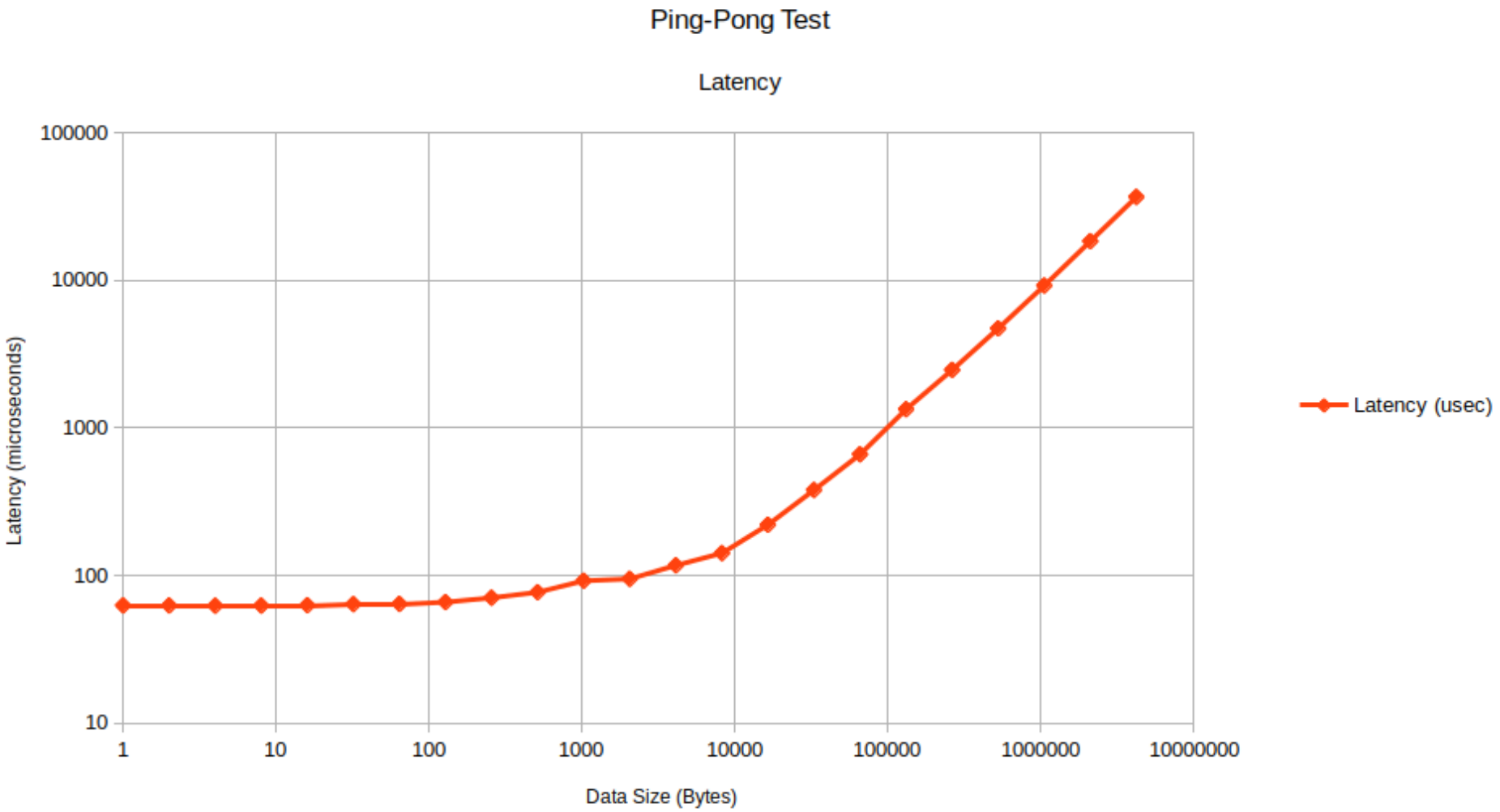


- What happened when number of data points were less than 10^5 ?

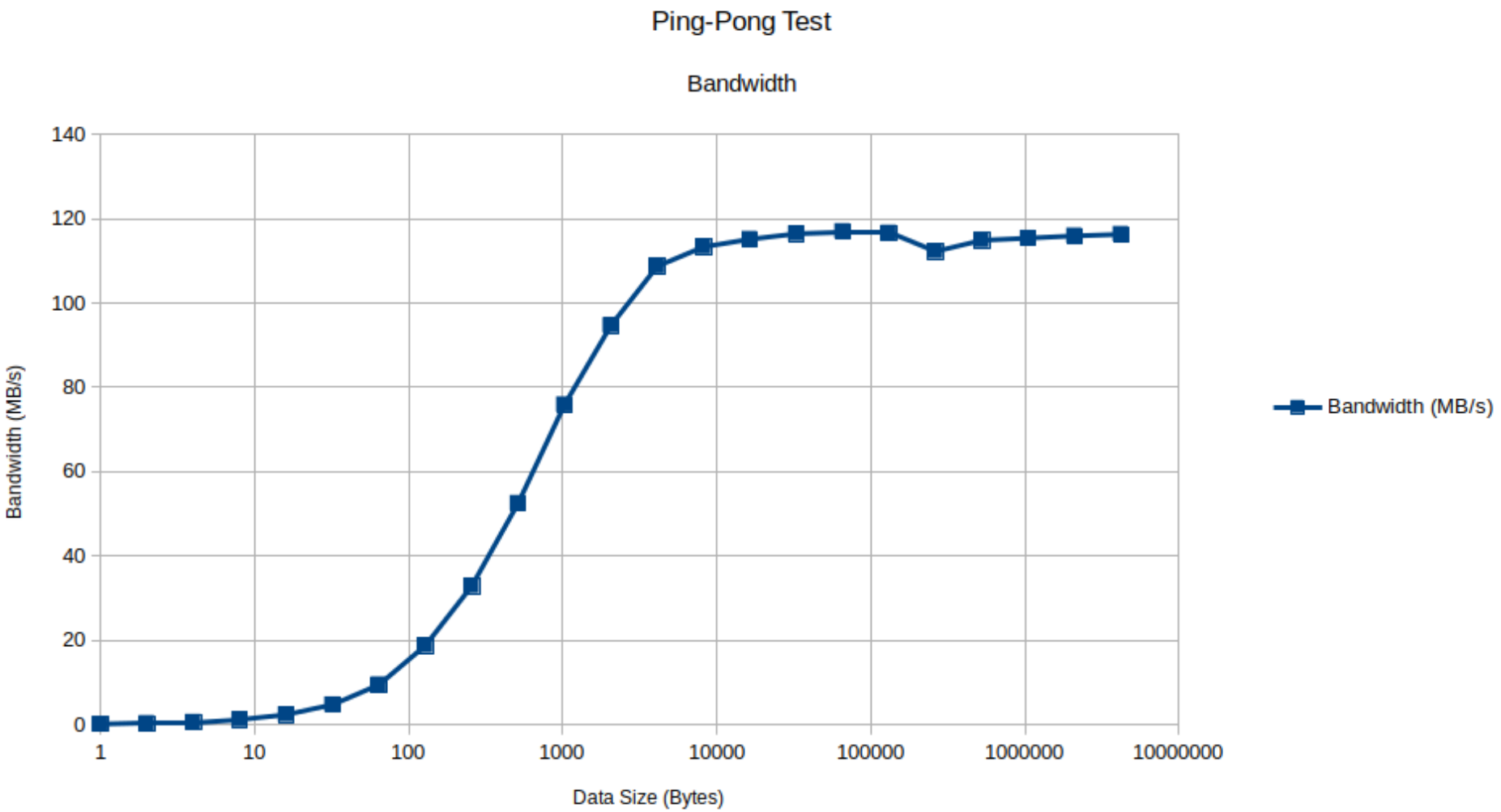
Ping-Pong Benchmark

- Point-to-point communication between 2 nodes, **A** and **B**
- Send message of size **n** from **A** to **B**
- Upon receiving message, **B** sends back the message to **A**
- Record the time taken **t** for the entire process
- Observe **t** for different values of **n** ranging from very small to very large

Ping-Pong Benchmark : Latency



Ping-Pong Benchmark : Bandwidth



π with MPI : Rough Performance Analysis

- Time taken in each loop iteration : α
- Minimum latency : λ
- Assume perfect scaling with p nodes
- MPI Parallel program can be faster only when $\lambda + \frac{\alpha n}{p} \leq \alpha n$, i.e., $n \geq \frac{\lambda p}{\alpha(p-1)}$
- Here, $p = 4$, $\lambda \approx 100 \mu sec$
- Every loop does 4 additions ($\sim$1 clock cycle each), 1 multiplication ($\sim$4 clock cycles) and 1 division ($\sim$4 clock cycles)
- Assume pipelining and superscalarity boost performance of the simple loop by $\sim$3x
- Server clock frequency : $3.2 GHz$
- $\alpha = \frac{12}{3 \times 3.2 \times 10^9}$, i.e., $\alpha \approx 0.00125 \mu sec$
- $n \geq \frac{100 \times 4}{0.00125 \times 3}$, i.e., $n \geq 1.06 \times 10^5$

Ring shift

```
//Make a ring
const int left = (myRank == 0 ? commSize-1 : myRank-1);
const int right = (myRank == commSize-1 ? 0 : myRank + 1);

//Create parcels
const int parcelSize = 10000;
int * leftParcel = (int *) malloc(parcelSize * sizeof(int));
int * rightParcel = (int *) malloc(parcelSize * sizeof(int));

//Send and Receive
MPI_Recv(leftParcel, parcelSize, MPI_INT, left, 0, MPI_COMM_WORLD,
       MPI_STATUS_IGNORE);
printf("Received parcel at process %d from %d\n", myRank, left);
MPI_Send(rightParcel, parcelSize, MPI_INT, right, 0, MPI_COMM_WORLD);
printf("Sent parcel from process %d to %d\n", myRank, right);
```

- MPI_Recv and MPI_Send are blocking functions
- Trying to receive before sending causes DEADLOCK

Idealized Communication

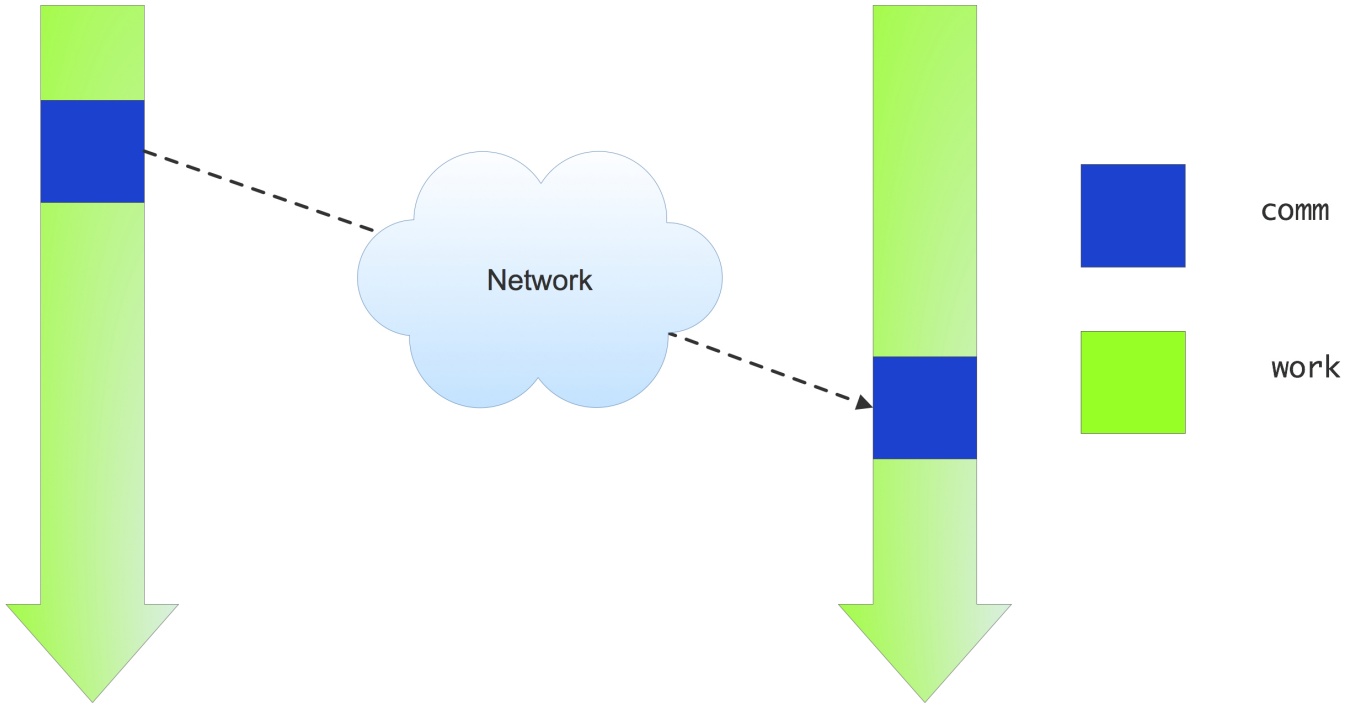


Figure : Courtesy of Victor Eijkhout

Actual Communication

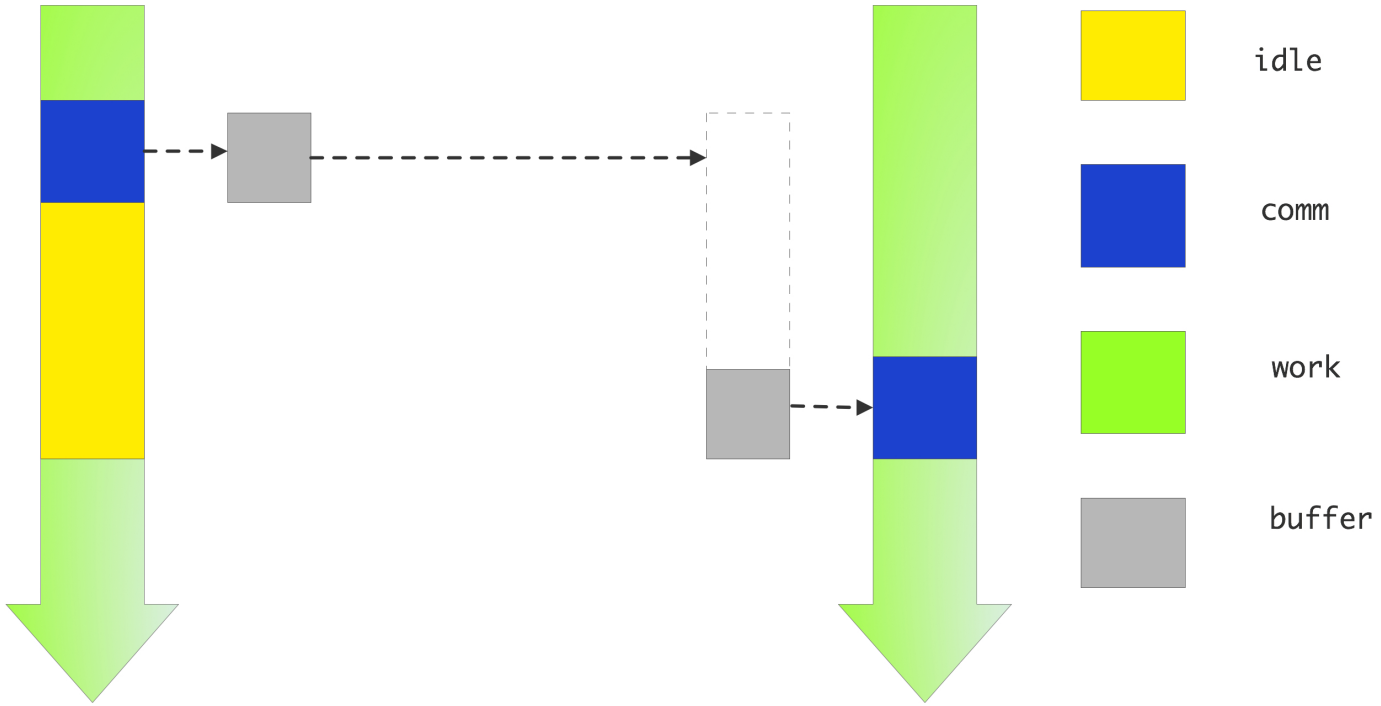


Figure : Courtesy of Victor Eijkhout

Ring shift : Send before Receive

```
//Make a ring
const int left = (myRank == 0 ? commSize-1 : myRank-1);
const int right = (myRank == commSize-1 ? 0 : myRank + 1);

//Create parcels
const int parcelSize = (argc < 2 ? 100000 : atoi(argv[1]));
int * leftParcel = (int *) malloc(parcelSize * sizeof(int));
int * rightParcel = (int *) malloc(parcelSize * sizeof(int));

//Send and Receive
MPI_Send(rightParcel, parcelSize, MPI_INT, right, 0, MPI_COMM_WORLD);
printf("Sent parcel from process %d to %d\n", myRank, right);
MPI_Recv(leftParcel, parcelSize, MPI_INT, left, 0, MPI_COMM_WORLD,
        MPI_STATUS_IGNORE);
printf("Received parcel at process %d from %d\n", myRank, left);
```

- MPI implementation provides an internal buffer for short messages
- MPI_Send is asynchronous when messages fit in this buffer
- Switch-over to synchronous mode beyond that
- In our case, the switch-over happens between $40kB$ and $400kB$

Ring shift : Staggered communication

```
//Send and Receive
if(myRank % 2 == 0)
{
    //Even numbered node
    MPI_Send(rightParcel, parcelSize, MPI_INT, right, 0, MPI_COMM_WORLD);
    printf("Sent parcel from process %d to %d\n", myRank, right);
    MPI_Recv(leftParcel, parcelSize, MPI_INT, left, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
    printf("Received parcel at process %d from %d\n", myRank, left);
}
else
{
    //Odd numbered node
    MPI_Recv(leftParcel, parcelSize, MPI_INT, left, 0, MPI_COMM_WORLD,
            MPI_STATUS_IGNORE);
    printf("Received parcel at process %d from %d\n", myRank, left);
    MPI_Send(rightParcel, parcelSize, MPI_INT, right, 0, MPI_COMM_WORLD);
    printf("Sent parcel from process %d to %d\n", myRank, right);
}
```

Non-blocking Communication

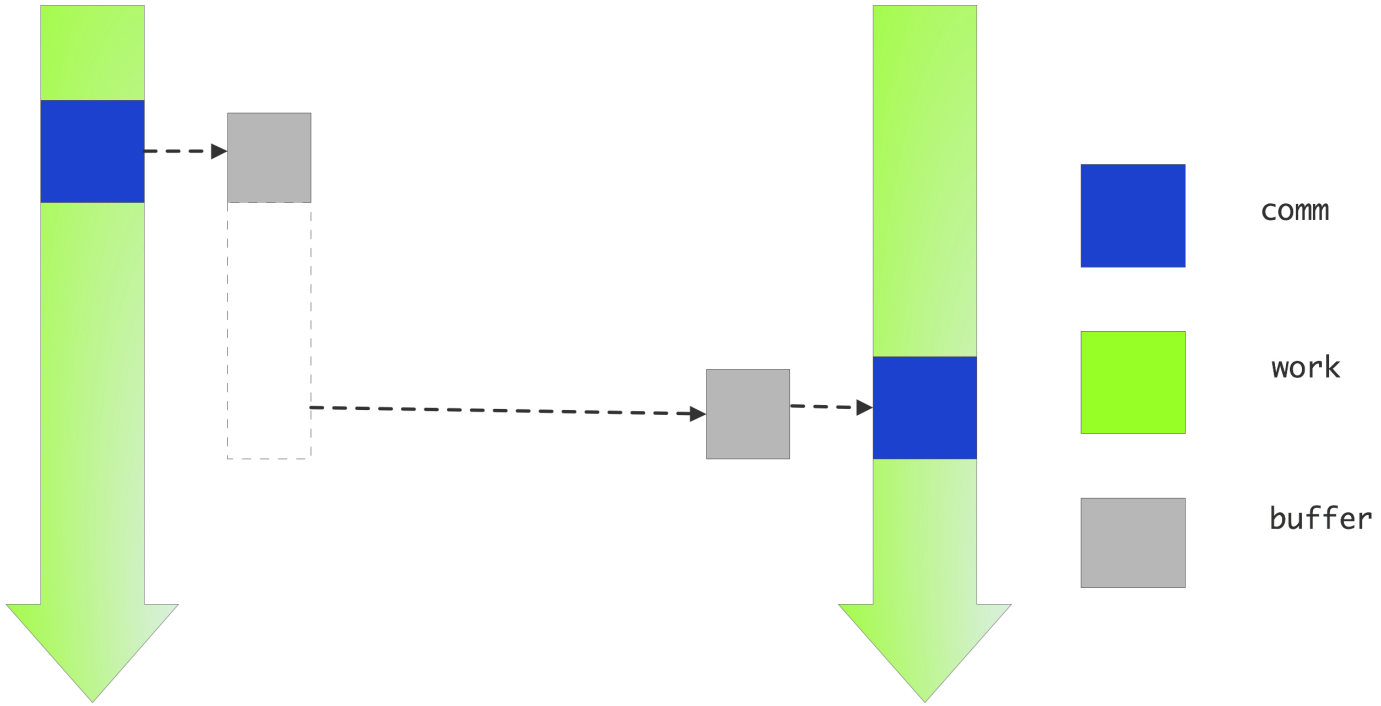


Figure : Courtesy of Victor Eijkhout

Ring shift : Non-blocking communication

```
//Send and Receive
MPI_Request sendRequest, receiveRequest;
MPI_Isend(rightParcel, parcelSize, MPI_INT, right, 0, MPI_COMM_WORLD,
        &sendRequest);
MPI_Irecv(leftParcel, parcelSize, MPI_INT, left, 0, MPI_COMM_WORLD,
        &receiveRequest);
MPI_Wait(&sendRequest, MPI_STATUS_IGNORE);
printf("Sent parcel from process %d to %d\n", myRank, right);
MPI_Wait(&receiveRequest, MPI_STATUS_IGNORE);
printf("Received parcel at process %d from %d\n", myRank, left);
```

- MPI_Isend : Non-blocking Send, MPI_Irecv : Non-blocking Receive
- MPI_Wait : Blocks until the non-blocking operation corresponding to *request* completes
- MPI_Test : Tests if the non-blocking operation corresponding to *request* has completed

Ring shift : Simultaneous Send and Receive

```
//Make a ring
const int left = (myRank == 0 ? commSize-1 : myRank-1);
const int right = (myRank == commSize-1 ? 0 : myRank + 1);

//Create parcels
const int parcelSize = (argc < 2 ? 100000 : atoi(argv[1]));
int * leftParcel = (int *) malloc(parcelSize * sizeof(int));
int * rightParcel = (int *) malloc(parcelSize * sizeof(int));

//Send and Receive
MPI_Sendrecv(rightParcel, parcelSize, MPI_INT, right, 0,
       leftParcel, parcelSize, MPI_INT, left, 0, MPI_COMM_WORLD,
       MPI_STATUS_IGNORE);
printf("Sent parcel from process %d to %d\n", myRank, right);
printf("Received parcel at process %d from %d\n", myRank, left);
```

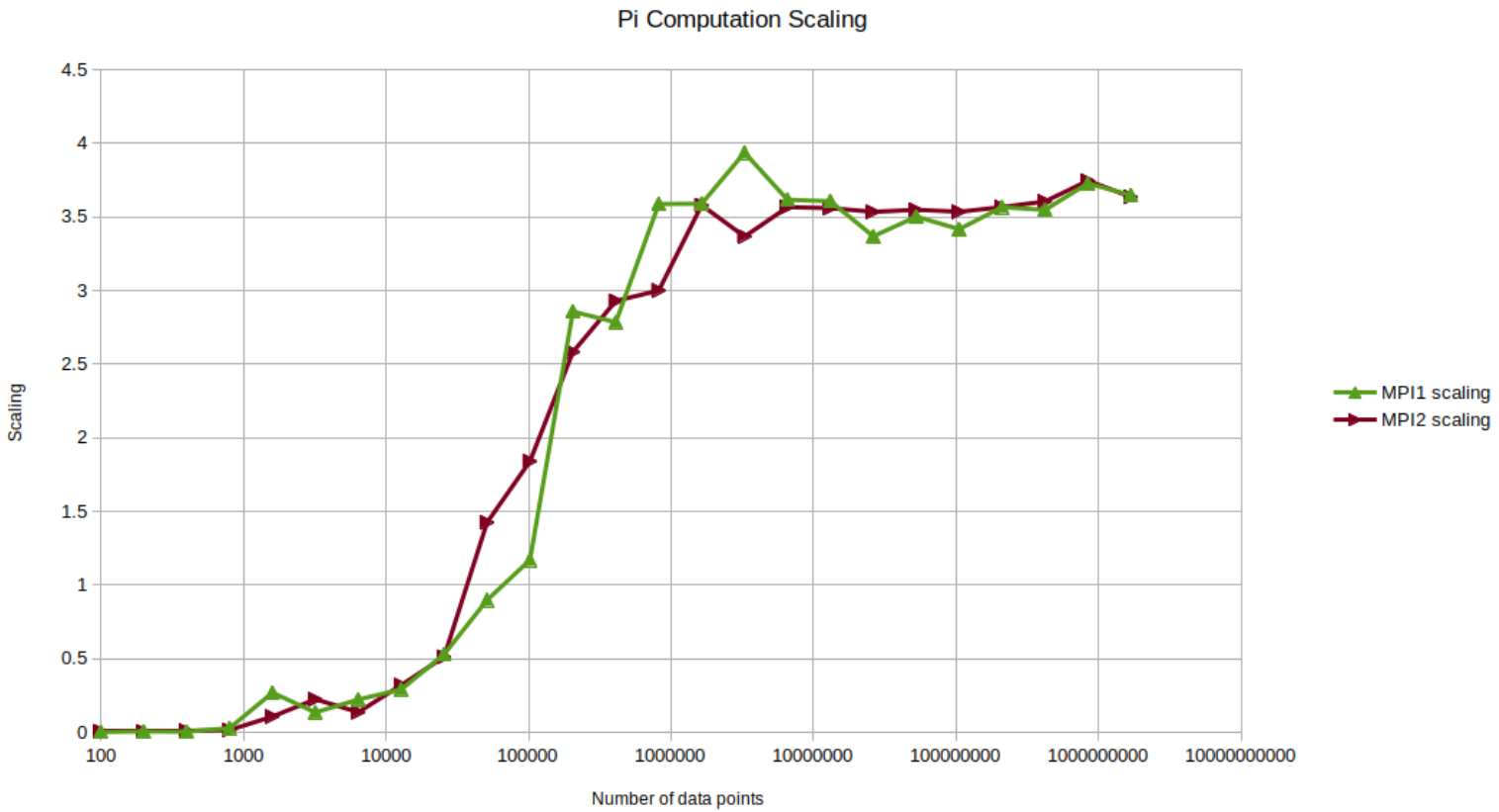
Outline

MPI_Reduce

```
double localPi = 0.0;
double xi = (0.5 + numPoints * myRank) * deltaX;
for(int i = 0; i < numPoints; ++i)
{
    localPi += 4.0/(1.0 + xi * xi);
    xi += deltaX;
}
double pi = 0.0;
MPI_Reduce(&localPi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
if(myRank == 0)
{
    pi *= deltaX;
    //printf("Value of pi : %.10g\n", pi);
}
```

- Similar (in spirit) to OpenMP reduction clause.
- Several predefined arithmetic and logic reduction operations like MPI_MAX/MIN, MPI_PROD/SUM, MPI_LAND/LOR, MPI_BAND/BOR, etc.
- MPI_Allreduce : Returns reduction result to all processes.

MPI_Reduce versus MPI_Send/MPI_Recv



- Difficult to see performance difference with only 4 nodes
- Should expect MPI_Reduce to perform better as the number of nodes increase in the MPI cluster

Measuring run-time in MPI programs

```
#include <mpi.h>
...
int main(const int argc, char ** argv)
{
    ...
    MPI_Barrier(MPI_COMM_WORLD);
    const double startTime = MPI_Wtime();
    ...
    // Parallel stuff
    ...
    MPI_Barrier(MPI_COMM_WORLD);
    const double stopTime = MPI_Wtime();
    ...
}
```

Matrix-Vector Multiply

```
void Multiply(const int n, const double * A, const double * x,
    double * y)
{
    for(int i = 0; i < n; ++i)
    {
        double product = 0.0;
        for(int j = 0; j < n; ++j)
            product += A[n*i+j] * x[j];
        y[i] += product;
    }
    return;
}
```

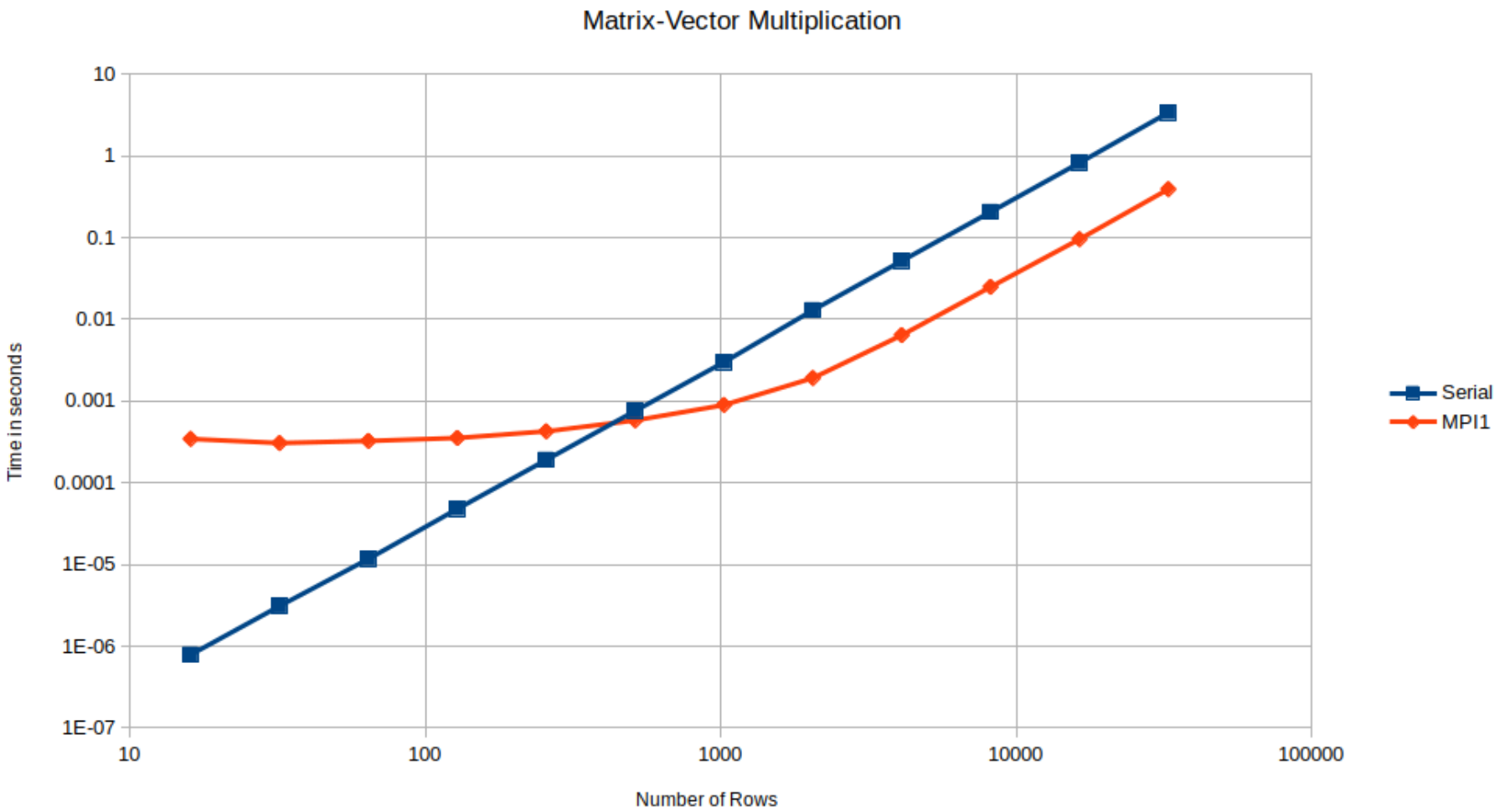
Parallel Mat-Vec Mult : Ground Rules

- p nodes
- Each node holds $\frac{n}{p}$ rows of the A matrix, specifically, node i holds the rows $\frac{n}{p}i$ to $\frac{n}{p}(i+1)-1$.
- Node n_i holds $\frac{n}{p}$ elements of the RHS vector y and the LHS vector x also, i.e., $[y_{\frac{n}{p}i}, y_{\frac{n}{p}(i+1)})$ and $[x_{\frac{n}{p}i}, x_{\frac{n}{p}(i+1)})$
- The only communication between nodes happens for constructing the full LHS vector x at all nodes. Each node needs to send it's part of LHS vector x to every other node.

Parallel Mat-Vec Mult : All Broadcast

```
void Multiply(const int myRank, const int commSize, const int n,
    const int m,
        const double * localA, double * x, double * localY)
{
    for(int i = 0; i < commSize; ++i)
        MPI_Bcast(&x[m*i], m, MPI_DOUBLE, i, MPI_COMM_WORLD);
    for(int i = 0; i < m; ++i)
    {
        double product = 0.0;
        for(int j = 0; j < n; ++j)
            product += localA[n*i+j] * x[j];
        localY[i] += product;
    }
    return;
}
```

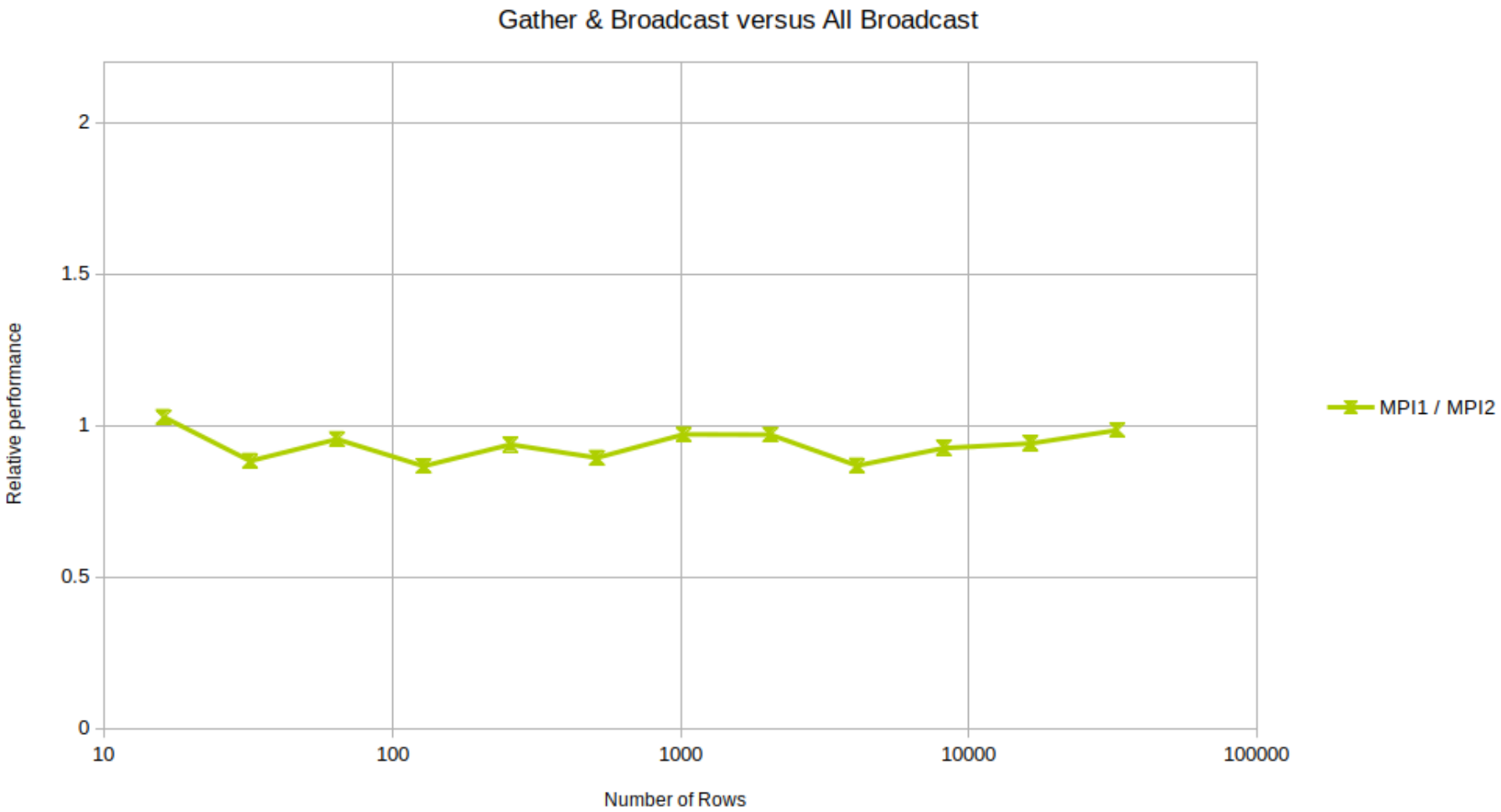
Parallel Mat-Vec Mult : Performance



Parallel Mat-Vec Mult : Gather and Broadcast

```
void Multiply(const int myRank, const int n, const int m,
        const double * localA, double * localX, double * localY,
            double * x)
{
    MPI_Gather(localX, m, MPI_DOUBLE, x, m, MPI_DOUBLE, 0,
        MPI_COMM_WORLD);
    MPI_Bcast(x, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    for(int i = 0; i < m; ++i)
    {
        double product = 0.0;
        for(int j = 0; j < n; ++j)
            product += localA[n*i+j] * x[j];
        localY[i] += product;
    }
    return;
}
```

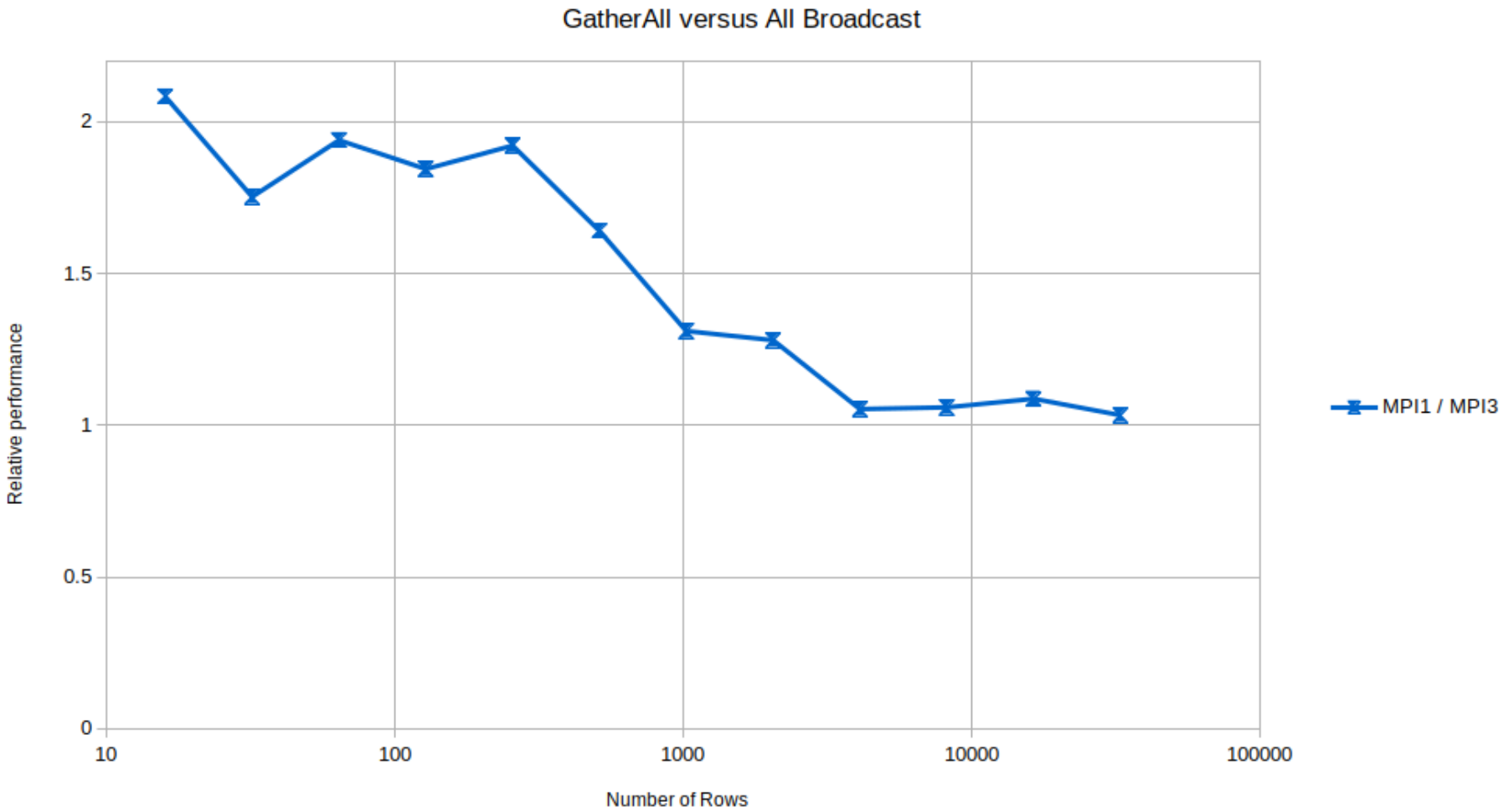
Gather and Broadcast versus All Broadcast



Parallel Mat-Vec Mult : All Gather

```
void Multiply(const int myRank, const int n, const int m,
        const double * localA, double * localX, double * localY,
            double * x)
{
    MPI_Allgather(localX, m, MPI_DOUBLE, x, m, MPI_DOUBLE,
        MPI_COMM_WORLD);
    for(int i = 0; i < m; ++i)
    {
        double product = 0.0;
        for(int j = 0; j < n; ++j)
            product += localA[n*i+j] * x[j];
        localY[i] += product;
    }
    return;
}
```

All Gather versus All Broadcast



Other Collective Operations

- MPI_Scan : Prefix Sum
- MPI_Alltoall : Each node to all nodes (including itself)

Outline

Why ?

- Communication can be much more expensive than computation
- Latency is a big component of communication cost
- Profitable to combine multiple messages for different datatypes into a single message

How ?

```
//Component bocks of new datatype
int a[10]; double b[8]; float c[6];
MPI_DATAtype array_of_types[3] = {MPI_INT, MPI_DOUBLE,
    MPI_FLOAT};
int count = 3; //Number of blocks
//Number of elements in each block
int array_of_blocklengths[3] = {10, 8, 6};
//Find byte displacment of the 3 blocks
MPI_Aint aAddress, bAddress, cAddress;
MPI_Get_address(&a[0], &aAddress);
MPI_Get_address(&b[0], &bAddress);
MPI_Get_address(&c[0], &cAddress);
int array_of_displacements[3];
array_of_displacements[0] = 0;
array_of_displacements[1] = bAddress - aAddress;
array_of_displacements[2] = cAddress - bAddress;
```

How ...

```
//Create new datatype
MPI_Datatype myMPItype_t;
MPI_Type_create_struct(count, array_of_blocklengths,
        array_of_displacements, array_of_types, &myMPItype_t);
MPI_Type_commit(myMPItype_t); //Commit myMPItype_t
...
//Use myMPItype_t
...
MPI_Type_free(myMPItype_t); //Free myMPItype_t
```

Function description

```
int MPI_Type_create_struct(int count, //number of blocks
    const int array_of_blocklengths[], //number of elements in
        each block
    const MPI_Aint array_of_displacements[], //byte displacement
        of each block
    const MPI_Datatype array_of_types[], //type of elements in
        each block
    MPI_Datatype *newtype); //output parameter; new datatype

int MPI_Get_address(const void *location, //location in caller
     memory
    MPI_Aint *address); //output parameter; address of location

int MPI_Type_commit(MPI_Datatype *datatype);

int MPI_Type_free(MPI_Datatype *datatype);
```

Further Reading

- Introduction to Parallel Computing, Second Edition - Grama, Gupta, Karypis, Kumar : Chapter 6
- An Introduction to Parallel Programming - Peter Pacheco : Chapter 3