Distributed-memory Programming with MPI

Colaboratorio Nacional de Computación Avanzada

Diego Jiménez Vargas
djimenez@cenat.ac.cr

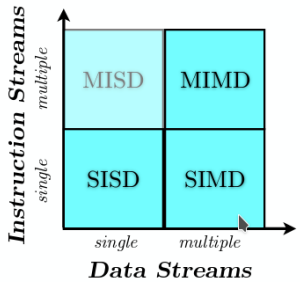
July, 2017

Contents

Distributed Memory Systems

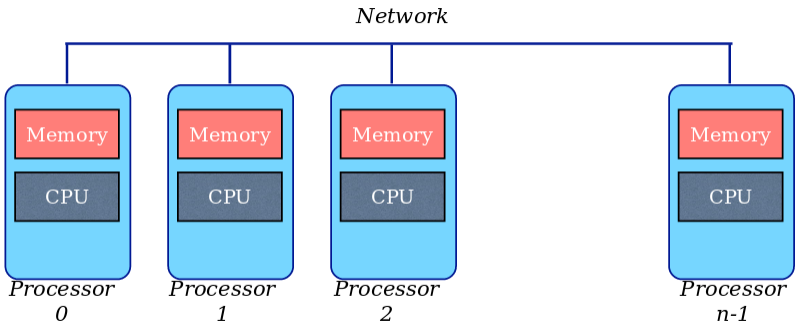
Distributed Memory Systems

- Current parallel systems have various architectures.
- **MIMD:** shared memory and **distributed memory systems**
- Why not use share-memory systems and programming models always?
  - System scaling is limited by the bus interconnect!
  - Distributed systems are better suited for problems requiring vast amount of data or computation.

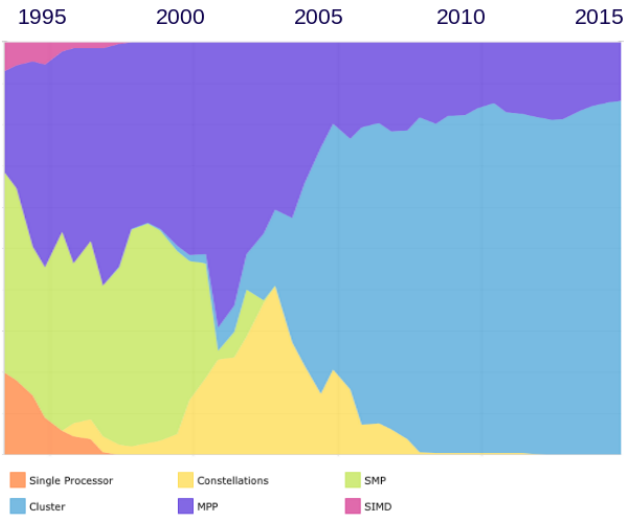


Distributed Memory Systems

- Each processor has its own private memory.
- A network connects all the processors.
- Processors can be asynchronous, programmer must impose synchronization.



Distributed Memory Systems - Top500



Parallel Programming Model

What is a Parallel Programming Model?

- An abstract machine on which parallel programs will execute.
- Components:
  - Execution model: how codes gets executed.
  - Memory model: how data moves between memory hierarchy.
- Most parallel systems expose multiple parallel programming models.

Desirable Features

- *Performant:* extracts as much performance as possible from the underlying hardware.
- *Productive:* expresses abstract algorithms easily.
- *Portable:* can be used on any computer.
- *Expressive:* allows a broad range of algorithms.
- *Scalable:* the general structure of the code persists as more hardware is used.

Implementation

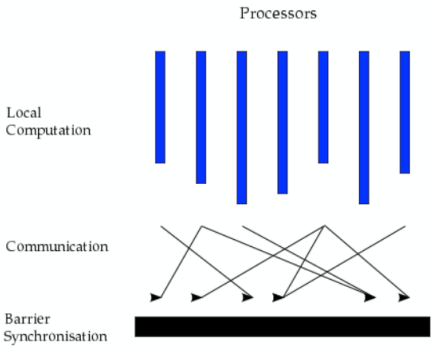
- *Library:*
  - An API of function calls.
  - Library gets linked with the executable; multiple languages.
- *Programming Language Extension:*
  - Additional constructs for parallelism.
  - Compiler support for translation.
- *New Programming Language:*
  - Design of new language grammar.
  - Flexibility to include features.

Message-Passing Paradigm

- A parallel program is decomposed into processes, called ranks.
- Each rank holds a portion of the program's data into its private memory.
- Communication among ranks is made explicit through messages.
- Channels honor first-in-first-out (FIFO) ordering.

Bulk Synchronous Parallelism

- Bridging model developed by Leslie Valiant 1980.
- Model components:
  - Processors
  - Network
  - Synchronization hardware
- Computation proceeds by executing *supersteps*:
  - Concurrent computation
  - Communication
  - Barrier Synchronization
- Concurrency level expands and shrinks.



Single-Program Multiple-Data (SPMD)

- All processes run the same program, each accesses a different portion of data.
- All processes are launched simultaneously.
- Communication:
  - Point-to-point messages.
  - Collective communication operations.
  - One-sided communication.

Features of Message Passing

- *Simplicity:* the basics of the paradigm are traditional communication operations.
- *Generality:* can be implemented on most parallel architectures.
- *Performance:* the implementation can match the underlying hardware.
- *Scalability:* the same program can be deployed on larger systems.

Message Passing Interface

Message Passing Interface

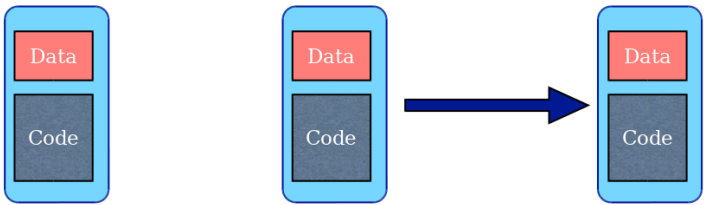
- Message-passing library interface specification.
- Standard for operations in message passing.
- Operations are expressed as functions, subroutines or methods depending on the language and implementation.
- Goals:
  - Allow efficient communication.
  - Allow overlap of communication and computation.
  - Allow for implementations that can be used in heterogeneous environments.
- Implementations:
  - Open-source: MPICH, OpenMPI.
  - Proprietary: Cray, IBM, Intel.
- Languages:
  - C/C++, Fortran, Python, Perl, Ruby, R, Java, CL, Haskell.

MPI Background

- Numerous message passing systems existed before MPI.
- MPI forum (academia and industry) was created in 1992.
  - MPI-1(1993): focused mainly on point-to-point communications.
  - MPI-2 (1997): included one-sided communications, collective communications and aspects of parallel I/O.
  - MPI-3 (2012): major update to the standard. Included non-blocking collectives and new one-sided operations.
  - MPI-3.1 (2015): minor revisions to the third version of the standard.
- Standard provides portability and ease of use!
- There is no automatic sequential-equivalent to an MPI program.

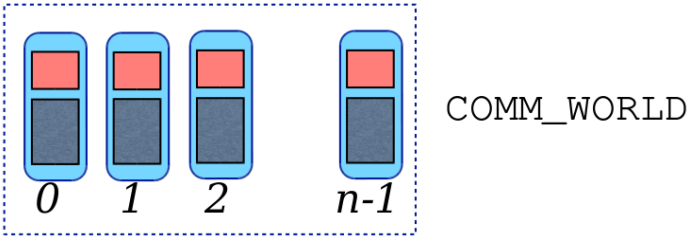
MPI Ranks

- Ranks have private memory.
- Each rank has a unique identification number.
- Ranks are numbered sequentially: [0,n-1].
- There is no automatic sequential-equivalent to an MPI program.



MPI Communicators

- Groups of ranks among which ranks can communicate.
- COMM_WORLD is a communicator including all ranks in the system.



MPI Primer

- Initialization:

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

- Get communicator size:

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

- Get rank number:

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

- Finalization:

```
int MPI_Finalize(void)
```

MPI Hello World

```
#include <mpi.h>  //Contains prototypes of MPI functions, macro and type definitions...
#include <stdio.h>
#include <stdlib.h>

// Main routine
int main (int argc, char *argv[]){
    int rank,size,length;
    char name[25];

    // initialize MPI
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &size);
    MPI_Get_processor_name(name, &length);
    printf( "Hello world from rank %d of %d on host %s\n", rank, size,name);

    // finalize MPI
    MPI_Finalize();
    return 0;
}
```

Kabré: running MPI code

- Compile application on login nodes using mpicc (wrapper for the C compiler).

```
mpicc hello.c -o hello
```

- To execute MPI applications on Kabré use a PBS script.

```
#PBS -N HelloMPI
#PBS -q phi-n2h72
#PBS -l nodes=2:ppn=4
#PBS -l walltime=0:30:00
cd $PBS_O_WORKDIR
module load mpich
mpiexec -n 8 ./hello
```

- Submit the job using queues.

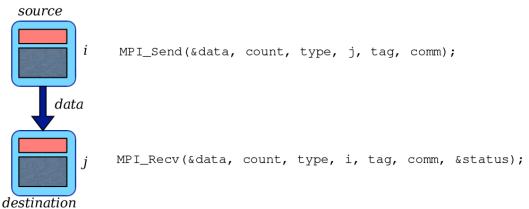
```
qsub hello.pbs
```

Point-to-point Communications

Blocking Point-to-point Communications

- Instructions to send a message from one source rank to a destination rank:

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)

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



Safe communications

If a send is not paired with a matching receive then the code will have a **deadlock**.

MPI Message Data

- Buffer specified by the communication operations consists of:
  - **count**: amount of succesive entries of the type specified.
  - **datatype**: corresponds to the basic datatypes of the host language.

MPI datatype	C datatype
MPI_CHAR	signed char
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	integer
MPI_LONG	long
MPI_BYTE	8 binary digits
MPI_PACKED	

- Type matching: the type specified by the send operation has to match the type specified by the receive operation.
- By using MPI derived types, more complex messages can be sent (values with different datatypes or noncontiguous data).

MPI Message Envelope

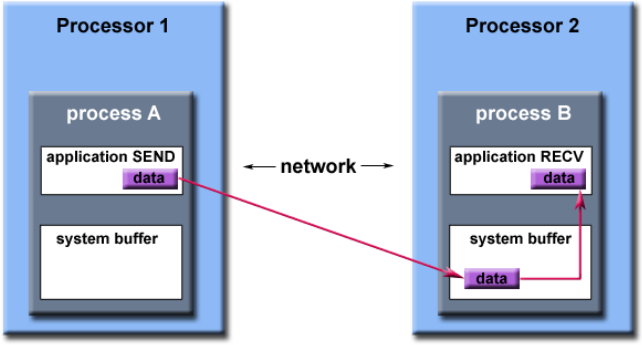
- Messages carry information that helps distinguish them and selectively receive data:
  - **source**
  - **destination**
  - **tag**: identifier to filter or order received messages.
  - **communicator**: communication context for an operation. Messages are always received within the context they were sent, and messages sent in different contexts do not interfere.
- A message is received by a receive operation only if its envelope matches the source, tag and communicator values specified in the operation.
- MPI provides certain wild card values: MPI_ANY_SOURCE and MPI_ANY_TAG.

Send-Receive Example

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

int main(int argc, char ** argv)
{
    int rank, data[3];
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0) {
        data[0] = 0; data[1] = 10; data[2] = 20;
        MPI_Send(data, 3, MPI_INT, 1, 0, MPI_COMM_WORLD);
    }

    else if (rank == 1) {
        data[0] = -1; data[1] = -1; data[2] = -1;
        printf("[%d] before receiving: %d,%d,%d\n", rank, data[0], data[1], data[2]);

        MPI_Recv(data, 3, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

        printf("[%d] after receiving: %d,%d,%d\n", rank, data[0], data[1], data[2]);
    }

    MPI_Finalize();
    return 0;

}
```

MPI under the hood

- MPI implementation must be able to deal with storing data when two or more processes or ranks are out of sync.
- A system buffer area is reserved to hold data in transit.



Communication modes

- **Blocking mode** (standard): send operation does not return until the message data and envelope have been safely stored away and the sender is free to modify the buffer.

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm)
```

- **Buffered mode**: if a send is executed and no matching receive is posted, MPI buffers the outgoing message to allow the send call to complete. Available buffer space is controlled by the user.

```
int MPI_Bsend(const void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm)
```

Communication modes

- **Synchronous mode**: send operation will complete successfully only if a matching receive is posted, and the receive operation has started to receive the message.

```
int MPI_Ssend(const void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm)
```

- **Ready mode**: send operation may be started *only* if a matching receive is already posted. This allows for the removal of a hand-shake operation that is otherwise required and results in improved performance.

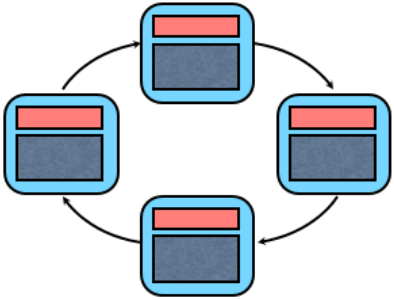
```
int MPI_Rsend(const void* buf, int count, MPI_Datatype datatype, int dest,
int tag, MPI_Comm comm)
```

Send-Receive exercise

- Implement a parallel ping-pong in MPI.
  - A message carrying an integer counter is exchanged between two ranks.
  - Only rank 1 will increment the counter upon reception.
  - Repeat the process 1000 times.

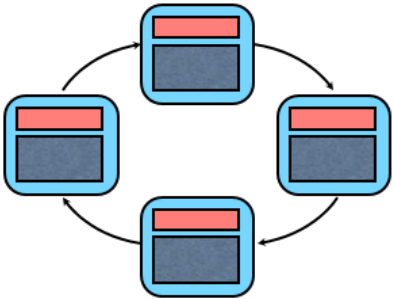
Send-Receive exercise

- Implement a parallel MPI program that creates a ring of ranks.
  - Each rank gets a random integer value [0,p].
  - The program computes in each rank the sum of all the values by circulating them around the ring.



Send-Receive exercise

- Using the implemented processors ring:
  - Instead of passing an scalar, try passing an array.
  - First pass an array of size 100
  - Try setting the size of the array to 100 000. What happens?



Problems with blocked communication

- On rank 0:

```
MPI_Send( buf, count, type, 1, tag, comm);
MPI_Recv( buf, count, type, 1, tag, comm, &status);
```

- On rank 1:

```
MPI_Send( buf, count, type, 0, tag, comm);
MPI_Recv( buf, count, type, 0, tag, comm, &status);
```

- A blocking send will only "return" after it is safe to modify the application buffer for reuse.
- A blocking send can be synchronous which means there is handshaking occurring with the receive task to confirm a safe send.

Solving deadlocks

1. Reorganize send and receive operations.
   - Even ranks send first.
   - Odd ranks receive first.
2. Let MPI take care of the problem:
   - MPI_Sendrecv
   - MPI_Sendrecv_replace
3. Use nonblocking communication.

Nonblocking Point-to-point Communications

- Performance can be boosted by overlapping communication and computation.
- Nonblocking operations simply request the MPI implementation to perform the operation when it is able, i.e they do not wait for the operation to complete.
- It's unsafe to modify the application buffer (variable) until we verify that the nonblocking operation has been completed.
- At least two function calls:
  1. Call to start operation.
  2. Call to complete the operation.

Nonblocking Point-to-point Communications

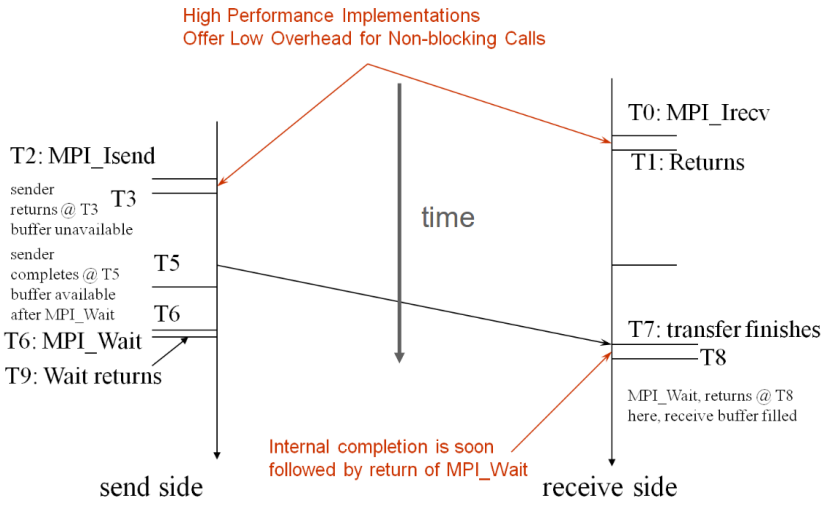
- Nonblocking operations return "request handles" that can be waited on and queried.

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

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

int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

- Completion can also be verified using the nonblocking function MPI_Test.

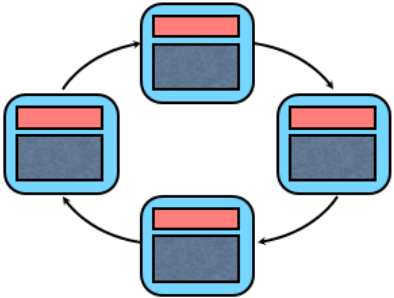
```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)
```

Nonblocking Point-to-point Communications



Isend-Ireceive exercise

- Adjust the deadlocked ring, implemented in the blocking communication section, so that it now uses non blocking point-to-point communications:
  - First pass an array of size 100
  - Try setting the size of the array to 100 000. What happens now?



Collective Communication Operations

Collective Communication Operations

- Instructions to exchange data including all the ranks in a communicator.
- The root rank indicates the source or destination of the operation.
- All ranks in the communicator must call the collective operation.
- Collectives can help implement different communication patterns:
  - One-to-many
  - Many-to-one
  - Many-to-many
- Collectives can serve different purposes:
  - Data movement
  - Collective computation
  - Synchronization

Collective Communication Operations

- *Broadcast*: one-to-many

```
int MPI_Bcast(void *buffer, int count, MPI_Datatype datatype,
   int root, MPI_Comm comm)
```

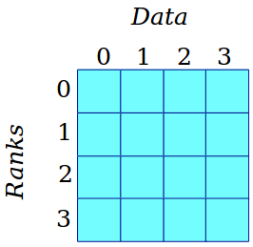
- *Reduce*: many to one.

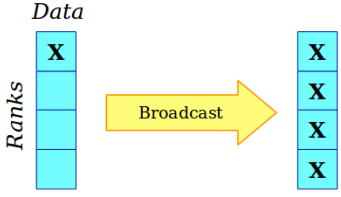
```
int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
      MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

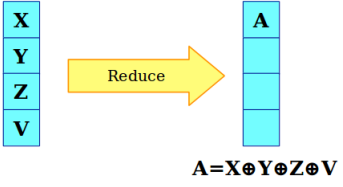
- *Barrier*: synchronization.

```
int MPI_Barrier(MPI_Comm comm)
```

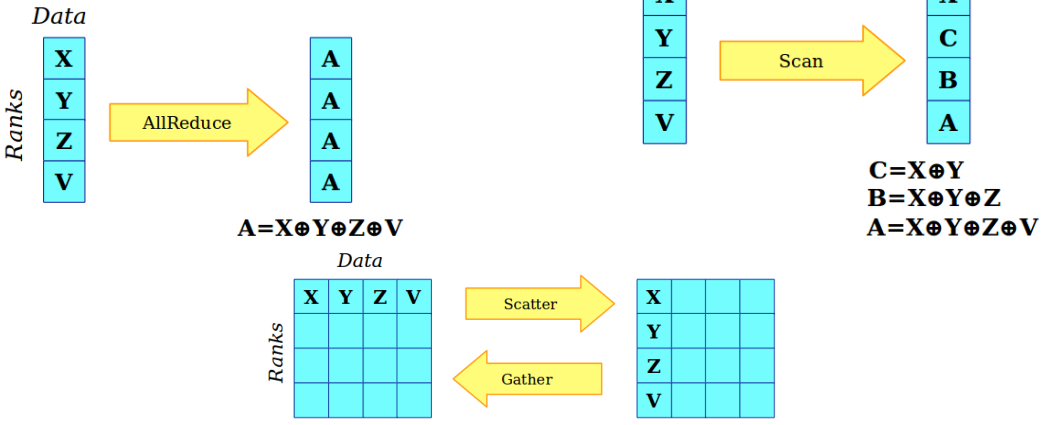
Basic Collectives



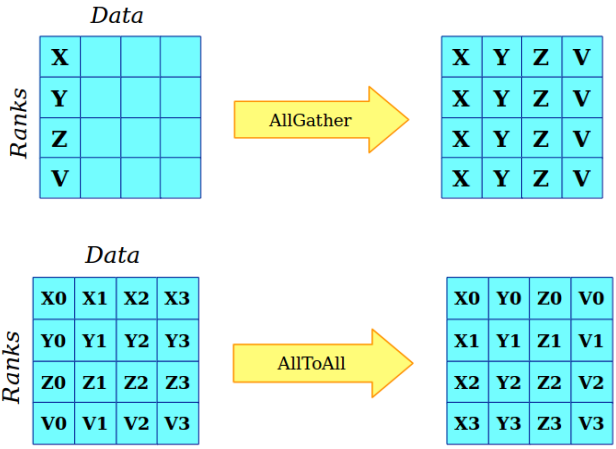




Basic Collectives



Basic Collectives



Basic Collectives

```
int MPI_Allreduce(const void *sendbuf, void *recvbuf,
    int count, MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
int MPI_Scan(const void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, MPI_Comm comm)
int MPI_Scatter(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, int recvcount,
    MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_Gather(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, int recvcount,
    MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_Allgather(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, int recvcount,
    MPI_Datatype recvtype, MPI_Comm comm)
int MPI_Alltoall(const void *sendbuf, int sendcount,
    MPI_Datatype sendtype, void *recvbuf, int recvcount,
    MPI_Datatype recvtype, MPI_Comm comm)
```

Predefined Reduction Operations

Name	Meaning
MPI_MAX	maximum
MPI_MIN	minimum
MPI_SUM	sum
MPI_PROD	product
MPI_LOR	logical OR
MPI_LAND	logical AND
MPI_MAXLOC	max value and location

Collectives example: Sequential code

```
#include <stdio.h>

int main(int argc, char *argv[]){
        int i, sum, upToVal;
        upToVal = 10000;
        sum = 0;

        for(i=1; i <= upToVal; i++){
                sum = sum +i;
        }
        printf("\nSum is %d\n", sum);
    return 0;
}
```

Collectives example: Parallel code

```
#include <stdio.h>
#include "mpi.h"

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

        int i, sum, sumTotal, upToVal;
        int start, end, size, myRank;

        upToVal = 10000;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &size);
        MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

        start = myRank*(upToVal/size)+1;

        if(myRank == (size-1)){
                end = upToVal;
        }else{
               end = start + (upToVal/size)-1;
        }
```

Collectives example: Parallel code

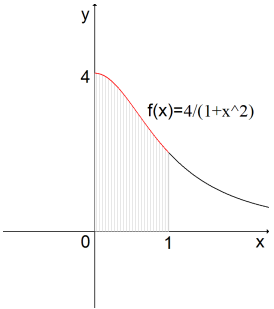
```
        sum = 0;
        sumTotal = 0;

        for(i=start; i <= end; i++){
                sum = sum +i;
        }

        MPI_Reduce(&sum, &sumTotal, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

        printf("\nRank: %d, local sum: %d, total sum: %d\n", myRank, sum, sumTotal);

        MPI_Finalize();

        return 0;

}
```

Collective communication exercise

- Write a parallel MPI program where each rank gets a random integer value [0,p]. The program gets in each rank the sum of all the values in the ranks using only broadcast and reduction.

Collective communication exercise

- Approximating pi via numerical integration



$$f(x) = \frac{4}{1 + x\text{^}2}$$

$$\int_0^1 f(x)dx = \int_0^1 \frac{4}{1+x^2}dx = \int_0^{\pi/4} \frac{4}{1+\tan\theta\ \text{^}2}d\theta = \int_0^{\pi/4} 4d\theta = \pi$$

$$PI = \int_0^1 f(x)dx = \int_0^1 \frac{4}{1+x^2}dx$$

Collective communication exercise

- Calculating pi using trapezoidal rule:
  - Divide interval up into subintervals.
  - Assign subintervals to processes
  - Each process calculates partial sum
  - Add all the partial sums together to get pi.

- Subinterval information:
  - Width of each subinterval (w) will be 1/n
  - Distance d(i) of segment "i" from the origin will be "i*w"
  - Height of segment "i" will be f(i)

Extra exercise

- Try to implement a matrix-matrix multiplication algorithm using MPI.
  1. Do it using only MPI_Send and MPI_Recv.
  2. Do it using MPI collectives (hint: MPI_Scatter, MPI_Gather may be useful)

$$\mathbf{A} = \begin{pmatrix} a & b & c \\ p & q & r \\ u & v & w \end{pmatrix}, \quad \mathbf{B} = \begin{pmatrix} \alpha & \beta & \gamma \\ \lambda & \mu & \nu \\ \rho & \sigma & \tau \end{pmatrix}$$

$$\mathbf{AB} = \begin{pmatrix} a & b & c \\ p & q & r \\ u & v & w \end{pmatrix} \begin{pmatrix} \alpha & \beta & \gamma \\ \lambda & \mu & \nu \\ \rho & \sigma & \tau \end{pmatrix} = \begin{pmatrix} a\alpha + b\lambda + c\rho & a\beta + b\mu + c\sigma & a\gamma + b\nu + c\tau \\ p\alpha + q\lambda + r\rho & p\beta + q\mu + r\sigma & p\gamma + q\nu + r\tau \\ u\alpha + v\lambda + w\rho & u\beta + v\mu + w\sigma & u\gamma + v\nu + w\tau \end{pmatrix}$$

Timing parallel execution

- Main goal: improving performance.
- Elapsed or wall time is used to measure speedup and efficiency, not CPU time.
- MPI provides timing functions

```
1       double start, finish;
2
3       start = MPI_Wtime();
4               .
5               .
6       /*Code being timed*/
7               .
8               .
9       finish = MPI_Wtime();
10      printf("Elapsed time:  %f seconds\n",finish-start);
```

- Minimize communication, maximize computation!

Concluding Remarks

Concluding Remarks

- MPI is an industry standard model for parallel programming.
  - Gives the programmer great parallelism power.
  - The programmer has to explictly indicate and control data movement and synchronization.
- Performance usually sensitive to assignment of tasks to processors due to concurrency, workload balance, communication patterns, and more.
- Special caution must be taken to avoid unsafe communications and possible deadlocks.
- Overlapping communication and computation is key to reducing execution times and improving performance.
- Not all problems are suitable for parallelization using the message-passing model.

To Infinity and Beyond

- Hybrid applications: MPI + X (OpenMP, PThreads, ACC)
- One-sided communications: operations that do not require both ends to synchronize during communication.
- Non-blocking collectives.
- Communicators and topologies: mapping virtual topologies to underlying physical structure.
- Neighborhood collectives.

Acknowledgements

Thank you!

- Lecture notes by Prof. Esteban Meneses.
- Notes from *Introduction to MPI* by Argonne National Laboratory