TATA CONSULTANCY SERVICES
Experience certainty.

TATA

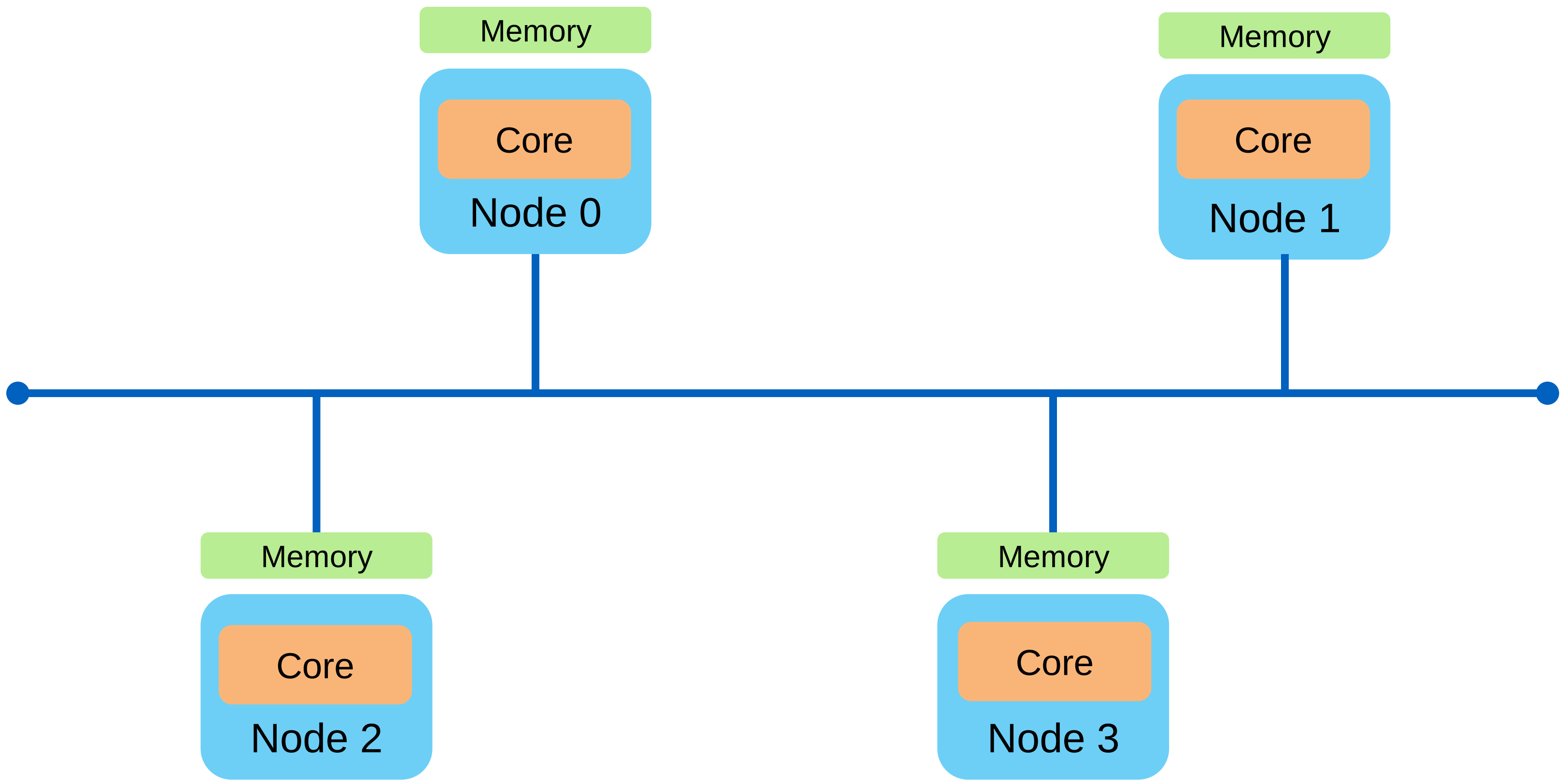
HPC Parallel Programing

Multi-node Computation with MPI - I

Parallelization and Optimization Group
TATA Consultancy Services, Sahyadri Park Pune, India

April 29, 2013

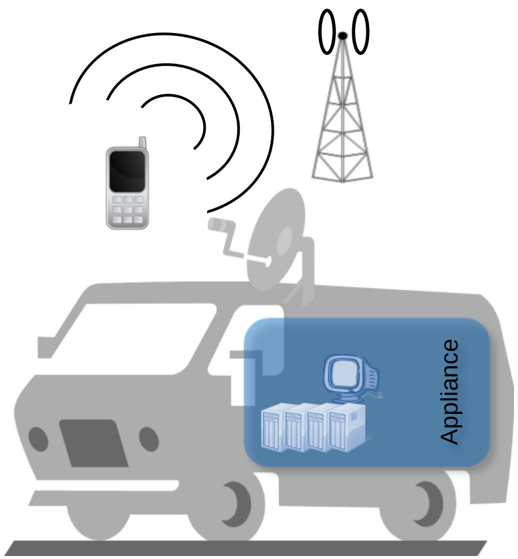
Multi node environment



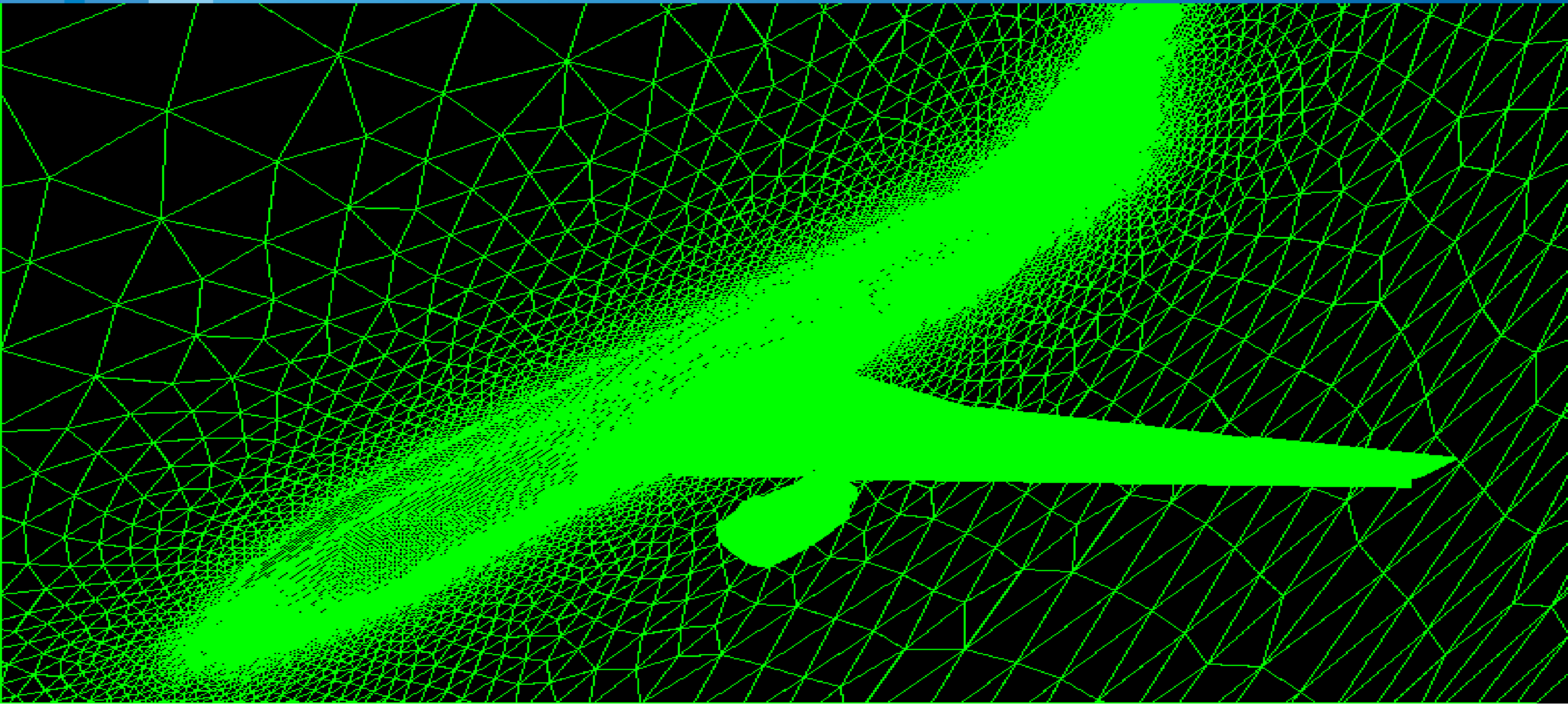
4 Nodes in a network

HPC in Cryptanalysis

- GSM Cipher breaking
- 6 x 10^6 CPU hours of one time computation
- 160 CPU hours of computation and 2^{30} searches in 5 – 10 TB data required to be accomplished in real time



High Lift System Analysis - Boeing Research Project



- Grid Size 60 M Cells
- Time taken 24 hours on 256 cores

MPI World

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main (int argc, char *argv[])
{
        int id; // process rank
        int p; // number of processes
        char hostname[128];
        gethostname(hostname,128);
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &id);
        MPI_Comm_size(MPI_COMM_WORLD, &p);

        printf("I am rank: %d out of %d
                       \n", id, p);

        MPI_Finalize();
        return 0;
}
```

Output –

$ I am rank: 0 out of 4
I am rank: 3 out of 4
I am rank: 1 out of 4
I am rank: 2 out of 4

Multi node computations

Outline

- MPI overview
- Point to Point communication
  - One to One communication
- Collective communication
  - One to all, All to one & All to All
- Tools for MPI

TATA CONSULTANCY SERVICES
Experience certainty.

Point to Point Communication

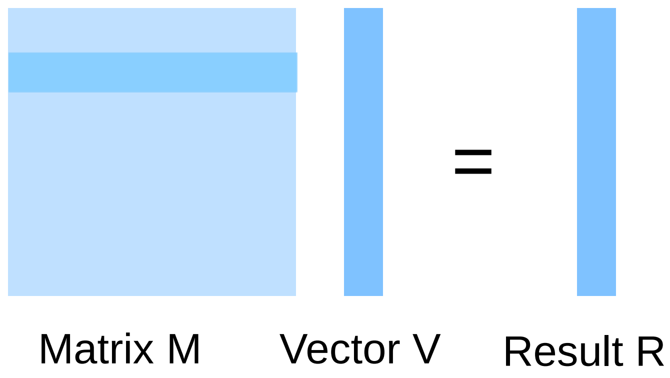
Send and Receive

May 9, 2013

Computational Problem

Overview

- The Matrix – Vector product
- Size MxM for some large M
- For row = 0 to M
  - row*vec
- Typically computed sequentially
- Multi threaded solution
- What if memory is not sufficient
- We have N compute nodes
  - Partitioning of data
  - Data communication
- Message Passing Interface



Message passing Interface – MPI

- Message Passing Interface
- A standard
- Implementations
  - Commercial – HP MPI, IBM MPI
  - Open Source – OpenMPI, mvapich, mpich
- Similarity with threads – parallel execution

MPI – First encounter

MPI Start and finish

```
int MPI_Init (int *argc, char **argv)
int MPI_Finalize (void)
```

Information for calculations

```
int MPI_Comm_size (MPI_Comm comm, int *size)
int MPI_Comm_rank (MPI_Comm comm, int *rank)
```

First Program

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main (int argc, char *argv[])
{
        int id; // process rank
        int p; // number of processes
        char hostname[128];
        gethostname(hostname,128);
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &id);
        MPI_Comm_size(MPI_COMM_WORLD, &p);

        printf("I am rank: %d out of %d
                        \n", id, p);

        MPI_Finalize(); // To be called
                                        last
and once
        return 0;
}
```

Compile –

$ mpicc my_first_mpi.c -o run.out

Run –

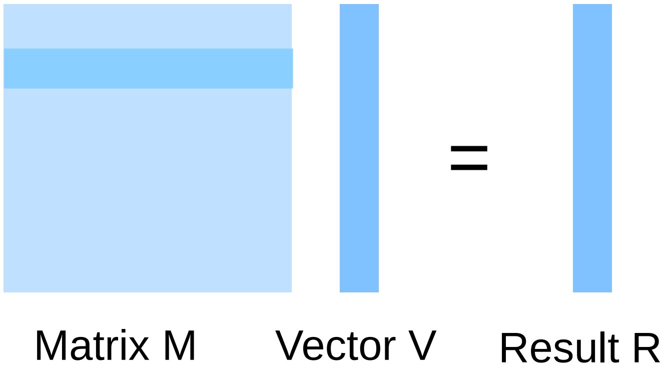
$ mpirun –np <num_cpu> ./run.out

Output –

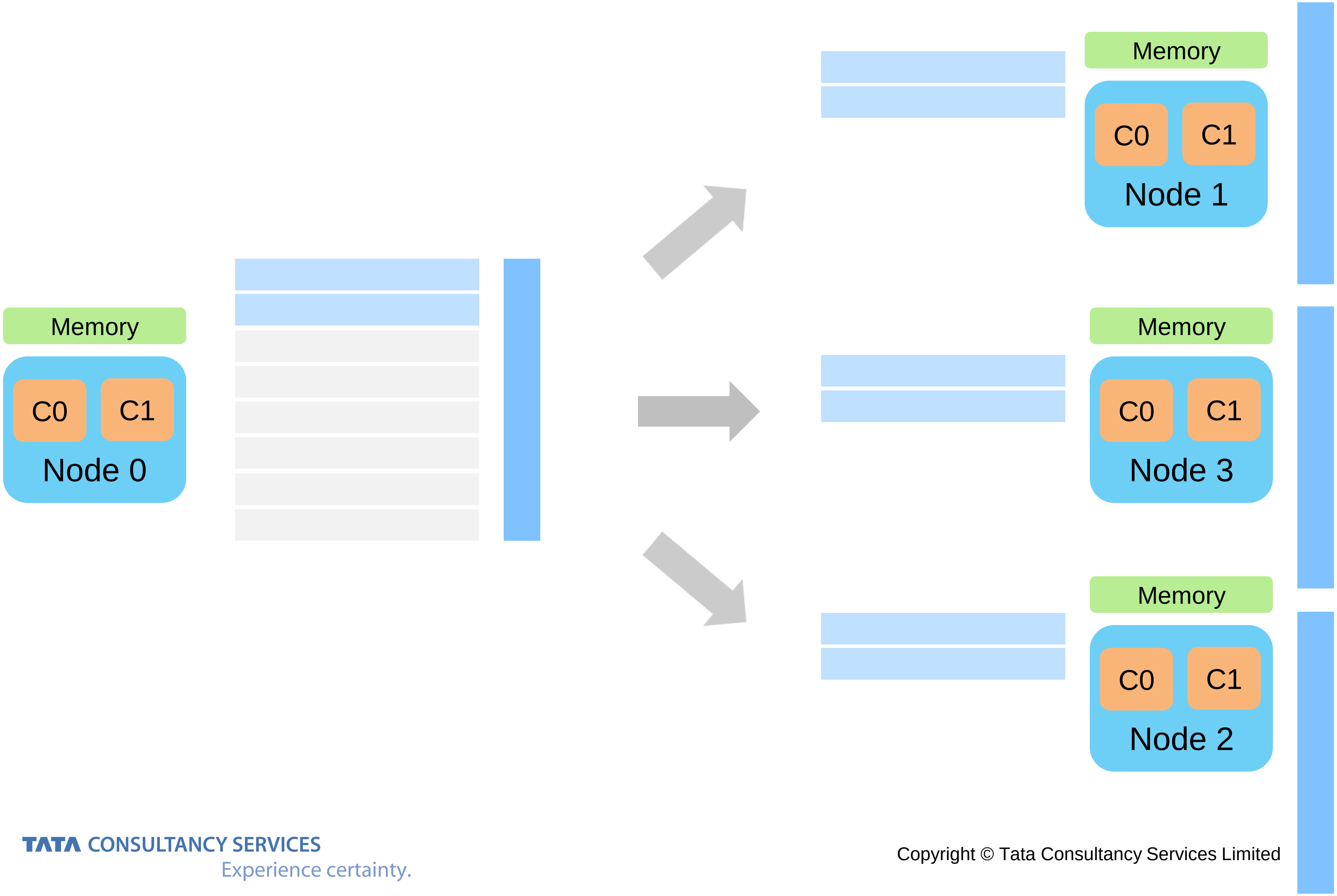
$ I am rank: 2 out of num_cpu

Matrix – Vector product

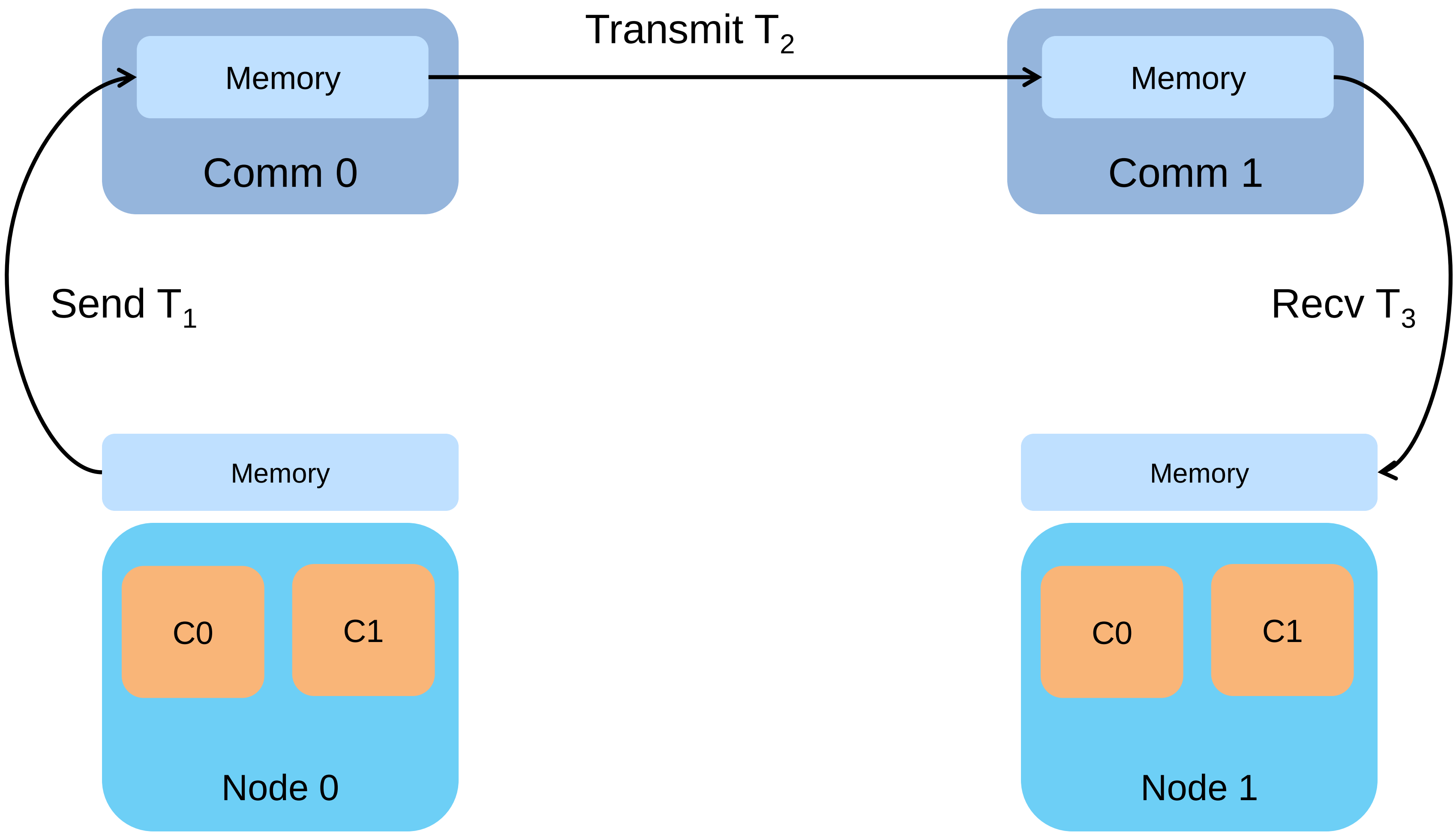
- M x M matrix for large M
- P compute nodes
- Partitioning of data, How?
  - M/P rows to each node
  - Vector V to all
- Message Passing Interface
  - MPI Send and receive
  - Performance gain ?
  - What factor?
- Data transfer between nodes
- Communication cost ?



Matrix – Vector Distribution



MPI Send and Recv



Communication cost

- Lets measure different timing in send/recv process
- Cost involved in data send is $(T_1+T_2+T_3)$

	Timing in μsec	
	Round Trip	One way Avg
1 char	3	1
10 chars	126	61
100 chars	926	467

MPI_Send & MPI_Recv

MPI Send and Recv (Blocking calls)

```
MPI_Send(void* data, int count, MPI_Datatype
datatype, int destination, int tag, MPI_Comm
communicator)
```

```
MPI_Recv(void* data, int count, MPI_Datatype
datatype, int source, int tag, MPI_Comm
communicator, MPI_Status* status)
```

MPI Send and Recv

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main (int argc, char *argv[])
{
        int id; // process rank
        int p; // number of processes
        int send_buff, recv_buff;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &id);
        MPI_Comm_size(MPI_COMM_WORLD, &p);
        if(0 == id)
        {
                send_buff = 10;
                MPI_Send(&send_buff, 1, MPI_INT, 1, TAG, MPI_COMM_WORLD);
        }
        if(1 == id)
        {
                MPI_Recv(&recv_buff, 1, MPI_INT, 0, TAG, MPI_COMM_WORLD,
                                                       &Status);
        }
        MPI_Finalize();
        return 0;
}
```

Things to remember

- Same program runs on each rank
- All ranks should have space allocated for recv_buff before actual recv call

Matrix – Vector product with MPI

MV_SendRecv.c

Summary

- Lets summarize
  - Introduction to MPI
  - Basic construct
  - Parallel computation comes with communication
  - Communication cost
  - Data send and receive
  - Matrix – Vector dot product using MPI

Non blocking Send and Recv

- Cost involved in data send/recv is $(T_1+T_2+T_3)$
- Process blocks till data is copied to/from comm buffer
- Can we do some thing else during this time?
  - Yes
  - Sender and receiver both can work on other tasks
- Non blocking calls
  - MPI_Isend & MPI_Irecv

MPI_Isend & MPI_Irecv

MPI Isend and Irecv (Non Blocking calls)

```
MPI_Isend(void *buf, int count, MPI_Datatype
datatype, int dest, int tag, MPI_Comm comm,
  MPI_Request *request)

MPI_Irecv(void *buf, int count, MPI_Datatype
datatype, int source, int tag, MPI_Comm comm,
  MPI_Request *request)
```

Example

```
#include <stdio.h>
#include <unistd.h>
#include <mpi.h>
int main (int argc, char *argv[])
{
        int id; // process rank
        int p; // number of processes
        int send_buff, recv_buff;
        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &id);
        MPI_Comm_size(MPI_COMM_WORLD, &p);
        if(0 == id)
        {
                send_buff = 10;
                MPI_Isend(&send_buff, 1, MPI_INT, 1, TAG, MPI_COMM_WORLD,
                                                &reqs[tag1]);
                my_task();
        }
        if(1 == id)
                MPI_Recv(&recv_buff, 1, MPI_INT, 0, TAG, MPI_COMM_WORLD,
                                                &Status);
        MPI_Finalize();
        return 0;
}
```

Example

- Lets consider an example where we send a buffer and also need to do some computation
- MPI_Send(&buff, …)
- Computation

```
For(i = 0; i < M; i++)
    c[i] = a[i] + b[i];
```

Program	Time in µsec
With MPI_Send	54430
With MPI_Isend	18488

TATA CONSULTANCY SERVICES
Experience certainty.

TATA

Thank You

May 9, 2013

TATA CONSULTANCY SERVICES
Experience certainty.

TATA

HPC Parallel Programing

Multi-node Computation with MPI - II

Parallelization and Optimization Group
TATA Consultancy Services, Sahyadri Park Pune, India

April 29, 2013

Multi node computations

Outline

- Collective communication
- One to all, all to one, all to all
- Barrier, Broadcast, Gather, Scatter, All gather, Reduce

TATA CONSULTANCY SERVICES
Experience certainty.

MPI Collectives – Part I

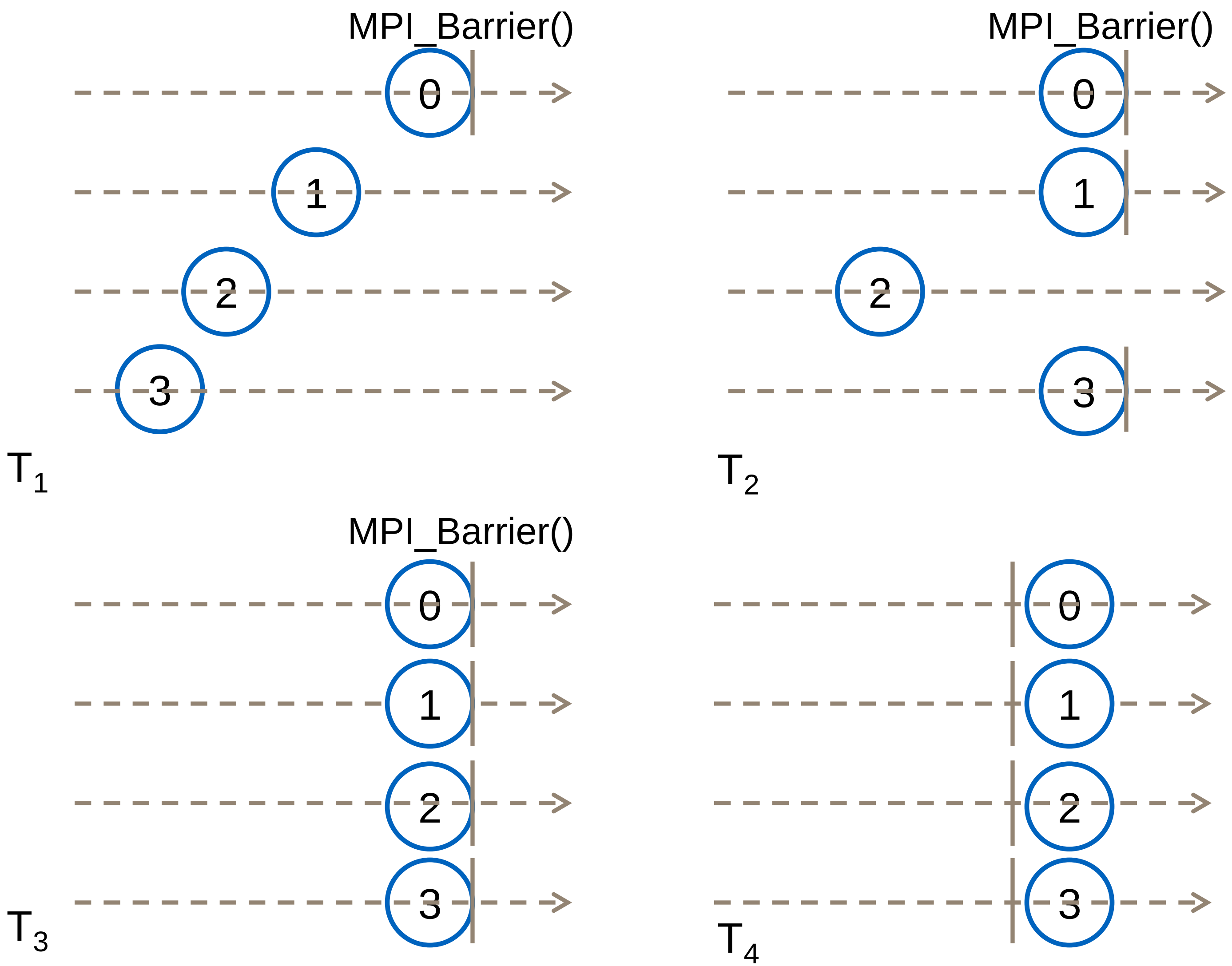
One to All communication

May 9, 2013

Collective Constructs

- So far we have seen point to point communication
  - One source and one destination
  - MPI_Send(), MPI_Recv
- Communication involving all processes
  - One to all, all to all, all to one
- Challenge?
  - Synchronization
    - Read modify write operations
    - All processes must reach a common point
  - Barrier

MPI Barrier


MPI_Barrier()
0
1
2
3
T_1
MPI_Barrier()
0
1
2
3
T_2
MPI_Barrier()
0
1
2
3
T_3
0
1
2
3
T_4

MPI Barrier

MPI Construct

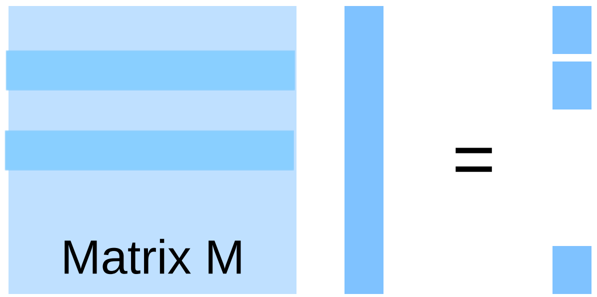
MPI_Barrier(MPI_Comm communicator)

```
for (i = 0; i < num_trials; i++)
{
        //Synchronize before starting
        MPI_Barrier(MPI_COMM_WORLD);
        my_mpi_function();
        // Synchronize again
        MPI_Barrier(MPI_COMM_WORLD);
}
```

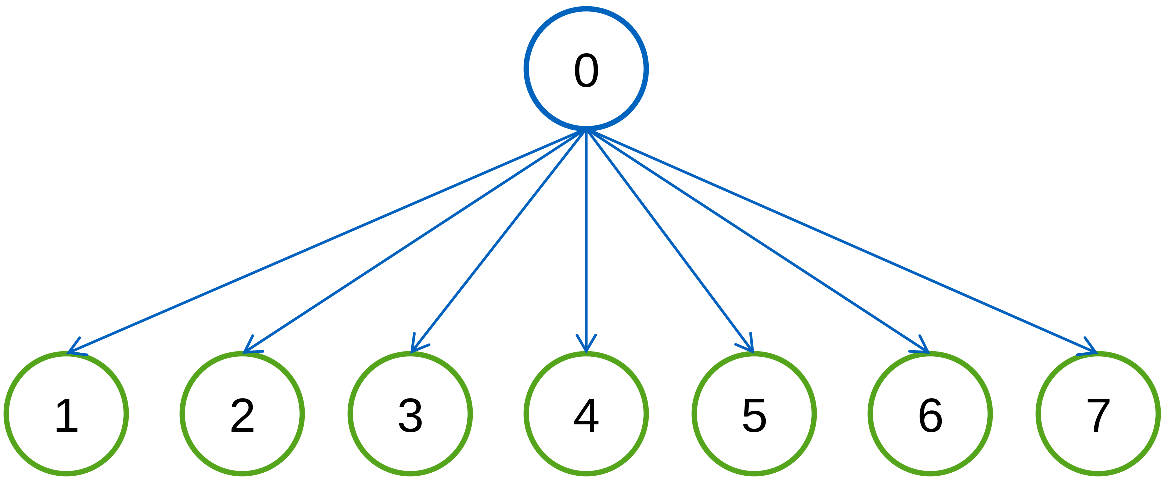
Matrix – Vector Product problem

Overview

- Matrix – Vector product
- Matrix M, vector V & result vector R
  - R = matvec_prod(M, V)
- On multi-node (P) setup?
- Data distribution
  - Distribute rows (M/P) to each node
  - Vector V to all



MPI Broadcast



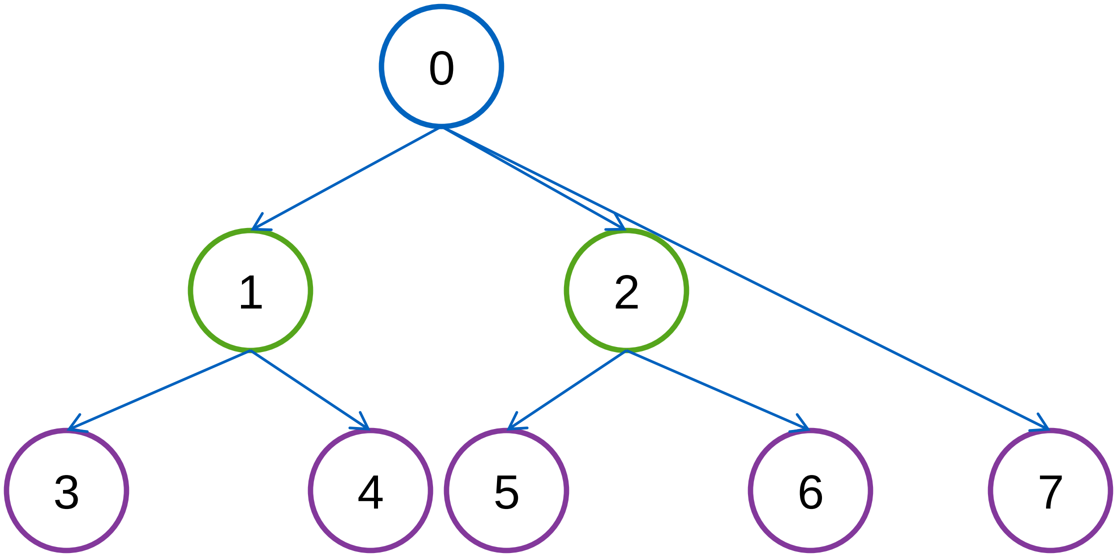
- Process 0 sends data to all
- Obvious choice
  - MPI_Send()

MPI Broadcast

```
if(0 == id)
{
        send_buff = 10;
        for (i = 1, i < num_procs; i++)
                MPI_Send(&send_buff, 1, MPI_INT, i, TAG,
                                                MPI_COMM_WORLD);
}
else
        MPI_Recv(&recv_buff, 1, MPI_INT, 0, TAG, MPI_COMM_WORLD,
                                                &status);
```

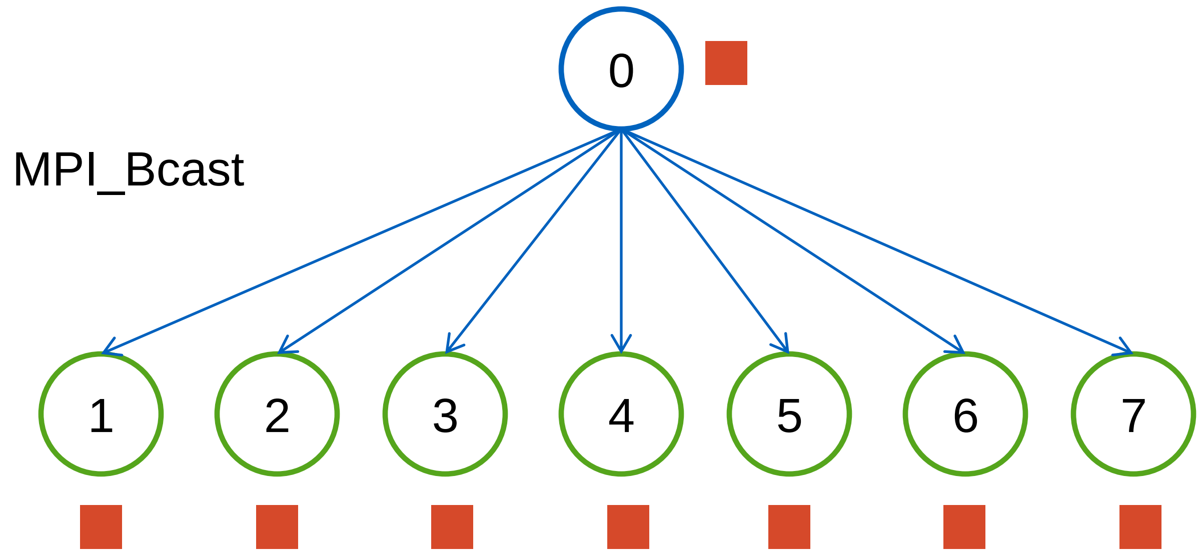
- Process 0 sends data to all
- Is it good enough?
- Can we do better?
    - Yes
- Loop is using only 1 network link (0 to other nodes)

MPI Broadcast



- Tree based approach is much more efficient
- More network links get utilized
- MPI provides a construct for this
- MPI_Bcast (MPI Broadcast)

MPI Broadcast



MPI Construct

MPI_Bcast(void* data, int count, MPI_Datatype datatype, int root, MPI_Comm communicator)

Efficiency

- Comparison of broadcast with MPI_Bcast() & My_Bcast()
- My_Bcast()
  - For loop MPI_Send() & MPI_Recv()

Num of Processors	My_Bcast	MPI_Bcast()
	Timing in µ sec	
2	132	60
4	147	66
8	3162	117
16	17985	136

First Example

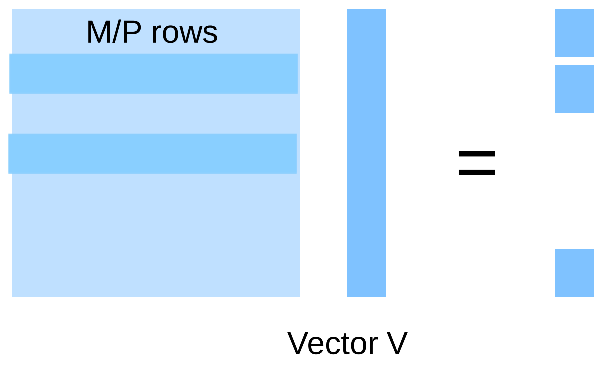
```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main (int argc, char *argv[])
{
        int id; // process rank
        int p; // number of processes
        int send_buff;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &id);
        MPI_Comm_size(MPI_COMM_WORLD, &p);
        if(0 == id)
                send_buff = 10;
        MPI_Bcast(&send_buff, 1, MPI_INT, 0, MPI_COMM_WORLD);

        MPI_Finalize();
        return 0;
}
```

Summary

- Synchronization of process
  - MPI_Barrier()
- Collective communication
  - One to all
- My broadcast using MPI send/recv
- MPI Broadcast – MPI_Bcast()
  - Tree based approach
  - Efficient
  - First example using MPI_Bcast()

Back to Matrix – Vector product

- Our partitioning approach
  - Each process gets M/P rows and full vector V
- What can we broadcast?
- Rows of M or vector V or both?
  - Vector V
- Our strategy would be
  - Process 0 sends M/P rows to each
  - Broadcast V to all
  - Each computes M/P elements of result vector



Matrix – Vector product

- We have all the inputs for Matrix-Vector product program
- So lets explore Matrix-vector product using MPI_Bcast()

Mv_bcast.c

TATA CONSULTANCY SERVICES
Experience certainty.

Thank You

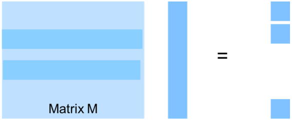
May 9, 2013

HPC Parallel Programing Multi-node Computation with MPI - III

Parallelization and Optimization Group
TATA Consultancy Services, SahyadriPark Pune, India

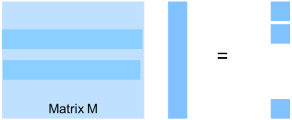
May 9, 2013

Discussions thus far: MV product



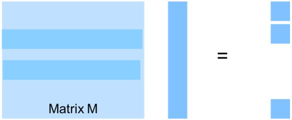
1. Matrix vector product parallel implementation.
2. Each process broadcasted vector V.

Matrix Vector product



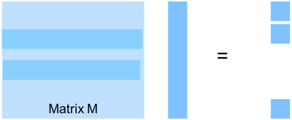
1. N rows, P processes.
2. Each process gets N/P rows for local computation.

Matrix Vector product



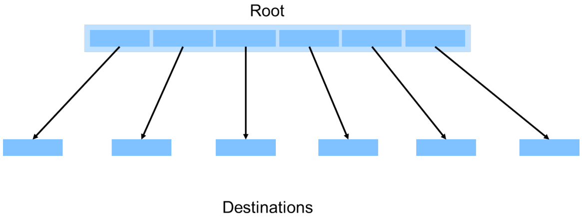
1. N rows, P processes.
2. Each process gets N/P rows for local computation.
3. Data can be sent to each process using send receive routines.
4. Will involve multiple pairs of data exchange among each process.

Matrix Vector product



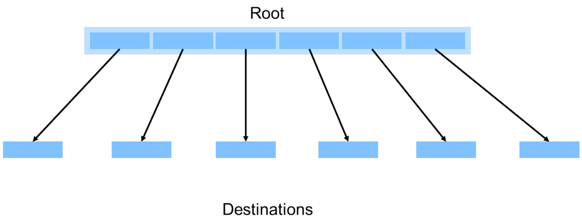
1. N rows, P processes.
2. Each process gets N/P rows for local computation.
3. Data can be sent to each process using send receive routines.
4. Will involve multiple pairs of data exchange among each process.
5. Scatter rows using MPI_Scatter

MPI Scatter



1. Distributes equal sized chunks of data from a root process to other processes within a group.

MPI Scatter



1. Distributes equal sized chunks of data from a root process to other processes within a group.
2. Distribution of data is taken care internally and sent in order of ranks.

MPI Scatter

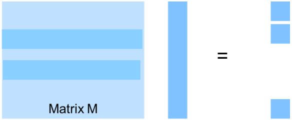
MPI_Scatter (&sendbuf, sendcnt, sendtype, &recvbuf, recvcnt, recvtype, root, comm)

1. sendbuf (starting address of send buffer)
2. sendcount (num elements sent to each process)
3. sendtype (type)
4. recvbuf (address of receive buffer)
5. recvcount (num elements in receive buffer)
6. recvtype (data type of receive elements)
7. root (rank of sending process)
8. comm (communicator)

Scattering Matrix

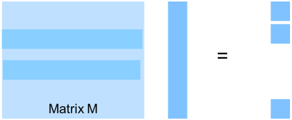
```
float A[N][N], Ap[N/P][N], b[N];

root = 0;

MPI_Scatter(A, N/P*N, MPI_Float, Ap, N/P*N, MPI_Float, root,
    MPI_COMM_WORLD);
```


Matrix Vector product



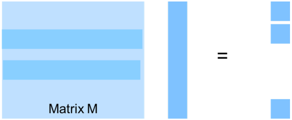
1. Partial results on each prosess: N / P rows multiplied with vector V.

Matrix Vector product



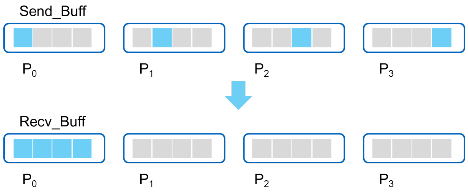
1. Partial results on each prosess: N / P rows multiplied with vector V.
2. Partial results from individual processes need to be assembled to one process.

Matrix Vector product



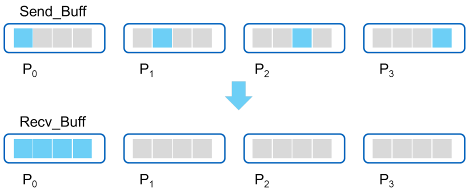
1. Partial results on each prosess: N / P rows multiplied with vector V.
2. Partial results from individual processes need to be assembled to one process.
3. Can be achieved using MPI_Gather.

MPI Gather



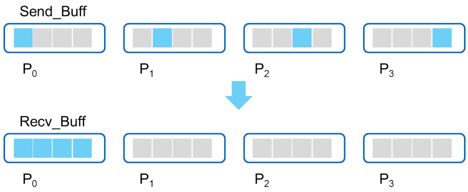
1. MPI_Gather collects results from individual processes to a root process.

MPI Gather



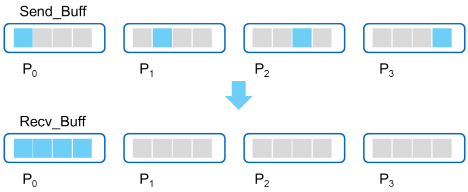
1. MPI_Gather collects results from individual processes to a root process.
2. Send receive routines would require multiple pairs of data exchange.

MPI Gather



1. MPI_Gather collects results from individual processes to a root process.
2. Send receive routines would require multiple pairs of data exchange.
3. MPI_Gather (&sendbuf, sendcnt, sendtype, &recvbuf, recvcount, recvtype, root, comm)

MPI Gather

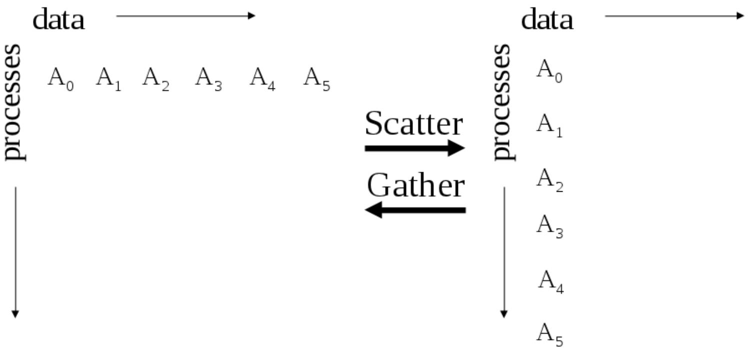


1. MPI_Gather collects results from individual processes to a root process.
2. Send receive routines would require multiple pairs of data exchange.
3. MPI_Gather (&sendbuf, sendcnt, sendtype, &recvbuf, recvcount, recvtype, root, comm)

Gather MV product elements

```
float A[N][N], Ap[N/P][N], b[N], c[N], cp[N/P];

for (i = 1; i < N/P; i++)
{
cp[i] = 0;
 for (k = 0; k < N; k++)
   cp[i] = cp[i] + Ap[i][k] * b[k];
 }
MPI_Gather(cp, N/P, MPI_Float, c, N/P, MPI_Float, root,
 MPI_COMM_WORLD);
```

Scatter - Gather



Summary

What we covered yet :

Summary

What we covered yet :

1. MPI_Scatter: distributuion of data to multiple processes.

Summary

What we covered yet :

1. MPI_Scatter: distributuion of data to multiple processes.
2. MPI_Gather: collect multiple process results to one process.

Summary

What we covered yet :

1. MPI_Scatter: distributuion of data to multiple processes.
2. MPI_Gather: collect multiple process results to one process.

Some more collectives :

1. MPI_AllGather

Summary

What we covered yet :

1. MPI_Scatter: distributuion of data to multiple processes.
2. MPI_Gather: collect multiple process results to one process.

Some more collectives :

1. MPI_AllGather
2. MPI_Reduce

Summary

What we covered yet :

1. MPI_Scatter: distributuion of data to multiple processes.
2. MPI_Gather: collect multiple process results to one process.

Some more collectives :

1. MPI_AllGather
2. MPI_Reduce
3. MPI_All Reduce

Summary

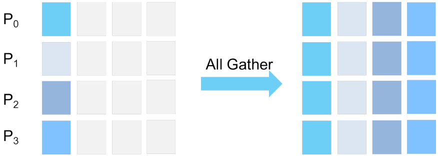
What we covered yet :

1. MPI_Scatter: distributuion of data to multiple processes.
2. MPI_Gather: collect multiple process results to one process.

Some more collectives :

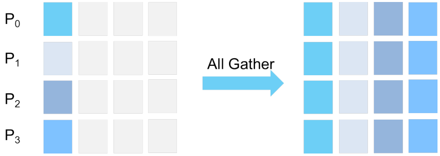
1. MPI_AllGather
2. MPI_Reduce
3. MPI_All Reduce
4. MPI_AlltoAll

MPI All Gather



1. Gathers data from all tasks and distribute the combined data to all tasks.

MPI All Gather

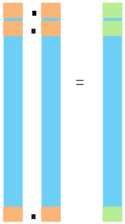


1. Gathers data from all tasks and distribute the combined data to all tasks.
2. MPI_Allgather (&sendbuf, sendcount, sendtype, &recvbuf, recvcount, recvtype, comm)

MPI All Gather

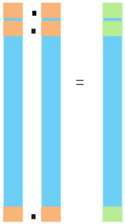
```
float A[N][N], Ap[N/P][N], b[N], c[N], cp[N/P];

for (i = 1; i < N/P; i++)

{
cp[i] = 0;
 for (k = 0; k < N; k++)
cp[i] = cp[i] + Ap[i][k] * b[k];

}
MPI_AllGather(cp, N/P, MPI_Float, c, N/P, MPI_Float,
 MPI_COMM_WORLD);
```

Problem : Inner Product of two Vectors



dotProduct = a1 * b1 + a2 * b2 + a3 * b3 + ...

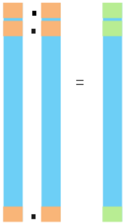
Problem : Inner Product of two Vectors



dotProduct = a1 * b1 + a2 * b2 + a3 * b3 + ...

1. Computation of local sums with multiple processes

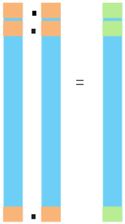
Problem : Inner Product of two Vectors



dotProduct = a1 * b1 + a2 * b2 + a3 * b3 + ...

1. Computation of local sums with multiple processes
2. Gathering of local sums to process root.

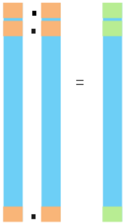
Problem : Inner Product of two Vectors



dotProduct = a1 * b1 + a2 * b2 + a3 * b3 + ...

1. Computation of local sums with multiple processes
2. Gathering of local sums to process root.
3. Summation of local sums on process root.

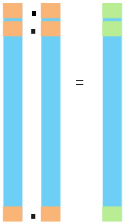
Problem : Inner Product of two Vectors



dotProduct = a1 * b1 + a2 * b2 + a3 * b3 + ...

1. Computation of local sums with multiple processes
2. Gathering of local sums to process root.
3. Summation of local sums on process root.
4. Gathering of data and summation can be combined using MPI_Reduce.

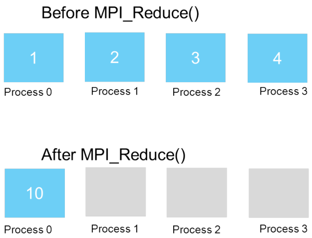
Problem : Inner Product of two Vectors



dotProduct = a1 * b1 + a2 * b2 + a3 * b3 + ...

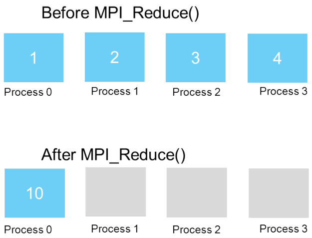
1. Computation of local sums with multiple processes
2. Gathering of local sums to process root.
3. Summation of local sums on process root.
4. Gathering of data and summation can be combined using MPI_Reduce.

MPI Reduce



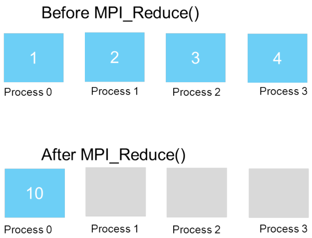
1. Applies a reduction operation on all tasks in the group and places the result in one task.

MPI Reduce



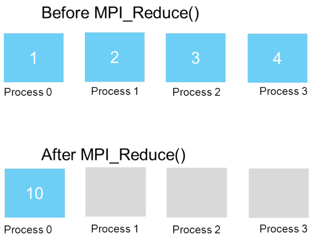
1. Applies a reduction operation on all tasks in the group and places the result in one task.
2. Operations like sum, product etc can be performed on the gathered data.

MPI Reduce



1. Applies a reduction operation on all tasks in the group and places the result in one task.
2. Operations like sum, product etc can be performed on the gathered data.
3. MPI_Reduce (&sendbuf,&recvbuf, count, datatype, op, root, comm)

MPI Reduce

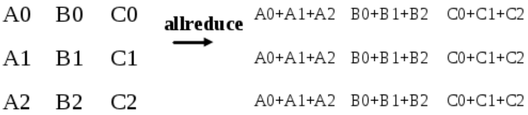


1. Applies a reduction operation on all tasks in the group and places the result in one task.
2. Operations like sum, product etc can be performed on the gathered data.
3. MPI_Reduce (&sendbuf,&recvbuf, count, datatype, op, root, comm)

MPI Reduce

```
loc_n = n/p;
bn = 1 + (my_rank) * loc_n;
en = bn + loc_n -1;
loc_dot = 0.0;
for (i = bn; i <= en; i++) {
   loc_dot = loc_dot + a[i]*b[i];
}

MPI_Reduce(&loc_dot, &globalsum, 1, MPI_FLOAT, MPI_SUM, 0,
     MPI_COMM_WORLD);
```

MPI All Reduce



1. Applies a reduction operation and places the result in all tasks in the group.

MPI All Reduce

A0	B0	C0	allreduce $\longrightarrow$	A0+A1+A2	B0+B1+B2	C0+C1+C2
A1	B1	C1		A0+A1+A2	B0+B1+B2	C0+C1+C2
A2	B2	C2		A0+A1+A2	B0+B1+B2	C0+C1+C2

1. Applies a reduction operation and places the result in all tasks in the group.
2. This is equivalent to an MPI_Reduce followed by an MPI_Bcast.

MPI All Reduce

A0	B0	C0	allreduce →	A0+A1+A2	B0+B1+B2	C0+C1+C2
A1	B1	C1		A0+A1+A2	B0+B1+B2	C0+C1+C2
A2	B2	C2		A0+A1+A2	B0+B1+B2	C0+C1+C2

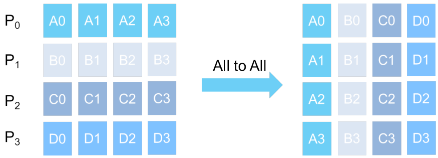
1. Applies a reduction operation and places the result in all tasks in the group.
2. This is equivalent to an MPI_Reduce followed by an MPI_Bcast.
3. MPI_Allreduce (&sendbuf, &recvbuf, count, datatype, op, comm)

MPI All Reduce

A0	B0	C0	**allreduce** →	A0+A1+A2	B0+B1+B2	C0+C1+C2
A1	B1	C1		A0+A1+A2	B0+B1+B2	C0+C1+C2
A2	B2	C2		A0+A1+A2	B0+B1+B2	C0+C1+C2

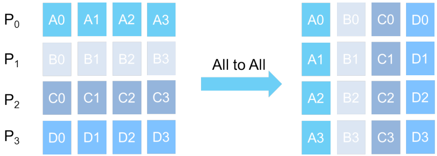
1. Applies a reduction operation and places the result in all tasks in the group.
2. This is equivalent to an MPI_Reduce followed by an MPI_Bcast.
3. MPI_Allreduce (&sendbuf, &recvbuf, count, datatype, op, comm)

MPI All to All



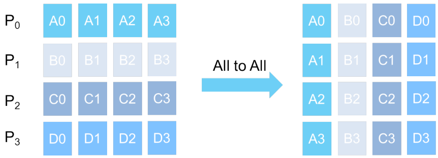
1. Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index.

MPI All to All



1. Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index.
2. MPI_Alltoall (&sendbuf,sendcount,sendtype,&recvbuf, recvcnt,recvtype,comm)

MPI All to All



1. Each task in a group performs a scatter operation, sending a distinct message to all the tasks in the group in order by index.
2. MPI_Alltoall (&sendbuf,sendcount,sendtype,&recvbuf, recvcnt,recvtype,comm)
3. Matrix transpose implementation for matrix distributed among several processors.

MPI AlltoAll

```
  int myrank, nprocs, nl, n, i, j;
  float *data, *data_l

/* local array size on each proc = nl */
  data_l = (float *) malloc(nl*sizeof(float)*nprocs);

  for (i = 0; i < nl*nprocs; ++i)
    data_l[i] = myrank;

  data = (float *) malloc(nprocs*sizeof(float)*nl);

  MPI_Alltoall(data_l, nl, MPI_FLOAT, data, nl, MPI_FLOAT,
      MPI_COMM_WORLD);
```

Summary

Summary

1. All to One: MPI_Gather, MPI_Reduce

Summary

1. All to One: MPI_Gather, MPI_Reduce
2. One to All: MPI_Scatter

Summary

1. All to One: MPI_Gather, MPI_Reduce
2. One to All: MPI_Scatter
3. All to All: MPI_AllGather, MPI_Allreduce, MPI_AlltoAll

Summary

1. All to One: MPI_Gather, MPI_Reduce
2. One to All: MPI_Scatter
3. All to All: MPI_AllGather, MPI_Allreduce, MPI_AlltoAll
4. Collective routines reduce implementation comlexity efficiently.

Summary

1. All to One: MPI_Gather, MPI_Reduce
2. One to All: MPI_Scatter
3. All to All: MPI_AllGather, MPI_Allreduce, MPI_AlltoAll
4. Collective routines reduce implementation comlexity efficiently.

Thank You

MPI: Assignments

Parallelization and Optimization Group
TATA Consultancy Services, SahyadriPark Pune, India

May 9, 2013

General Instructions

1. The assignment consists of a set of problem codes.
2. Each code is written partially.
3. The codes need to be written completely, wherever indicated with comments.
4. The codes need to be compiled and excecuted.
5. Instructions for each problem are indicated in the following slides.

Problem 1

1. Send one double value from rank 0.
2. Receive value at rank 1.
3. Print value at rank 0.

Problem 2

1. Fill arrays a[], b[] at rank 0.
2. Send arrays to rank 1.
3. Sum elements of arrays at rank 1 and print.

Problem 3

1. Broadcast array to 8 processes.
2. Print array at odd ranked processes.

Problem 4

1. Construct a NxN Matrix with each element equal to 1 and $N = 200$ on process 0.
2. Construct a Vector V of size $N = 200$ with each element equal to 1 on process 0.
3. Partition the Matrix for 8 processes and send partitioned Matrix rows to each process.
4. Send vector V to each process.
5. Mutiply partitioned Matrix rows with vector V on each process.

Problem 5

1. Fill vectors x[], y[] at rank 0.
2. Scatter them to 4 processes.
3. Compute partial dot products on each process and print.

Problem 6

1. Broadcast vector V to all processes.
2. Undertake Matrix Vector product computation on each process.
3. Gather partial results in a single vector at rank 0.

Problem 7

1. Partition two vectors (compute start point, end point for partition)
2. Compute local dot product of partitioned vectors on each process.
3. Also print the partition parameters (start point, end point) for each process.
4. Reduce local dot products to global sum at rank 0 and print the global sum.

Acknowledgements

The Parallelization and Optimization group of the TCS HPC group have created and delivered this HPC training. The specific people who have contributed are:

1. OpenMP presentation and Cache/OpenMP assignments: Anubhav Jain, Pthreads presentation: Ravi Teja.
2. Tools presentation and Demo: Rihab, Himanshu, Ravi Teja and Amit Kalele.
3. MPI presentation: Amit Kalele and Shreyas.
4. Cache assignments: Mastan Shaik.
5. Computer and Cluster Architecture and Sequential Optimization using cache.Multicore Synchronization, Multinode Infiniband introduction and general coordination and overall review: Dhananjay Brahme.

Thank You