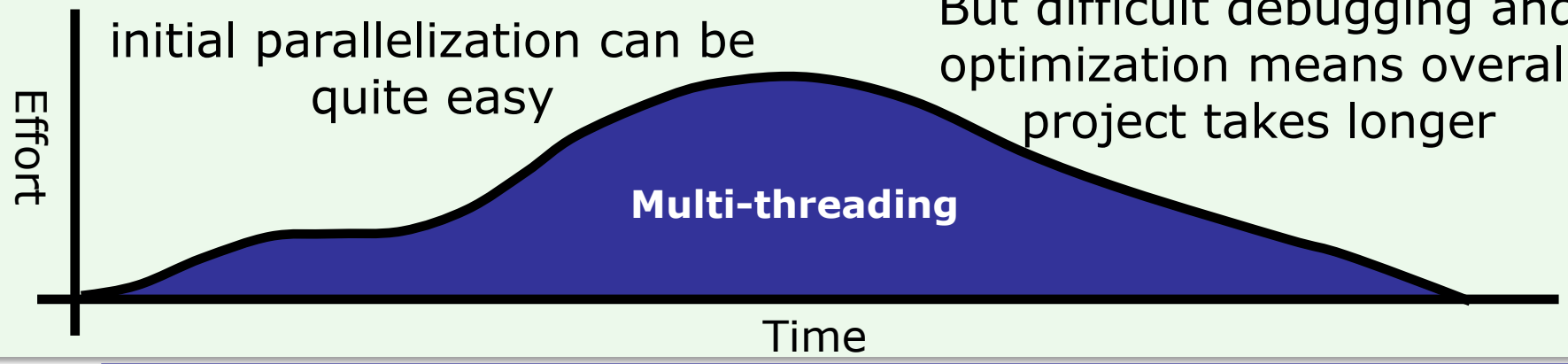
# Parallel Programming

## Distributed-Memory Programming with MPI

## (Part I)

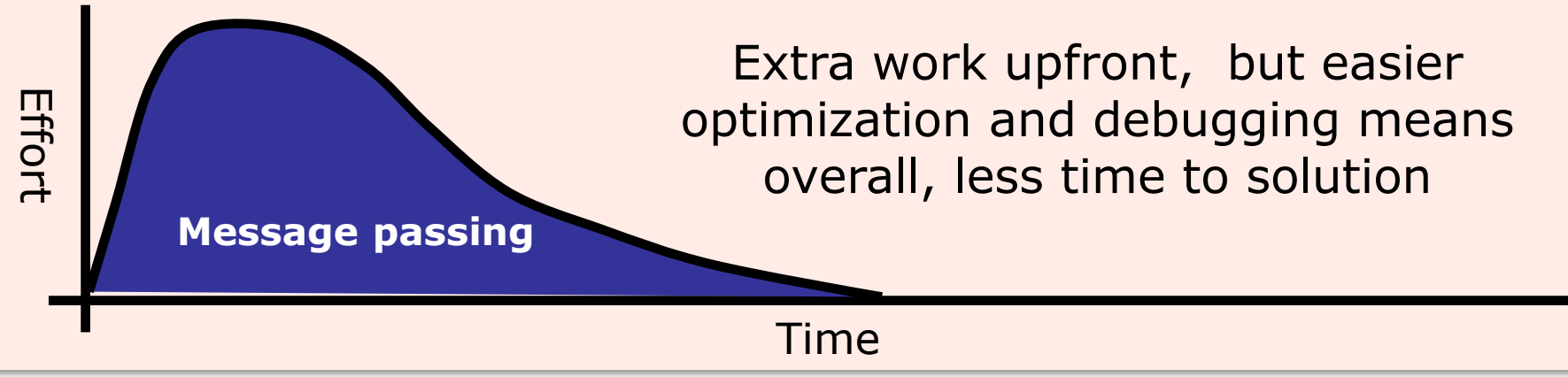
Professor Yi-Ping You (游逸平)
Department of Computer Science
http://www.cs.nctu.edu.tw/~ypyou/

# Multithreading vs Message Passing




Proving that a shared address space program using semaphores is race free is an NP-complete problem

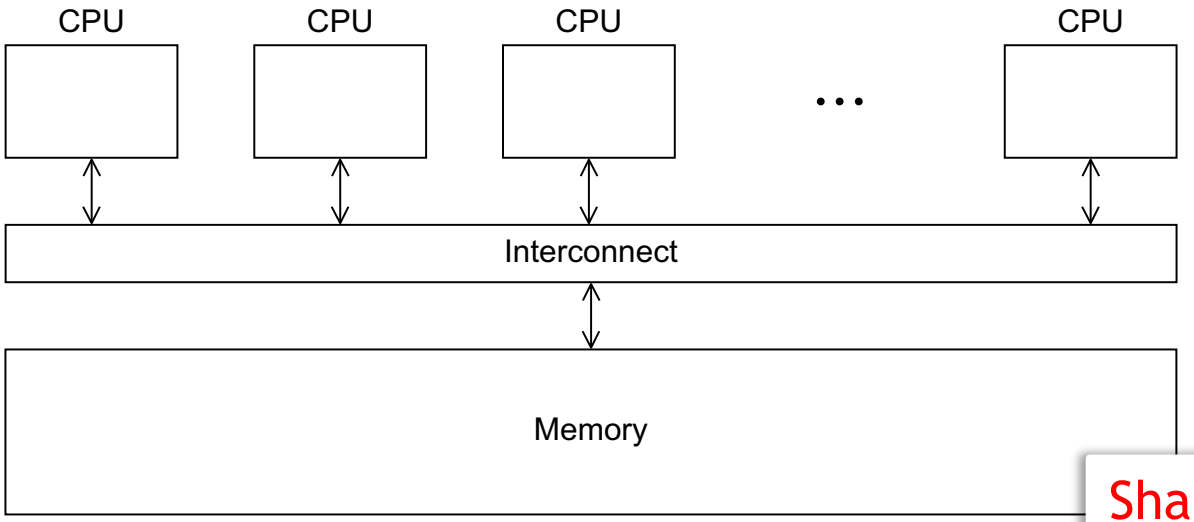
P. N. Klein, H. Lu, and R. H. B. Netzer, Detecting Race Conditions in Parallel Programs that Use Semaphores, Algorithmica, vol. 35 pp. 321-345, 2003



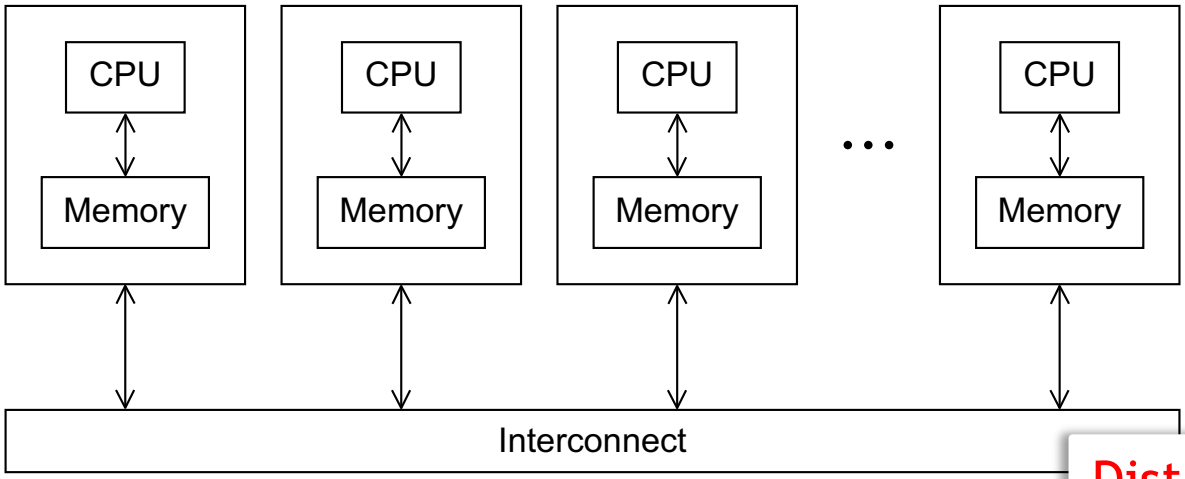

We should just abandon threading?

Source: Tim Mattson, "Recent developments in parallel programming: the good, the bad, and the ugly", Euro-Par 2013

# Shared Memory vs Distributed Memory

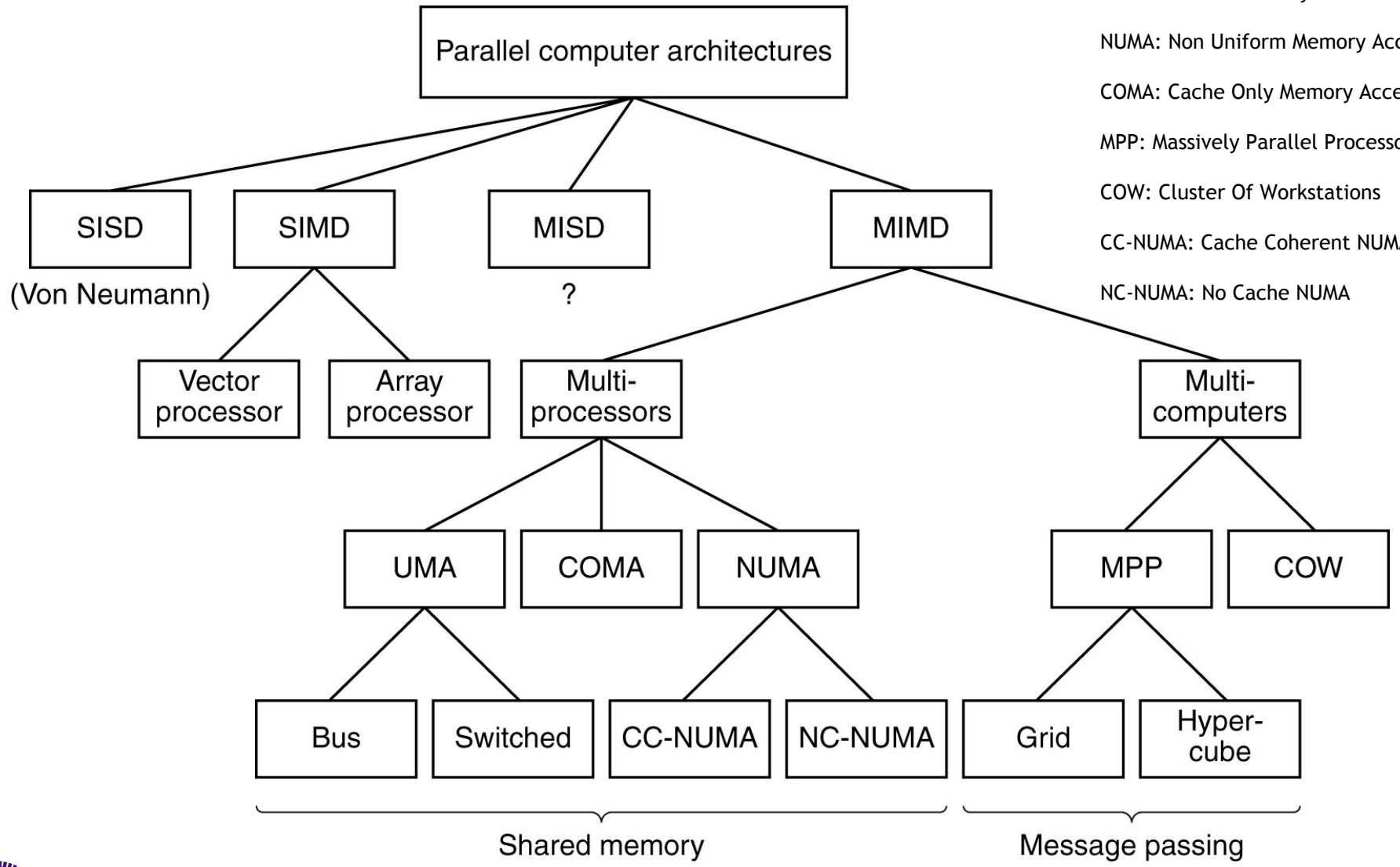



Shared-memory system




Distributed-memory system

# Taxonomy of Parallel Architectures




UMA: Uniform Memory Access

NUMA: Non Uniform Memory Access

COMA: Cache Only Memory Access

MPP: Massively Parallel Processor

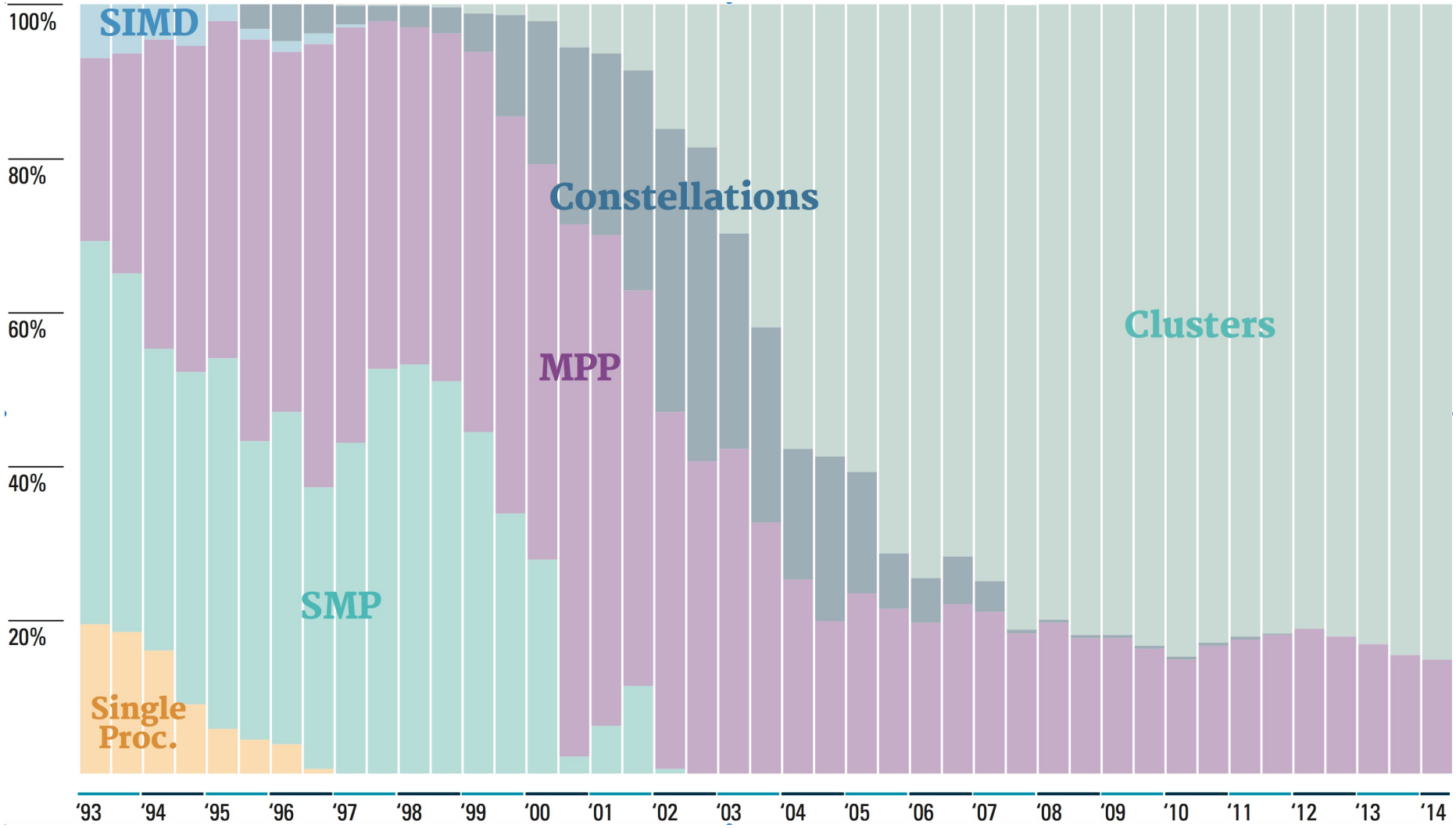
COW: Cluster Of Workstations

CC-NUMA: Cache Coherent NUMA

NC-NUMA: No Cache NUMA

Source: Tanenbaum, Structured Computer Organization

# TOP500: System Architectures




Source: http://s.top500.org/static/lists/2014/06/TOP500_201406_Poster.pdf

# Acknowledgement


- The following set of slides is mainly developed by Prof. Thomas Sterling and Dr. Steve Brandt from Louisiana State University


# Topics

- Introduction
- MPI Standard
- MPI-1 Model and Basic Calls
- MPI Communicators
- Point to Point Communication
- Point to Point Communication in-depth
- Deadlock
- Trapezoidal Rule : A Case Study
- Using MPI & Jumpshot to profile parallel applications
- Summary

# Topics

- Introduction
- MPI Standard
- MPI-1 Model and Basic Calls
- MPI Communicators
- Point to Point Communication
- Point to Point Communication in-depth
- Deadlock
- Trapezoidal Rule : A Case Study
- Using MPI & Jumpshot to profile parallel applications
- Summary

# Opening Remarks

- Context: distributed memory parallel computers
- We have communicating sequential processes, each with their own memory, and no access to another process's memory
  - A fairly common scenario from the mid 1980s (Intel Hypercube) to today
  - Processes interact (exchange data, synchronize) through message passing
  - Initially, each computer vendor had its own library and calls
  - First standardization was PVM (Parallel Virtual Machine)
    - Started in 1989, first public release in 1991
    - Worked well on distributed machines
    - Next was MPI (1994)

# What you'll Need to Know

- What is a standard API
- How to build and run an MPI-1 program
- Basic MPI functions
  - 4 basic environment functions
    - Including the idea of communicators
  - Basic point-to-point functions
    - Blocking and non-blocking
    - Deadlock and how to avoid it
    - Data types
  - Basic collective functions

# Topics

- Introduction
- MPI Standard
- MPI-1 Model and Basic Calls
- MPI Communicators
- Point to Point Communication
- Point to Point Communication in-depth
- Deadlock
- Trapezoidal Rule : A Case Study
- Using MPI & Jumpshot to profile parallel applications
- Summary

# MPI Standard

- From 1992-1994, a community representing both vendors and users decided to create a standard interface to message passing calls in the context of distributed memory parallel computers (MPPs, there weren't really clusters yet)

- MPI-1 was the result
  - "Just" an API
  - FORTRAN77 and C bindings
  - Reference implementation (mpich) also developed
  - Vendors also kept their own internals (behind the API)
  - SPMD programming

# MPI Standard

- Since then
  - MPI-1.1
    - Fixed bugs, clarified issues
  - MPI-2
    - Included MPI-1.2
      - Fixed more bugs, clarified more issues
    - Extended MPI
      - New datatype constructors, language interoperability
    - New functionality
      - One-sided communication
      - MPI I/O
      - Dynamic processes
    - FORTRAN90 and C++ bindings
  - There are over 330 functions in the MPI spec, but most programs only use a small subset
- Best MPI reference
  - MPI Standard - on-line at: http://www.mpi-forum.org/

# MPI (1992-today)

- The message passing interface (MPI) is a standard library
- MPI Forum first met April 1992,
  - MPI 1.0 in June 1994
  - MPI 2.0 in July 1997
  - MPI 3.0 in September 2012
  - MPI 3.1 in June 2015
  - MPI 4.0 in June 2021
- Hardware-portable, multi-language communication library
- Enabled billions of dollars of applications
- PI implementations
  - MPICH (http://www.mpich.org/)
  - Open MPI (https://www.open-mpi.org/)

# Topics

- Introduction
- MPI Standard
- MPI-1 Model and Basic Calls
- MPI Communicators
- Point to Point Communication
- Point to Point Communication in-depth
- Deadlock
- Trapezoidal Rule : A Case Study
- Using MPI & Jumpshot to profile parallel applications
- Summary

# MPI : Basics

- Every MPI program must contain the preprocessor directive

```
#include "mpi.h"
```

- The mpi.h file contains the definitions and declarations necessary for compiling an MPI program.
- mpi.h is usually found in the "include" directory of most MPI installations. For example on arete:

```
lsu00@master:~
File  Edit  View  Terminal  Help
[lsu00@master ~]$ ls /usr/include/mpich2-x86_64/
clog_commset.h   mpe_graphicsf.h  mpe_log_thread.h  mpif.h   mpi_sizeofs.mod   primitives
clog_const.h     mpe_graphics.h   mpe_misc.h        mpi.h    opa_config.h
clog_inttypes.h  mpe.h            mpi_base.mod      mpi.mod  opa_primitives.h
clog_uuid.h      mpe_logf.h       mpi_constants.mod mpiof.h  opa_queue.h
mpe_callstack.h  mpe_log.h        mpicxx.h          mpio.h   opa_util.h
[lsu00@master ~]$
```

```
...
#include "mpi.h"
...
MPI_Init(&Argc,&Argv);
...
...
MPI_Finalize();
...
```

# MPI: Initializing MPI Environment

Function: MPI_init()

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

Description:

Initializes the MPI execution environment. MPI_init() must be called before any other MPI functions can be called and it should be called only once. It allows systems to do any special setup so that MPI Library can be used. ***argc*** is a pointer to the number of arguments and ***argv*** is a pointer to the argument vector. On exit from this routine, all processes will have a copy of the argument list.

```
...
#include "mpi.h"
...
MPI_Init(&argc,&argv);
...
...
MPI_Finalize();
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Init.html

# MPI: Terminating MPI Environment

Function: MPI_Finalize()

int MPI_Finalize()

Description:

Terminates MPI execution environment. All MPI processes must call this routine before exiting. MPI_Finalize() need not be the last executable statement or even in main; it must be called at somepoint following the last call to any other MPI function.

```
...
#include "mpi.h"
...
MPI_Init(&argc,&argv);
...
...
MPI_Finalize();
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Finalize.html

# MPI Hello World

- C source file for a simple MPI Hello World

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

int main( int argc, char *argv[])
{
    MPI_Init( &argc, &argv);
    printf("Hello, World!\n");
    MPI_Finalize();
    return 0;
}
```

Include header files

Initialize MPI Context

Finalize MPI Context

MPI_Init and MPI_Finalize "bracket" every MPI program

# Building an MPI Executable

- Library version
  - User knows where header file and library are, and tells compiler

```
gcc -Iheaderdir -Llibdir mpicode.c –lmpich
```

- Wrapper version
  - Does the same thing, but hides the details from the user

```
mpicc -o executable mpicode.c
```

You can do **either one**, but don't try to do both!

- use ”mpicc -compile_info -o executable mpicode.c" to figure out the gcc line
- For Open MPI:

```
# Show the flags necessary to compile MPI C applications
shell$ mpicc --showme:compile
# Show the flags necessary to link MPI C applications
shell$ mpicc --showme:link
```

For our “Hello World” example on arete use:

```
gcc -m64 -O2 -fPIC -Wl,-z,noexecstack -o hello
hello.c -I/usr/include/mpich2-x86_64
-L/usr/lib64/mpich2/lib -L/usr/lib64/mpich2/lib
-Wl,-rpath,/usr/lib64/mpich2/lib -lmpich -lopa
-lpthread -lrt
```

OR `mpicc -o hello hello.c`

# Running an MPI Executable

- Some number of processes are started somewhere
  - Standard doesn't talk about this
  - Implementation and interface varies
  - Usually, some sort of `mpiexec` command starts some number of copies of an executable according to a mapping
  - Example:

    '`mpiexec -n 2 ./a.out`' command runs two copies of `./a.out` where the system specifies number of processes to be 2
  - Most production supercomputing resources wrap the mpiexec command with higher level scripts that interact with scheduling systems such as PBS / LoadLeveler for efficient resource management and multi-user support
  - Sample PBS / Load Leveler job submission scripts :

**PBS File:**

```
#!/bin/bash
#PBS -l walltime=120:00:00,nodes=8:ppn=4
cd /home/cdekate/S1_L2_Demos/adc/
pwd
date
PROCS=`wc -l < $PBS_NODEFILE`
mpdboot  --file=$PBS_NODEFILE
mpiexec -n $PROCS ./padcirc
mpdallexit
date
```

**LoadLeveler File:**

```
#!/bin/bash
#@ job_type = parallel
#@ job_name = SIMID
#@ wall_clock_limit = 120:00:00
#@ node = 8
#@ total_tasks = 32
#@ initialdir = /scratch/cdekate/
#@ executable = /usr/bin/poe
#@ arguments = /scratch/cdekate/padcirc
#@ queue
```

# Running the Hello World example

- Using mpiexec :

```
mpd &
mpiexec  -n 8  ./hello
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
```

mpiexec -n 8 **-host node1,node2** ./hello

mpiexec -n 8 **-hostfile <hostfile>** ./hello

- Using PBS

**hello.pbs :**

```
#!/bin/bash
#PBS -N hello
#PBS -l walltime=00:01:00,nodes=2:ppn=4
cd /home/cdekate/2008/l7
wd
date
PROCS=`wc -l < $PBS_NODEFILE`
mpdboot  -f $PBS_NODEFILE
mpiexec -n $PROCS ./hello
mpdallexit
date
```

**more hello.o10030**

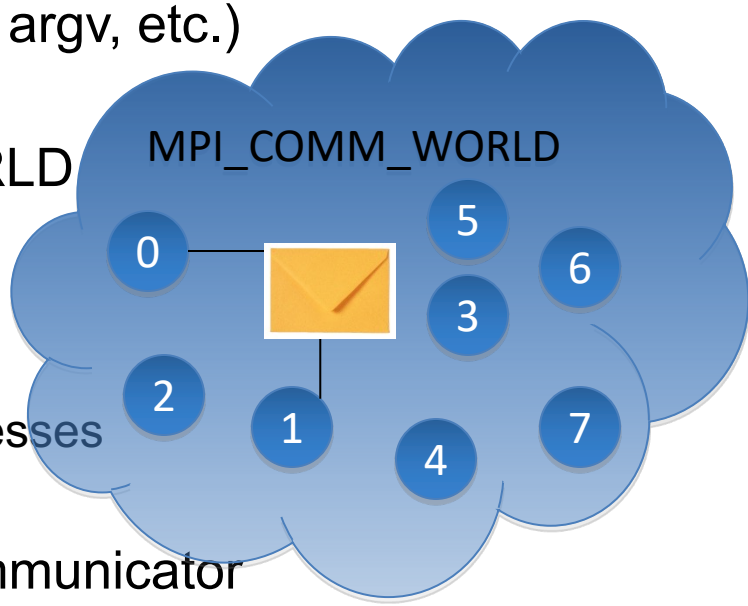
```
/home/cdekate/2008/l7
Wed Feb  6 10:58:36 CST 2008
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Hello, World!
Wed Feb  6 10:58:37 CST 2008
```

# Topics

- Introduction
- MPI Standard
- MPI-1 Model and Basic Calls
- MPI Communicators
- Point to Point Communication
- Point to Point Communication in-depth
- Deadlock
- Trapezoidal Rule : A Case Study
- Using MPI & Jumpshot to profile parallel applications
- Summary

# MPI Communicators

- Communicator is an internal object
- MPI Programs are made up of communicating processes
- Each process has its own address space containing its own attributes such as rank, size (and argc, argv, etc.)
- MPI provides functions to interact with it
- Default communicator is MPI_COMM_WORLD
  - All processes are its members
  - It has a size (the number of processes)
  - Each process has a rank within it
  - One can think of it as an ordered list of processes
- Additional communicator(s) can co-exist
- A process can belong to more than one communicator
- Within a communicator, each process has a unique rank



# MPI: Size of Communicator

Function: MPI_Comm_size()

int MPI_Comm_size ( MPI_Comm comm, int *size )

Description:

Determines the size of the group associated with a communicator (comm). Returns an integer number of processes in the group underlying *comm* executing the program. If *comm* is an inter-communicator (i.e. an object that has processes of two inter-communicating groups) , return the size of the local group (a size of a group where request is initiated from). The *comm* in the argument list refers to the communicator-group to be queried, the result of the query (size of the *comm* group) is stored in the variable size.

```
...
#include "mpi.h"
...
int size;
MPI_Init(&Argc,&Argv);
...
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
err = MPI_Finalize();
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Comm_size.html

# MPI: Rank of a process in comm

Function: MPI_Comm_rank()

int MPI_Comm_rank ( MPI_Comm comm, int *rank )

Description:

Returns the rank of the calling process in the group underlying the comm. If the *comm* is an inter-communicator, the call MPI_Comm_rank returns the rank of the process in the local group. The first parameter *comm* in the argument list is the communicator to be queried, and the second parameter *rank* is the integer number rank of the process in the group of *comm*.

```
...
#include "mpi.h"
...
int rank;
MPI_Init(&Argc,&Argv);
...
MPI_Comm_size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
...
err = MPI_Finalize();
...
```

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Comm_rank.html

# Example : communicators

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

int main( int argc, char *argv[])
{
    int rank, size;
    MPI_Init( &argc, &argv);
    MPI_Comm_rank( MPI_COMM_WORLD, &rank);
    MPI_Comm_size( MPI_COMM_WORLD, &size);
    printf("Hello, World! from %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

Determines the rank of the current process in the communicator-group MPI_COMM_WORLD

Determines the size of the communicator-group MPI_COMM_WORLD

```
…
Hello, World! from 1 of 8
Hello, World! from 0 of 8
Hello, World! from 5 of 8
…
```

# Example : Communicator & Rank

- Compiling :

```
mpicc -o hello2 hello2.c
```

- Result :

```
mpirun -np 8 hello2   or mpiexec -n 8 hello2

Hello, World! from 4 of 8
Hello, World! from 3 of 8
Hello, World! from 1 of 8
Hello, World! from 0 of 8
Hello, World! from 5 of 8
Hello, World! from 6 of 8
Hello, World! from 7 of 8
Hello, World! from 2 of 8
```

# Topics

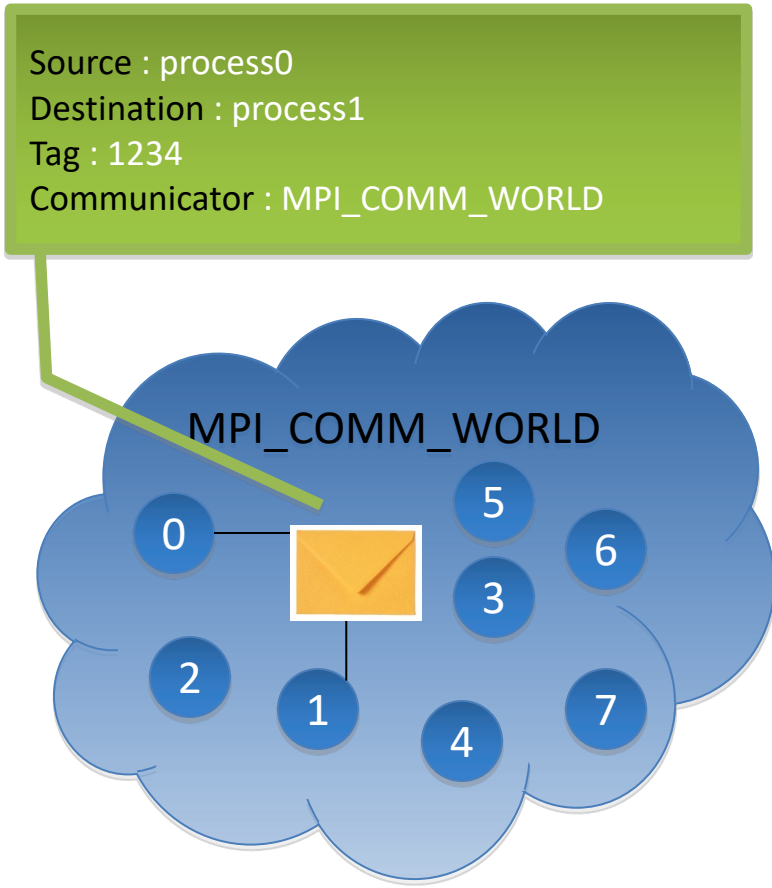
- Introduction
- MPI Standard
- MPI-1 Model and Basic Calls
- MPI Communicators
- Point to Point Communication
- Point to Point Communication in-depth
- Deadlock
- Trapezoidal Rule : A Case Study
- Using MPI & Jumpshot to profile parallel applications
- Summary

# MPI : Point to Point Communication primitives

- A basic communication mechanism of MPI between a pair of processes in which one process is sending data and the other process receiving the data, is called "*point to point communication*"
- Message passing in MPI program is carried out by 2 main MPI functions
  - **MPI_Send** – sends message to a designated process
  - **MPI_Recv** – receives a message from a process
- Each of the *send* and *recv* calls is appended with additional information along with the data that needs to be exchanged between application programs
- The message envelope consists of the following information
  - The rank of the receiver
  - The rank of the sender
  - A tag
  - A communicator
- The source argument is used to distinguish messages received from different processes
- Tag is user-specified *int* that can be used to distinguish messages from a single process

# Message Envelope

- Communication across processes is performed using messages.
- Each message consists of a fixed number of fields that is used to distinguish them, called the Message Envelope :
  - Envelope comprises **source, destination, tag, communicator**
  - Message = Envelope + Data
- Communicator refers to the namespace associated with the group of related processes



# MPI: (blocking) Send message

Function: MPI_Send()

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

Description:

The contents of message are stored in a block of memory referenced by the first parameter *message*. The next two parameters, count and datatype, allow the system to determine how much storage is needed for the message: the message contains a sequence of *count* values, each having *MPI* type *datatype.* MPI allows a message to be received as long as there is sufficient storage allocated. If there isn't sufficient storage an overflow error occurs. The *dest* parameter corresponds to the rank of the process to which message has to be sent.

http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Send.html

# MPI : Data Types

| MPI datatype | C datatype |
|---|---|
| MPI_CHAR | signed char |
| MPI_SHORT | signed short int |
| MPI_INT | signed int |
| MPI_LONG | signed long int |
| MPI_UNSIGNED_CHAR | unsigned char |
| MPI_UNSIGNED_SHORT | unsigned short int |
| MPI_UNSIGNED | unsigned int |
| MPI_UNSIGNED_LONG | unsigned long int |
| MPI_FLOAT | float |
| MPI_DOUBLE | double |
| MPI_LONG_DOUBLE | long double |
| MPI_BYTE | |
| MPI_PACKED | |

You can also define your own (derived datatypes), such as an array of ints of size 100, or more complex examples, such as a struct or an array of structs

# MPI: (blocking) Receive message

Function: MPI_Recv()

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

Description:

The contents of message are stored in a block of memory referenced by the first parameter *message*. The next two parameters, count and datatype, allow the system to determine how much storage is needed for the message: the message contains a sequence of *count* values, each having *MPI* type *datatype.* MPI allows a message to be received as long as there is sufficient storage allocated. If there isnt sufficient storage an overflow error occurs. The *source* parameter corresponds to the rank of the process from which the message has been received. The MPI_Status parameter in the MPI_Recv() call returns information on the data that was actually received It references a record with 2 fields – one for the source and one for the tag http://www-unix.mcs.anl.gov/mpi/www/www3/MPI_Recv.html

# MPI_Status object

Object: MPI_Status

Example usage :

```
MPI_Status    status;
```

Description:

The MPI_Status object is used by the receive functions to return data about the message, specifically the object contains the id of the process sending the message (MPI_SOURCE), the message tag (MPI_TAG), and error status (MPI_ERROR) .

```
#include "mpi.h"
…
MPI_Status status; /* return status for   */
…
MPI_Init(&argc, &argv);
…
if (my_rank != 0) {
…
    MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }
  else { /* my rank == 0 */
    for (source = 1; source < p; source++ ) {
      MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
      MPI_Get_count(&status,recvtype,&count);
      … status.MPI_SOURCE…status.MPI_TAG…;
      …
  MPI_Finalize();
…
```

# MPI : Example send/recv

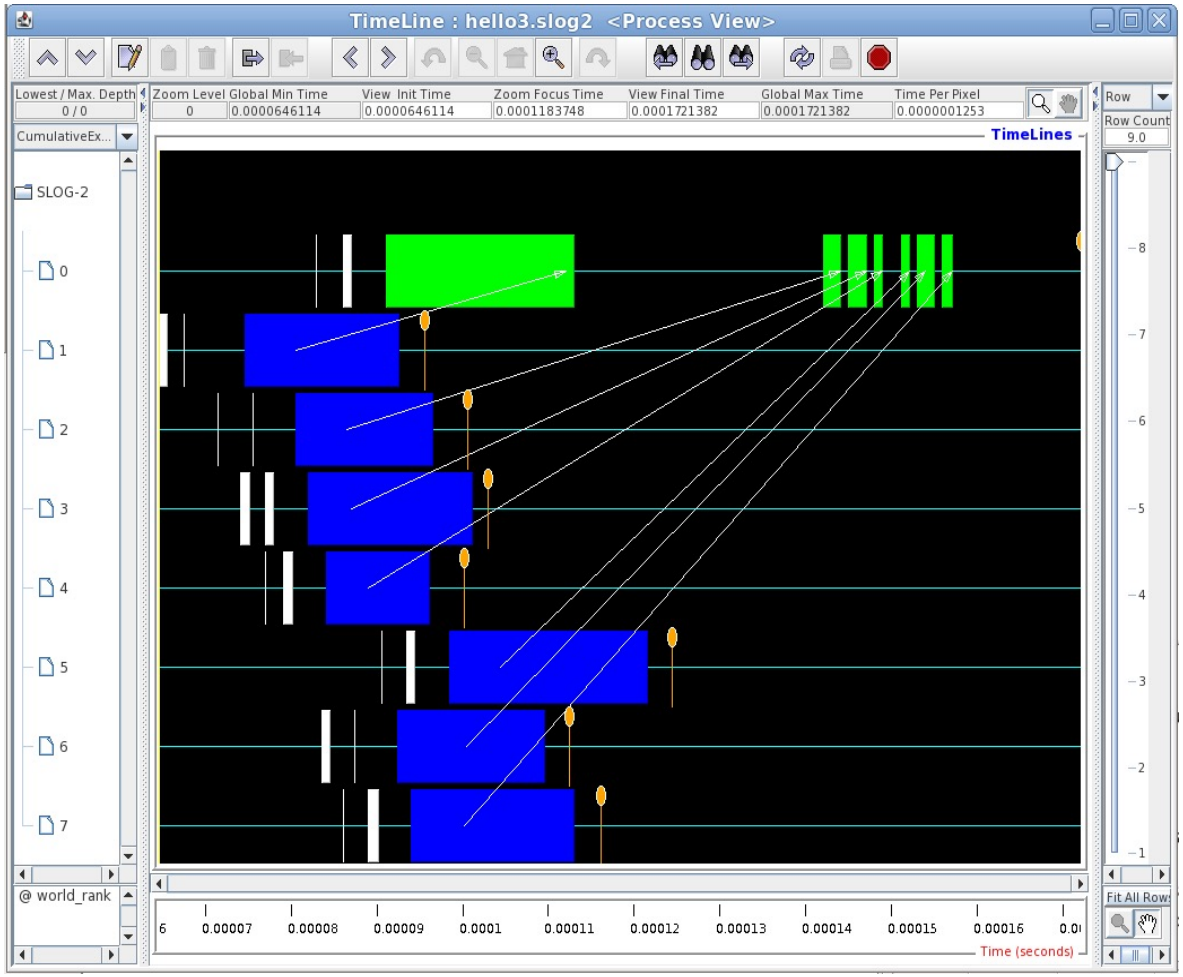
```
/* hello world, MPI style */

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

int main(int argc, char* argv[])
{
 int my_rank;  /* rank of process     */
 int p;        /* number of processes */
 int source;   /* rank of sender      */
 int dest;     /* rank of receiver    */

 int tag=0;    /* tag for messages    */
 char message[100];          /* storage for message */
 MPI_Status status;          /* return status for   */
                             /* receive             */

 /* Start up MPI */
 MPI_Init(&argc, &argv);

 /* Find out process rank */
 MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

 /* Find out number of processes */
 MPI_Comm_size(MPI_COMM_WORLD, &p);
```

```
 if (my_rank != 0) {
   /* Create message */
   sprintf(message, "Greetings from process %d!", my_rank);
   dest = 0;
   /* Use strlen+1 so that \0 gets transmitted */
   MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag,
MPI_COMM_WORLD);
 }
 else { /* my rank == 0 */
   for (source = 1; source < p; source++ ) {
     MPI_Recv(message, 100, MPI_CHAR, source, tag,
MPI_COMM_WORLD, &status);
     printf("%s\n", message);
   }
   printf("Greetings from process %d!\n", my_rank);
 }

 /* Shut down MPI */
 MPI_Finalize();

} /* end main */
```

Src : Prof. Amy Apon

# Communication map for the example.

```
mpiexec -n 8 ./hello3
Greetings from process 1!
Greetings from process 2!
Greetings from process 3!
Greetings from process 4!
Greetings from process 5!
Greetings from process 6!
Greetings from process 7!
Greetings from process 0!
Writing logfile....
Finished writing logfile.
[cdekate@celeritas l7]$
```



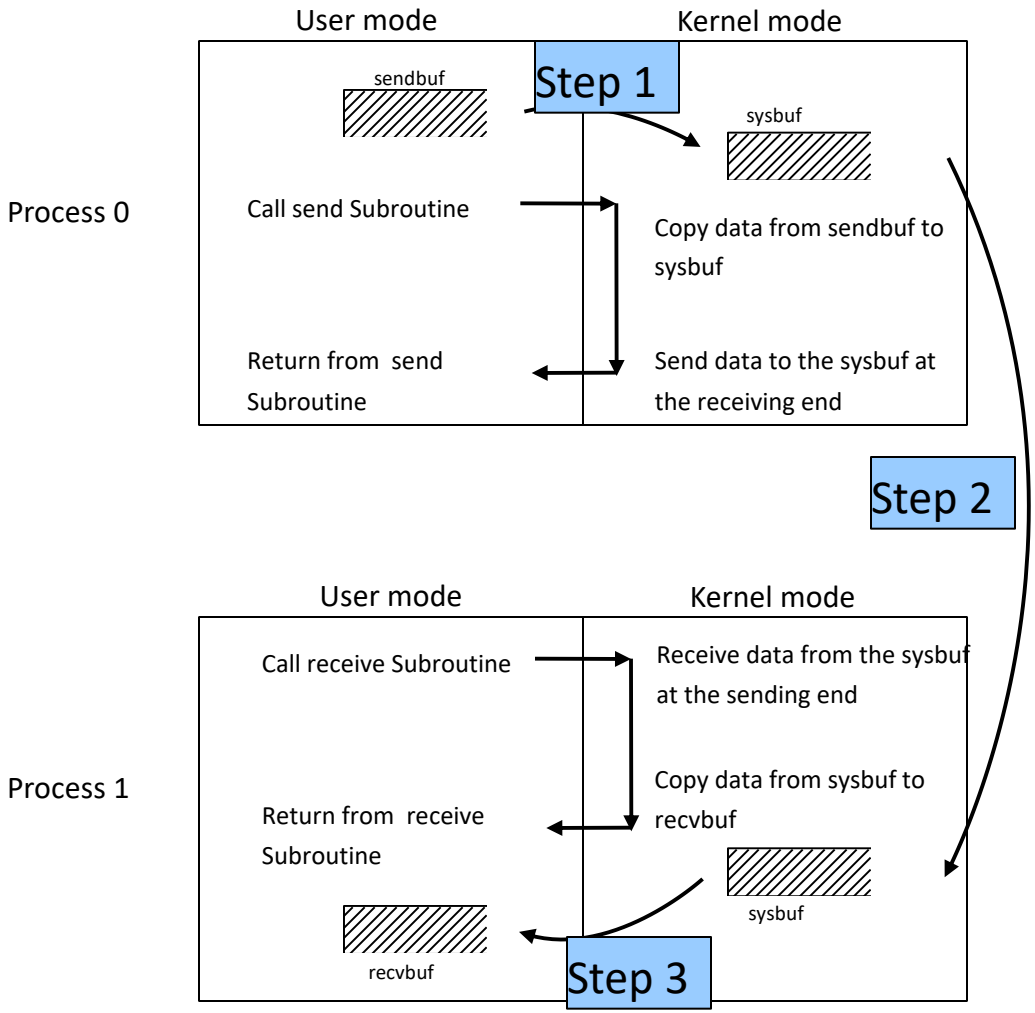
# Topics

- Introduction
- MPI Standard
- MPI-1 Model and Basic Calls
- MPI Communicators
- Point to Point Communication
- Point to Point Communication in-depth
- Deadlock
- Trapezoidal Rule : A Case Study
- Using MPI & Jumpshot to profile parallel applications
- Summary

# Point-to-point Communication

- How two processes interact
- Most flexible communication in MPI
- Two basic varieties
  - Blocking and non-blocking
- Two basic functions
  - Send and receive
- With these two functions, and the four functions we already know, you can do everything in MPI
  - But there's probably a better way to do a lot things, using other functions

# Point to Point Communication : Basic concepts (buffered)

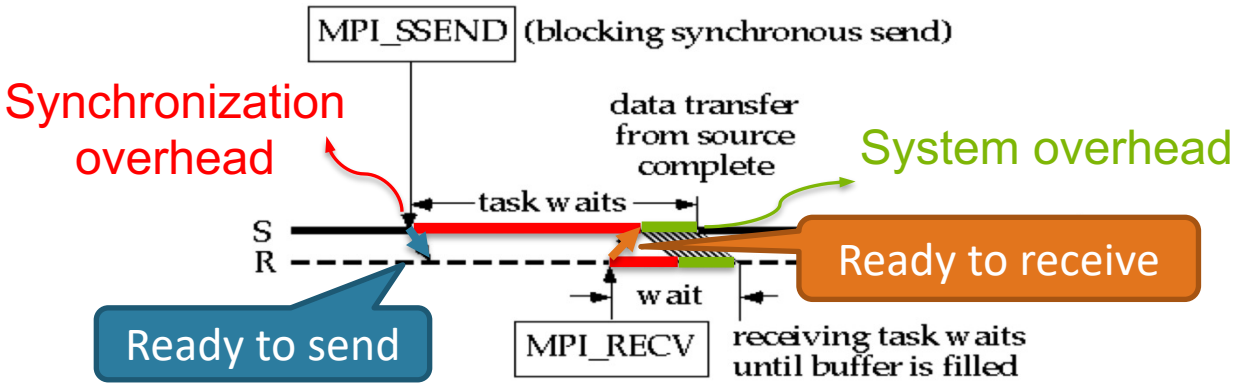



1. Data to be sent by the user is copied from the user memory space to the system buffer
2. The data is sent from the system buffer over the network to the system buffer of receiving process
3. The receiving process copies the data from system buffer to local user memory space

# MPI communication modes

- MPI offers several different types of communication modes, each having implications on data handling and performance:
  - Buffered
  - Ready
  - Standard
  - Synchronous
- Each of these communication modes has both blocking and non-blocking primitives
  - In blocking point to point communication the send call blocks until the send block can be reclaimed. Similarly the receive function blocks until the buffer has successfully obtained the contents of the message.
  - In the non-blocking point to point communication the send and receive calls allow the possible overlap of communication with computation. Communication is usually done in 2 phases: the posting phase and the test for completion phase.
- **Synchronization Overhead**: the time spent waiting for an event to occur on another task.
- **System Overhead**: the time spent when copying the message data from sender's message buffer to network and from network to the receiver's message buffer.

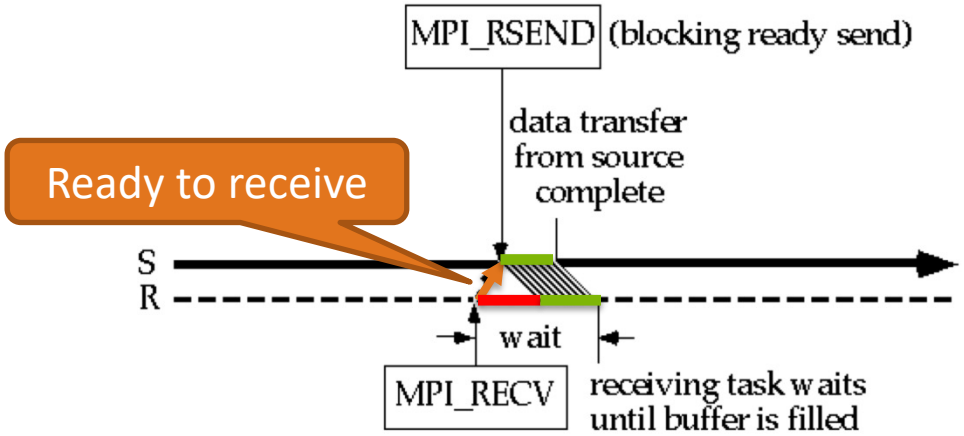
# Point to Point Communication
# Blocking Synchronous Send




- The communication mode is selected while invoking the **send** routine.
- When ***blocking synchronous send*** is executed (***MPI_Ssend()***) , "*ready to send*" message is sent from the sending task to receiving task.
- When the ***receive call*** is executed (***MPI_Recv()***), "*ready to receive*" message is sent, followed by the transfer of data.
- The sender process must wait for the receive to be executed and for the handshake to arrive before the message can be transferred. (**Synchronization Overhead**)
- The receiver process also has to wait for the handshake process to complete. (**Synchronization Overhead**)
- Overhead incurred while copying from sender & receiver buffers to the network.

http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3_pt2pt2.html

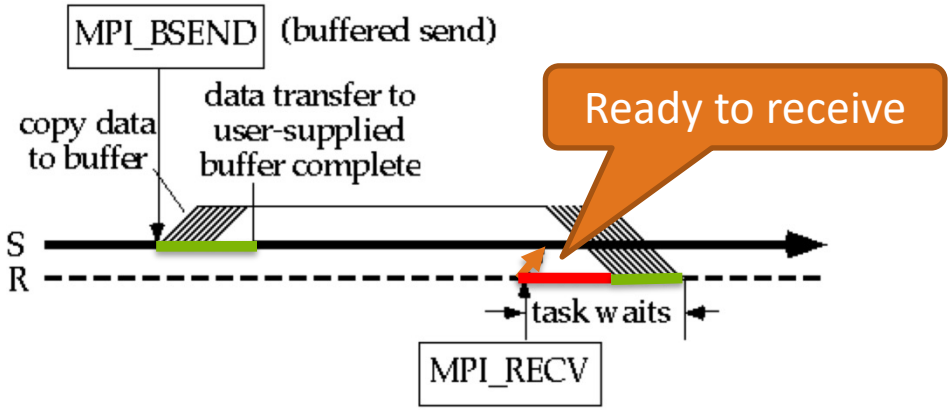
# Point to Point Communication
# Blocking Ready Send




- The **ready mode send call** (**MPI_Rsend**) sends the message over the network once the "ready to receive" message is received.
- If "ready to receive" message hasn't arrived, the ready mode send will incur an error and exit. The programmer is responsible to provide for handling errors and overriding the default behavior.
- The ready mode send call minimizes system overhead and synchronization overhead incurred during sending of the task.
- The receive still incurs substantial synchronization overhead depending on how much earlier the receive call is executed.

http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3_pt2pt2.html

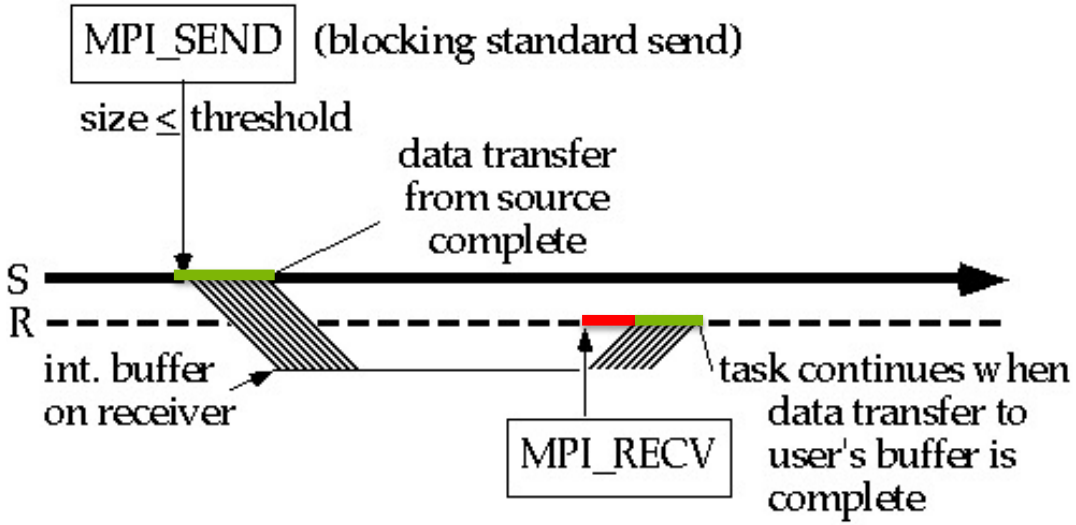
# Point to Point Communication
# Blocking Buffered Send




- The blocking buffered send call (MPI_Bsend()) copies the data from the message buffer to a user-supplied buffer and then returns.
- The message buffer can then be reclaimed by the sending process without having any effect on any data that is sent.
- When the "ready to receive" notification is received the data from the user-supplied buffer is sent to the receiver.
- Replicated copies of the buffer results in added system overhead.
- Synchronization overhead on the sender process is eliminated as the sending process does not have to wait on the receive call.
- Synchronization overhead on the receiving process can still be incurred, because if the receive is executed before the send, the process must wait before it can return to the execution sequence

http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3_pt2pt2.html

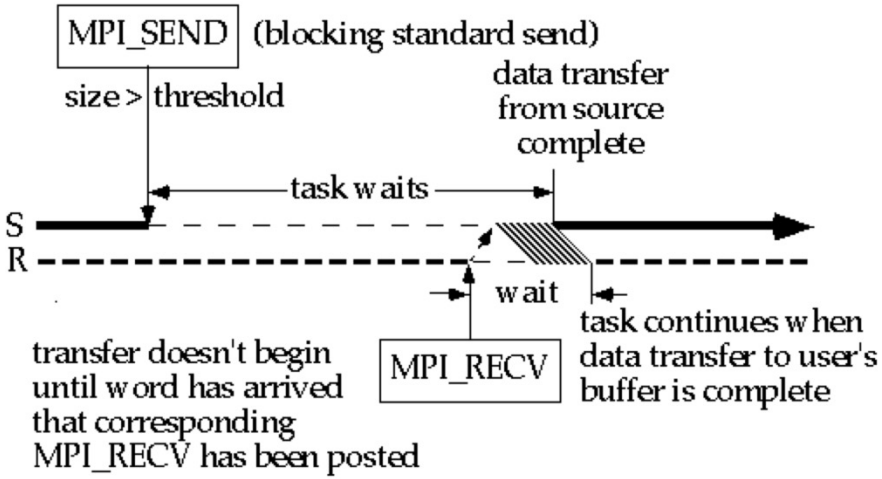
# Point to Point Communication
# Blocking Standard Send (small data size)




- The MPI_Send() operation is implementation dependent
- When the data size is smaller than a threshold value (*varies for each implementation)*:
  - The blocking standard send call (MPI_Send()) copies the message over the network into the system buffer of the receiving node, after which the sending process continues with the computation
  - When the receive call (MPI_Recv()) is executed the message is copied from the system buffer to the receiving task
  - The decreased synchronization overhead is usually at the cost of increased system overhead due to the extra copy of buffers

http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3_pt2pt2.html

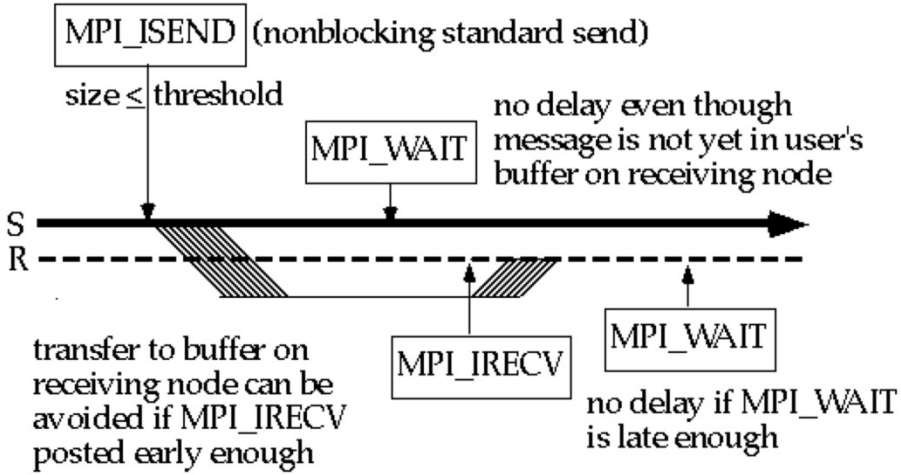
# Point to Point Communication
# Blocking Standard Send (large data size)




- When the message size is greater than a threshold
  - The behavior is same as for the synchronous mode
  - Small messages benefit from the decreased chance of synchronization overhead
  - Large messages results in increased cost of copying to the buffer and system overhead

http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3_pt2pt2.html

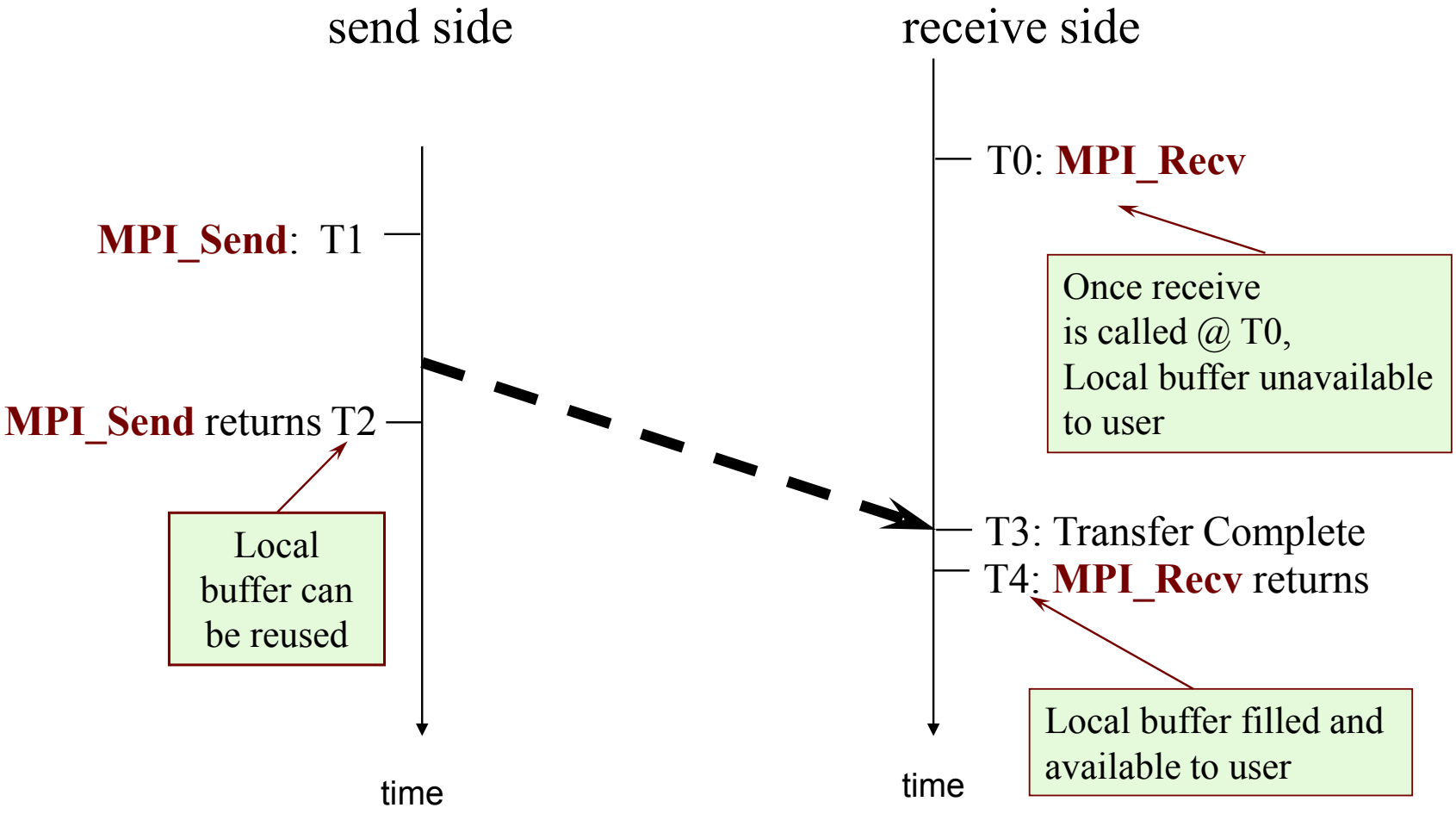
# Point to Point Communication
# Non-blocking Calls




- The non-blocking send call (MPI_Isend()) posts a non-blocking standard send when the message buffer contents are ready to be transmitted
- The control returns immediately without waiting for the copy to the remote system buffer to complete. MPI_Wait is called just before the sending task needs to overwrite the message buffer
- Programmer is responsible for checking the status of the message to know whether data to be sent has been copied out of the send buffer
- The receiving call (MPI_Irecv()) issues a non-blocking receive as soon as a message buffer is ready to hold the message. The non-blocking receive returns without waiting for the message to arrive. The receiving task calls MPI_Wait when it needs to use the incoming message data

http://ib.cnea.gov.ar/~ipc/ptpde/mpi-class/3_pt2pt2.html

# Blocking Send-Receive Timing Diagram (Receive before Send)



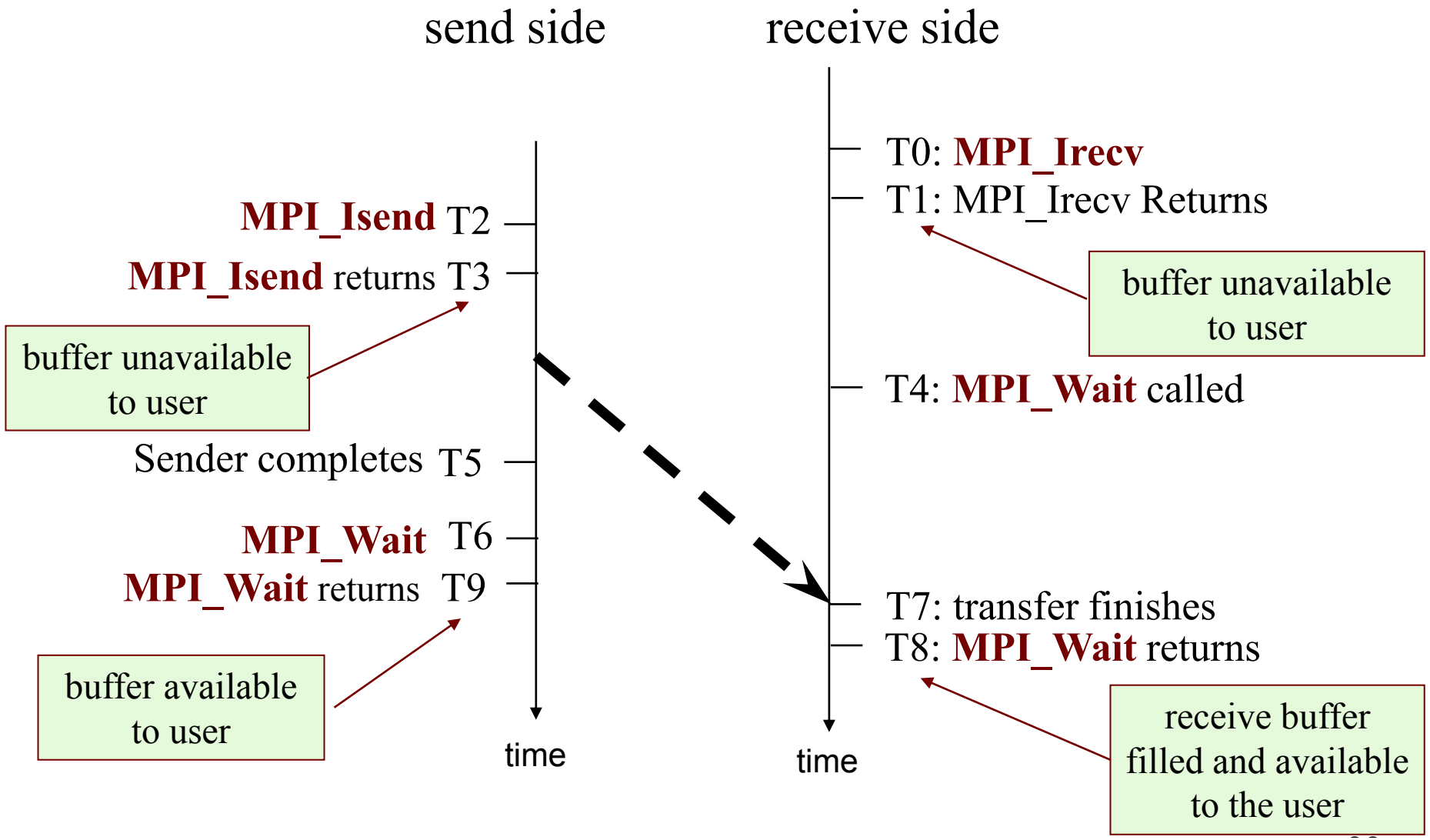

It is important to post the receive before sending, for highest performance

# Non-Blocking Communication

- Non-blocking operations return immediately and pass "request handles" that can be waited on and queried
  - MPI_ISEND( start, count, datatype, dest, tag, comm, request )
  - MPI_IRECV( start, count, datatype, src, tag, comm, request )
  - MPI_WAIT( request, status )
- One can also test without waiting using MPI_TEST
  - MPI_TEST( request, flag, status )
- Anywhere you use MPI_Send or MPI_Recv, you can use the pair of MPI_Isend/MPI_Wait or MPI_Irecv/MPI_Wait

Non-blocking operations are extremely important ... they allow you to overlap computation and communication

# Non-Blocking Send-Receive Diagram



# Point to Point Communication
# Non-blocking Calls

- When the system buffer is full, the blocking send would have to wait until the receiving task pulled some message data out of the buffer. Use of non-blocking call allows computation to be done during this interval, allowing for interleaving of computation and communication
- **Non-blocking calls ensure that deadlock will not result**

# Topics

- Introduction
- MPI Standard
- MPI-1 Model and Basic Calls
- MPI Communicators
- Point to Point Communication
- Point to Point Communication in-depth
- Deadlock
- Trapezoidal Rule : A Case Study
- Using MPI & Jumpshot to profile parallel applications
- Summary

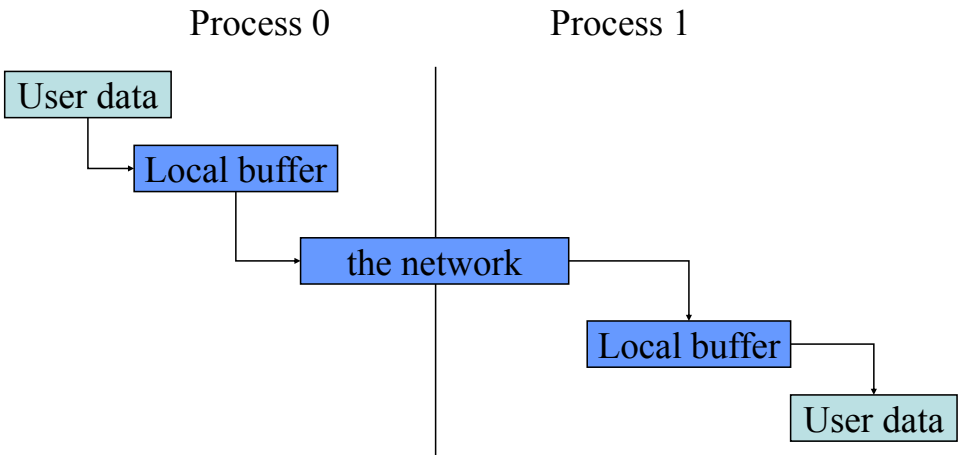
# Deadlock

- Something to avoid
- A situation where the dependencies between processors are cyclic
  - One processor is waiting for a message from another processor, but that processor is waiting for a message from the first, so nothing happens
    - Until your time in the queue runs out and your job is killed
    - MPI does not have timeouts

# Deadlock Example

```
If (rank == 0) {
  err = MPI_Send(sendbuf, count, datatype, 1, tag, comm);
  err = MPI_Recv(recvbuf, count, datatype, 1, tag, comm, &status);
}else {
  err = MPI_Send(sendbuf, count, datatype, 0, tag, comm);
  err = MPI_Recv(recvbuf, count, datatype, 0, tag, comm, &status);
}
```

- If the message sizes are small enough, this should work because of systems buffers
- If the messages are too large, or system buffering is not used, this will hang



# Deadlock Example Solutions

```
If (rank == 0) {
  err = MPI_Send(sendbuf, count, datatype, 1, tag, comm);
  err = MPI_Recv(recvbuf, count, datatype, 1, tag, comm, &status);
}else {
  err = MPI_Recv(recvbuf, count, datatype, 0, tag, comm, &status);
  err = MPI_Send(sendbuf, count, datatype, 0, tag, comm);
}
```

or

```
If (rank == 0) {
  err = MPI_Isend(sendbuf, count, datatype, 1, tag, comm, &req);
  err = MPI_Irecv(recvbuf, count, datatype, 1, tag, comm);
  err = MPI_Wait(req, &status);
}else {
  err = MPI_Isend(sendbuf, count, datatype, 0, tag, comm, &req);
  err = MPI_Irecv(recvbuf, count, datatype, 0, tag, comm);
  err = MPI_Wait(req, &status);
}
```

# Topics

- Introduction
- MPI Standard
- MPI-1 Model and Basic Calls
- MPI Communicators
- Point to Point Communication
- Point to Point Communication in-depth
- Deadlock
- Trapezoidal Rule : A Case Study
- Using MPI & Jumpshot to profile parallel applications
- Summary

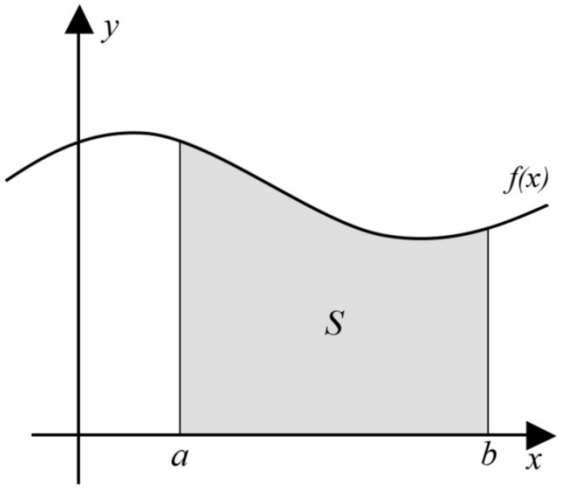
# Numerical Integration Using Trapezoidal Rule: A Case Study

- In review, the 6 main MPI calls:
  - MPI_Init
  - MPI_Finalize
  - MPI_Comm_size
  - MPI_Comm_rank
  - MPI_Send
  - MPI_Recv
- Using these 6 MPI function calls we can begin to construct several kinds of parallel applications
- In the following section we discuss how to use these 6 calls to parallelize Trapezoidal Rule

# Approximating Integrals: Definite Integral

- Problem: to find an approximate value to a definite integral $\int_a^b f(x)\,dx.$

- A definite integral from a to b of a non negative function f(x) can be thought of as the area bound by the X-axis, the vertical lines x=a and x=b, and the graph of f(x)

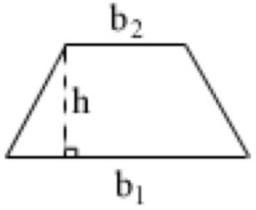


# Approximating Integrals: Trapezoidal Rule

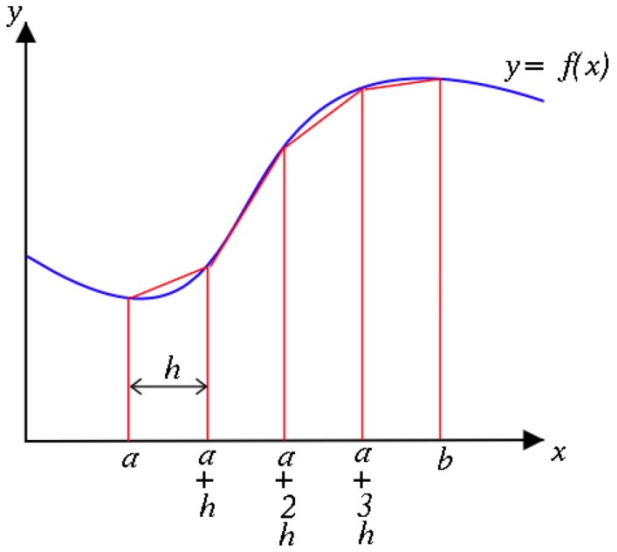
- Approximating area under the curve can be done by dividing the region under the curve into regular geometric shapes and then adding the areas of the shapes.
- In Trapezoidal Rule, the region between *a* and *b* can be divided into *n* trapezoids of base
  *h = (b-a)/n*
- The area of a trapezoid can be calculated as $\frac{h(b_1 + b_2)}{2}$
- In the case of our function the area for the first block can be represented as

$$\frac{h(f(a) + f(a+h))}{2}$$

- The area under the curve bounded by a & b can be approximated as :




$$\frac{h(b_1 + b_2)}{2}$$




$$\left[\frac{h(f(a) + f(a+h))}{2}\right] + \left[\frac{h(f(a+h) + f(a+2h))}{2}\right] + \left[\frac{h(f(a+2h) + f(a+3h))}{2}\right] + \left[\frac{h(f(a+3h) + f(b))}{2}\right]$$

# Approximating Integrals: Trapezoidal Rule

- We can further generalize this concept of approximation of integrals as a summation of trapezoidal areas

$$\frac{1}{2}h[f(x_0)+f(x_1)]+\frac{1}{2}h[f(x_1)+f(x_2)]+\ldots+\frac{1}{2}h[f(x_{n-1})+f(x_n)]$$

$$=\frac{h}{2}[f(x_0)+f(x_1)+f(x_1)+f(x_2)+\ldots+f(x_{n-1})+f(x_n)]$$

$$=\frac{h}{2}[f(x_0)+2f(x_1)+2f(x_2)+\ldots+2f(x_{n-1})+f(x_n)]$$

$$=h\left[\frac{f(x_0)}{2}+f(x_1)+f(x_2)+\ldots+f(x_{n-1})+\frac{f(x_n)}{2}\right]$$

# Trapezoidal Rule – Serial / Sequential program in C

```
/* serial.c -- serial trapezoidal rule
 *
 * Calculate definite integral using trapezoidal rule.
 * The function f(x) is hardwired.
 * Input: a, b, n.
 * Output: estimate of integral from a to b of f(x)
 *    using n trapezoids.
 *
 * See Chapter 4, pp. 53 & ff. in PPMPI.
 */

#include <stdio.h>

main() {
   float  integral;   /* Store result in integral   */
   float  a, b;       /* Left and right endpoints   */
   int    n;          /* Number of trapezoids       */
   float  h;          /* Trapezoid base width       */
   float  x;
   int    i;
   float f(float x);  /* Function we're integrating */

   printf("Enter a, b, and n\n");
   scanf("%f %f %d", &a, &b, &n);
   h = (b-a)/n;
   integral = (f(a) + f(b))/2.0;
   x = a;
   for (i = 1; i <= n-1; i++) {
      x = x + h;
      integral = integral + f(x);
   }
   integral = integral*h;

   printf("With n = %d trapezoids, our estimate\n",
      n);
   printf("of the integral from %f to %f = %f\n",
      a, b, integral);
} /* main */


float f(float x) {
   float return_val;
   /* Calculate f(x).  Store calculation in return_val. */

   return_val = x*x;
   return return_val;
} /* f */
```

Trapezoidal Example Adapted from Parallel Programming in MPI P.Pacheco Ch 4

# Results for the Serial Trapezoidal Rule

| a | b | n | f(x) single precision | f(x) double precision |
|---|---|---|---|---|
| 2 | 25 | 1 | 7233.500000 | 7233.500000 |
| 2 | 25 | 2 | 5712.625000 | 5712.625000 |
| 2 | 25 | 10 | 5225.945312 | 5225.945000 |
| 2 | 25 | 30 | 5207.916992 | 5207.919815 |
| 2 | 25 | 40 | 5206.934082 | 5206.934062 |
| 2 | 25 | 50 | 5206.475098 | 5206.477800 |
| 2 | 25 | 1000 | 5205.664551 | 5205.668694 |

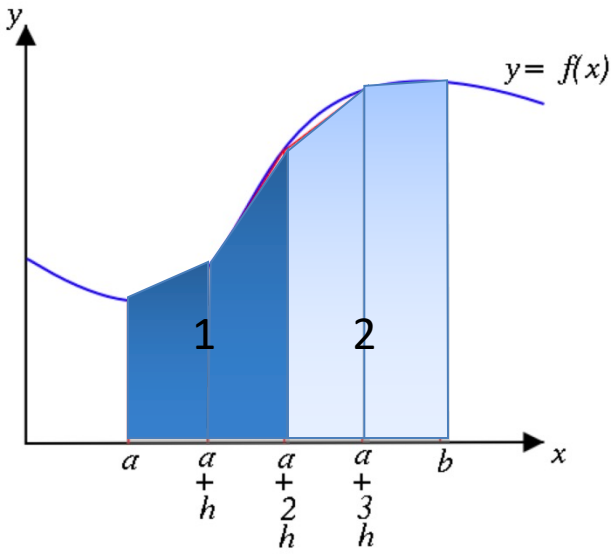
```
[LSU760000@n00 l7]$ ./serial
Enter a, b, and n
2 25 5000
With n = 5000 trapezoids, our estimate
of the integral from 2.000000 to 25.000000 = 5205.550293
```

Trapezoidal Example Adapted from Parallel Programming in MPI P.Pacheco Ch 4

# Parallelizing Trapezoidal Rule

- One way of parallelizing Trapezoidal rule :
  - Distribute chunks of workload (each workload characterized by its own subinterval of ***[a,b]*** to each process)
  - Calculate ***f*** for each subinterval
  - Finally add the ***f*** calculated for all the sub intervals to produce result for the complete problem ***[A,B]***
- Issues to consider
  - Number of trapezoids (*n*) are equally divisible across (*p*) processes (load balancing).
  - First process calculates the area for the first *n/p* trapezoids, second process calculates the area for the next *n/p* trapezoids and so on
- Key information related to the problem that each process needs is the
  - Rank of the process
  - Ability to derive the workload per processor as a function of rank

Assumption : Process 0 does the summation



# Parallelizing Trapezoidal Rule

- Algorithm
  Assumption: Number of trapezoids *n* is evenly divisible across *p* processors
  - Calculate: $$h = \frac{(b-a)}{n}$$
  - Each process calculates its own workload (interval to integrate)
    - local number of trapezoids ( local_n) = ***n/p***
    - local starting point (local_a) = a+(process_rank *local_n* h)
    - local ending point (local_b) = (local_a + local_n * h)
  - Each process calculates its own integral for the local intervals
    - For each of the local_n trapezoids calculate area
    - Aggregate area for local_n trapezoids
  - If PROCESS_RANK == 0
    - Receive messages (containing sub-interval area aggregates) from all processors
    - Aggregate (ADD) all sub-interval areas
  - If PROCESS_RANK > 0
    - Send sub-interval area to PROCESS_RANK(0)

Classic SPMD: all processes run the same program on different datasets.

# Parallel Trapezoidal Rule

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

float Trap(float local_a, float local_b, int local_n,
           float h);    /* Calculate local integral  */

main(int argc, char** argv) {
   int       my_rank;    /* My process rank         */
   int       p;          /* The number of processes   */
   float      a = 0.0;   /* Left endpoint           */
   float      b = 1.0;   /* Right endpoint          */
   int       n = 1024;   /* Number of trapezoids     */
   float      h;         /* Trapezoid base length    */
   float      local_a;   /* Left endpoint my process  */
   float      local_b;   /* Right endpoint my process */
   int       local_n;   /* Number of trapezoids for my calculation          */
   float      integral;  /* Integral over my interval */
   float      total;     /* Total integral          */
   int       source;    /* Process sending integral  */
   int       dest = 0;  /* All messages go to 0     */
   int       tag = 0;
   MPI_Status  status;
```

Trapezoidal Example Adapted from Parallel Programming in MPI P.Pacheco Ch 4

# Parallel Trapezoidal Rule

```
/* Let the system do what it needs to start up MPI */
MPI_Init(&argc, &argv);

/* Get my process rank */
MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

/* Find out how many processes are being used */
MPI_Comm_size(MPI_COMM_WORLD, &p);

h = (b-a)/n;    /* h is the same for all processes */
local_n = n/p;  /* So is the number of trapezoids */

/* Length of each process' interval of
 * integration = local_n*h.  So my interval
 * starts at: */
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
integral = Trap(local_a, local_b, local_n, h);
```

Trapezoidal Example Adapted from Parallel Programming in MPI P.Pacheco Ch 4

# Parallel Trapezoidal Rule

```
/* Add up the integrals calculated by each process */
  if (my_rank == 0) {
     total = integral;
     for (source = 1; source < p; source++) {
        MPI_Recv(&integral, 1, MPI_FLOAT, source, tag,
           MPI_COMM_WORLD, &status);
        total = total + integral;
     }
  } else {
     MPI_Send(&integral, 1, MPI_FLOAT, dest,
        tag, MPI_COMM_WORLD);
  }
  /* Print the result */
  if (my_rank == 0) {
     printf("With n = %d trapezoids, our estimate\n",
        n);
     printf("of the integral from %f to %f = %f\n",
        a, b, total);
  }
  /* Shut down MPI */
  MPI_Finalize();
} /*  main  */
```

Trapezoidal Example Adapted from Parallel Programming in MPI P.Pacheco Ch 4

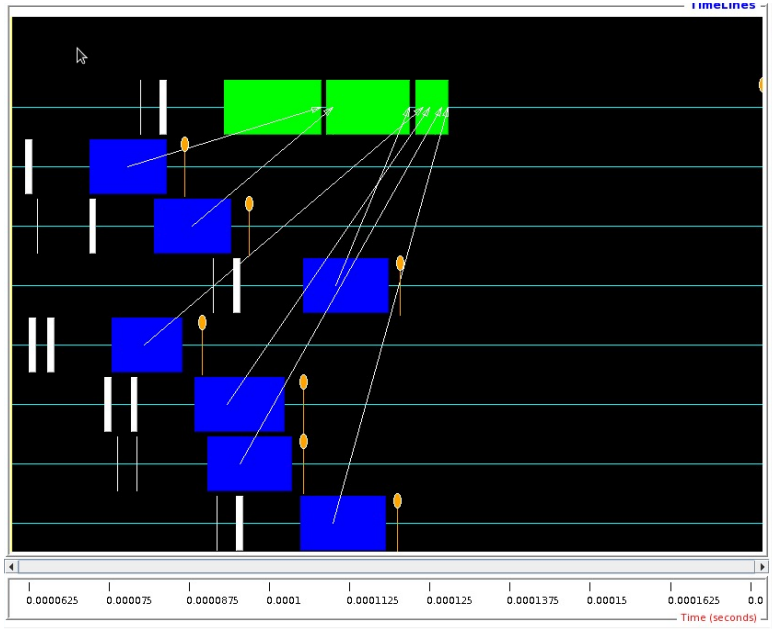
# Parallel Trapezoidal Rule

```
float Trap(
        float  local_a   /* in */,
        float  local_b   /* in */,
        int    local_n   /* in */,
        float  h         /* in */) {

   float integral;   /* Store result in integral  */
   float x;
   int i;

   float f(float x); /* function we're integrating */

   integral = (f(local_a) + f(local_b))/2.0;
   x = local_a;
   for (i = 1; i <= local_n-1; i++) {
      x = x + h;
      integral = integral + f(x);
   }
   integral = integral*h;
   return integral;
} /*  Trap  */
float f(float x) {
   float return_val;
   /* Calculate f(x). */
   /* Store calculation in return_val. */
   return_val = x*x;
   return return_val;
} /* f */
```

Trapezoidal Example Adapted from Parallel Programming in MPI P.Pacheco Ch 4

# Parallel Trapezoidal Rule

[cdekate@celeritas l7]$ mpiexec -n 8 … trap
With n = 1024 trapezoids, our estimate
of the integral from 2.000000 to 25.000000 = 5205.667969
Writing logfile....
Finished writing logfile.



# Topics

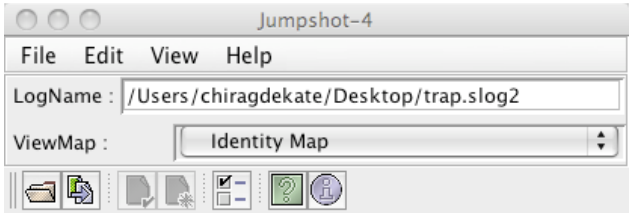
- Introduction
- MPI Standard
- MPI-1 Model and Basic Calls
- MPI Communicators
- Point to Point Communication
- Point to Point Communication in-depth
- Deadlock
- Trapezoidal Rule : A Case Study
- Using MPI & Jumpshot to profile parallel applications
- Summary

# Profiling Applications

- To Profile your parallel applications:
  1. Compile the applications with
     ***mpicc -profile=mpe_mpilog -o trap trap.c*** (mpich)
  2. Run your applications using the standard procedure using PBS/mpirun
  3. After your run is complete you might see lines like these in your stdout (standardout / output file of your pbs-based run
     ***Writing logfile....***
     ***Finished writing logfile.***
  4. You will also see a file with an extension "*clog2*"
  5. I.e. if your executable was named "parallel_program" you would see a file named "parallel_program.clog2"
  6. Convert the "clog2" to "slog2" format by issuing the command
     ***"clog2TOslog2 parallel_program.clog2"***
     Maintain the capitalization in the clog2TOslog2 command
  7. Step 6 will result in a ***parallel_program.slog2*** file
  8. Use Jumpshot to visualize this file

# Using Jumpshot

- Note : You need Java Runtime Environment on your machine in order to be able to run Jumpshot
- Download your parallel_program.slog2 file from Arete
- Download Jumpshot from :
  - ftp://ftp.mcs.anl.gov/pub/mpi/slog2/slog2rte.tar.gz
  - Uncompress the tar.gz file to get a folder : slog2rte-1.2.6/
  - In the ***slog2rte-1.2.6/lib/***
    type ***java -jar jumpshot.jar parallel_program.slog2***
- Or click on the jumpshot_launcher.jar
- Open the file using the jumpshot file menu




Or simply use the following commands in the new version of MPE
***mpecc -mpilog -o trap trap.c***
***jumpshot trap.clog2***

# Summary

- MPI is an API.  It can be used directly from C, C++, and Fortran applications
  - 3rd parties have created bindings in other languages.
- MPI presents an interface that is deals with sending and receiving typed messages.
  - MPI does not expose a streaming interface; messages are sent and received as atomic units.
- MPI sends and receives ordered, typed, point-to-point messages (i.e., one sender and one receiver) and "collective" message passing (i.e., multiple participants in the communication).
- MPI abstracts away the underlying network
  - e.g., TCP sockets, shared memory, an OpenFabrics-based network, …etc.