ICRAR

International Centre for Radio Astronomy Research

MPI

Research Associate Professor
Kevin Vinsen

Curtin University

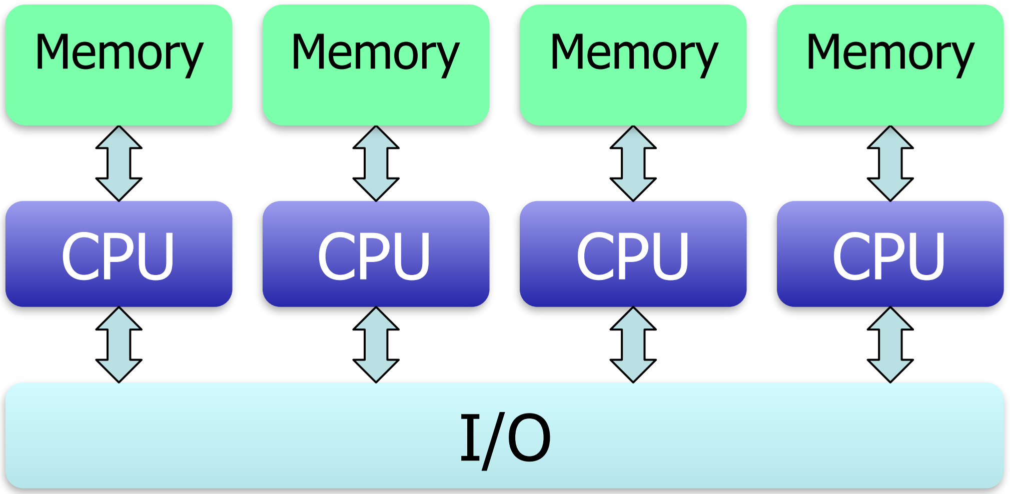
THE UNIVERSITY OF WESTERN AUSTRALIA

Government of Western Australia
Department of the Premier and Cabinet
Office of Science

Distributed Memory HPC

Communicate via messages

Message Passing Interface (MPI)



MPI - Message Passing Interface

- **MPI is a standard, not a library!**
  - It defines the functions, constants, and behaviours for a set of commands to exchange information between computers.
- **Many libraries implement the MPI standard. ie:**
  - mpich2
  - lam-mpi
  - OpenMPI
  - HP MPI
  - Intel MPI

MPI binding

MPI is not a programming language

- **MPI libraries support**
  - C
  - C++
  - Fortran
  - Python
  - Java
  - many others!

What MPI can Not

MPI cannot magically transform your program into an efficient parallel program

1. The algorithm must be suitable for parallelization
2. Communication between processors takes time
3. Remember the model

When Not to Parallelise Code

- **Code will only be used once (or infrequently)**
  Efficient parallel code takes time to develop!
- **Current performance is acceptable and execution time is short**
- **Frequent & significant code changes**
- **Some algorithms simply do not parallelize**

When To Parallelise Code

Code is physically incapable of running on one computer

- memory requirements are too great
- run time would be months

Code will be reused frequently

- Parallelization is an investment

Data structures are simple, calculations are local

- Easy to communicate and synchronize between processors

MPI Programming Model

MPI employs MIMD in SPMD framework.

1. The user issues a directive to the operating system that has the effect of placing a copy of the executable program on each processor.
2. Each processor begins execution of its copy of the executable.
3. Different processes can execute different statements by branching within the program based on their process identity (rank)

```
if (my_rank != 0)
        .
        .
        .
else
        .
        .
        .
```

MPI Basics

1. **Starting and Finishing**

MPI_Init	initialise MPI
MPI_Finalize	terminate computation

2. **Identifying yourself**

MPI_Comm_size	number of processes
MPI_Comm_rank	my process identifier

3. **Sending and Receiving messages**

MPI_Send	send a message
MPI_Recv	receive a message

MPI_Init

C: *int MPI_Init(int *argc, /* in/out */*

```
            char **argv) /* in/out */
```

Fortran: call MPI_Init(ierr)

Initialises MPI subroutines

- Connects to other processes
- Processes any MPI specific command line arguments
- Must be called before any other MPI function is used
- Must be called before your program examines any command lines arguments

MPI_Finalize

C: MPI_Finalize(void)

Fortran: call MPI_Finalize(ierr)

Cleans up MPI sessions

- Must be called by all processes before exiting
- After being called, MPI functions cannot be used
- If forgotten, program will sometimes (but not always) crash at the end with strange errors

MPI Communicator

Communicator

- Collection of processes
- Determines scope to which messages are related
- Identifier of process (rank) is relative to communicator
- Defines the scope of communications (broadcast, etc.)

MPI_Comm_size

C: *MPI_Comm_size(MPI_Comm comm, /* in */*
*int *size) /* out */*

```
MPI_Comm_size(MPI_Comm comm,   /* in */
              int *size)       /* out */
```

Fortran: **call MPI_Comm_size(***MPI_Comm***, integer, integer ierr)**

Retrieves the number of processes in the communicator

The MPI_Comm comm parameter specifies the “communicator”, or communication group, to perform the function on
In general, use MPI_COMM_WORLD, which means all processes

MPI_Comm_rank

C: *MPI_Comm_rank(MPI_Comm comm, /* in */*
*int *rank) /* out */*

Fortran: call MPI_Comm_rank(MPI_Comm, integer, integer ierr)

Retrieves the rank of the process

- Rank is 0-based. (ie. the first process has rank 0, the last has a rank of nProcesses-1)
- Use this value to decide which work this process should do

MPI Messages

Message content, a sequence of bytes

Message needs wrapper (envelope for a letter)

- Destination
- Source
- Message type
- Size (count)
- Communicator
- Broadcast

Point to Point Blocking

MPI_Send	send a message
MPI_Recv	receive a message

MPI message protocol

Communicator (sixth parameter in MPI_Send and MPI_Recv) determines the context for destination and source ranks

MPI_COMM_WORLD is automatically supplied communicator, which includes all processes created at start-up

Other communicators can be defined by user to group processes and to create virtual topologies

MPI_Send

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

Fortran: INTEREG err

buf is a pointer to the data you want to send

count is the number of data elements to send

datatype is the type of the data

MPI_INT, MPI_FLOAT, MPI_DOUBLE, etc

dest is the *rank* of the process that should receive data

tag is a unique message identifier

comm is generally MPI_COMM_WORLD

MPI_Recv

```
MPI_Recv(void *buf,                /* out */
         int count,          /* in */
         MPI_Datatype datatype,       /* in */
         int src,            /* in */
         int tag,            /* in */
         MPI_Comm comm,               /* in */
         MPI_Status *status)          /* out */
```

Fortran: INTEREG err

buf is a pointer to the location the data should be stored

count is the number of data elements to receive

datatype is the type of the data

MPI_INT, MPI_FLOAT, MPI_DOUBLE, etc

src is the *rank* of the process that sends the data (or MPI_ANY_SOURCE)

tag is a unique message identifier (or MPI_ANY_TAG)

comm is generally MPI_COMM_WORLD

status contains information about the message received

MPI message protocol

Status of message received by MPI_Recv is returned in the *status* **parameter**

Number of items actually received can be determined from status by using function MPI_Get_count

```
int MPI_Get_count( MPI_Status *status,  MPI_Datatype datatype, int *count )
```

First MPI Program (in C)

```
#include <mpi.h>
#include <string.h>
#include <stdio.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 messages
        MPI_Status status;              // return status for receive
…
```

First MPI Program (cont.)

```
/* 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);
```

First MPI Program (cont.)

```
if(my_rank != 0) {                              // for slave processes
        sprintf(message, "Greetings from process %d!", my_rank);
        dest = 0;

          /* lets send message */
       MPI_Send(message, strlen(message)+1, MPI_CHAR, dest,
                    tag, MPI_COMM_WORLD);
    }
```

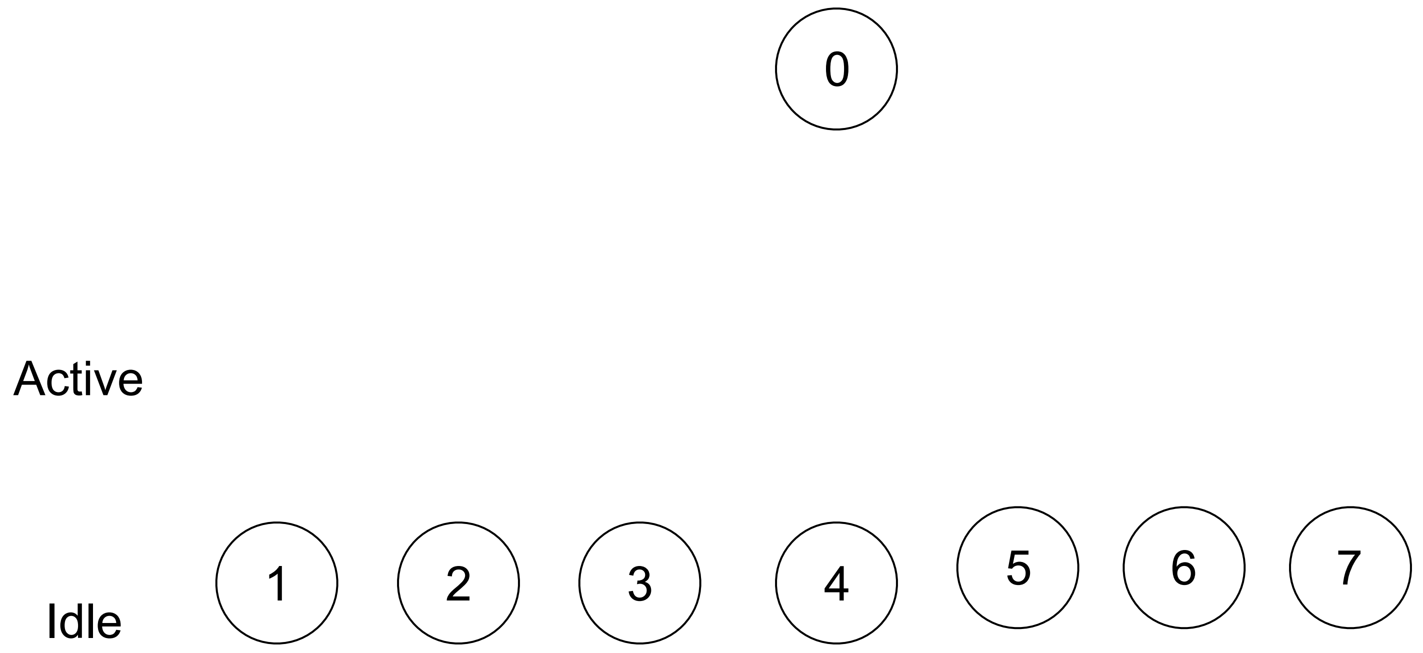
First MPI Program (cont.)

```
else {                                   // for master process
        printf("I am the Master! My rank is %d.\n ", my_rank);
        for(source = 1; source < p; source++) {
        /* lets receive message from source */
                MPI_Recv(message, 100, MPI_CHAR, source, tag,
    MPI_COMM_WORLD, &status);
                printf("%s\n", message);
        }
  }
```

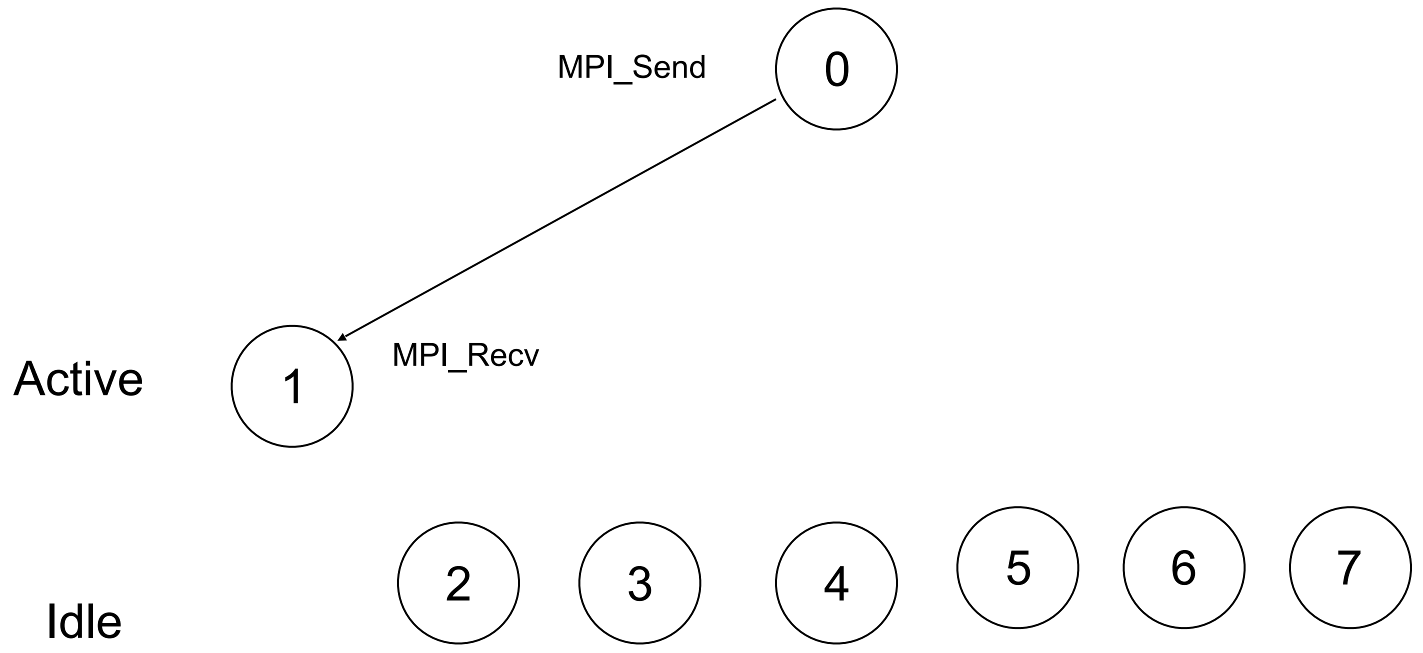
First MPI Program (cont.)

```
  /* Shut down MPI */
        MPI_Finalize();
}
```

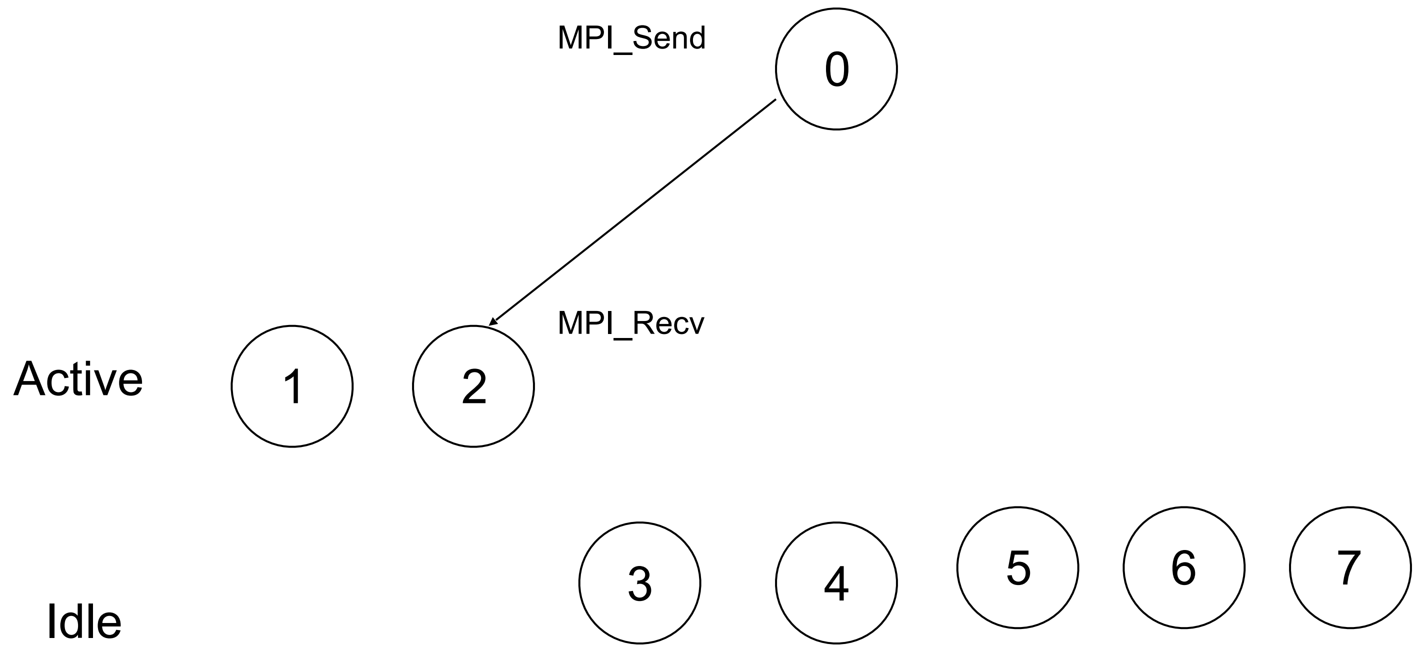
Blocking Type of Communications in Trapezoid Rule Program



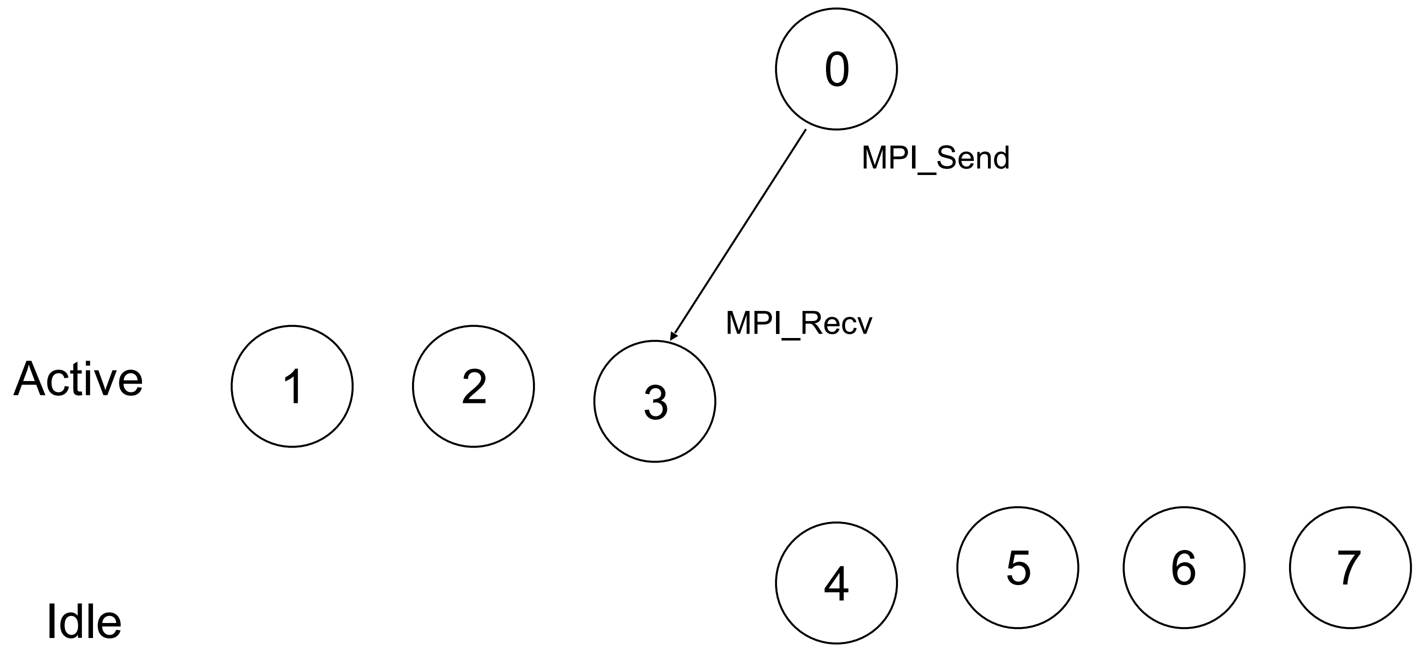
Blocking Type of Communications in Trapezoid Rule Program



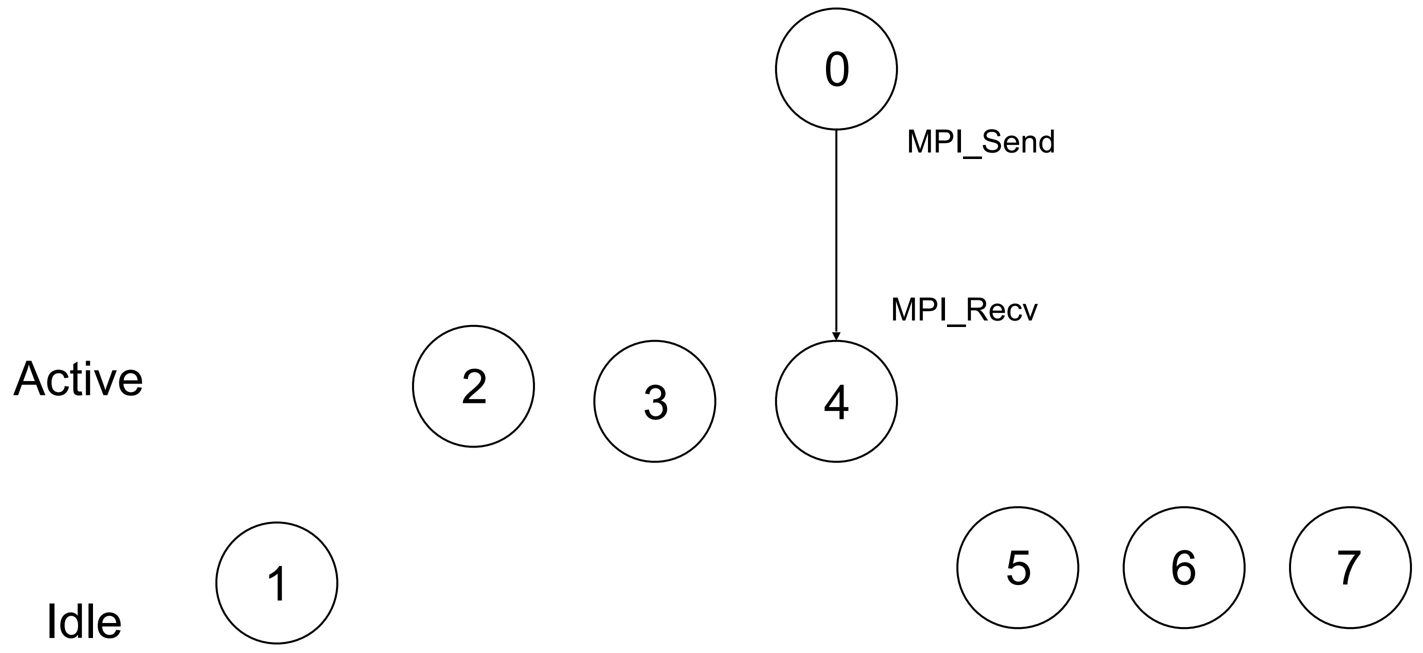
Blocking Type of Communications in Trapezoid Rule Program



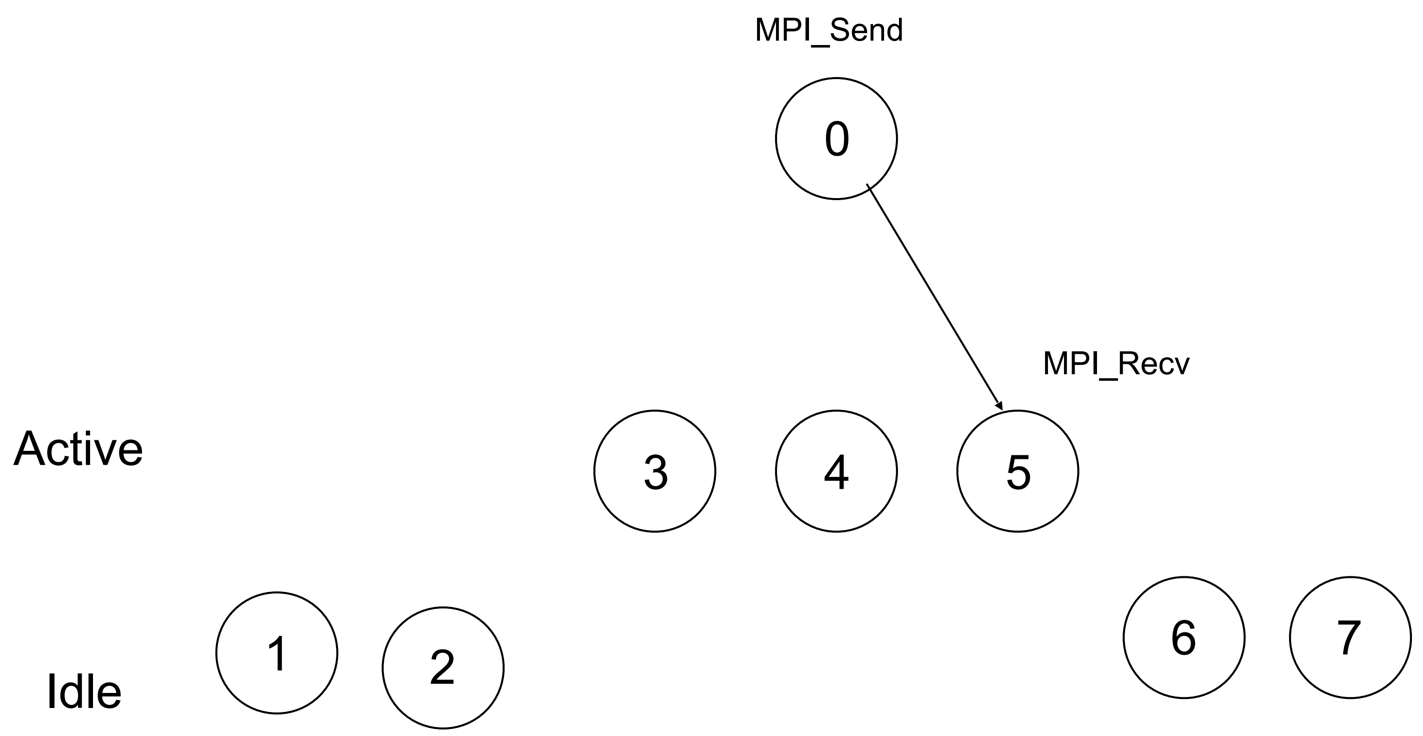
Blocking Type of Communications in Trapezoid Rule Program



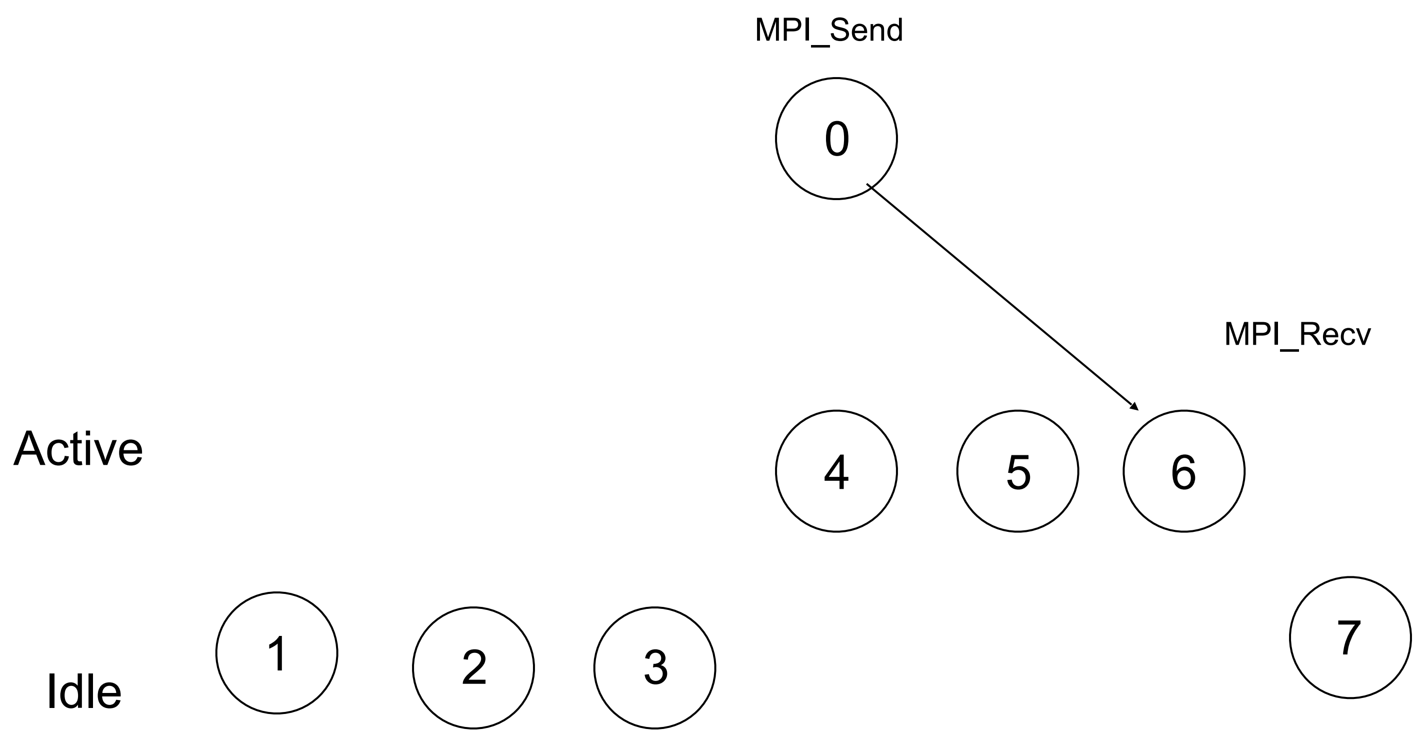
Blocking Type of Communications in Trapezoid Rule Program



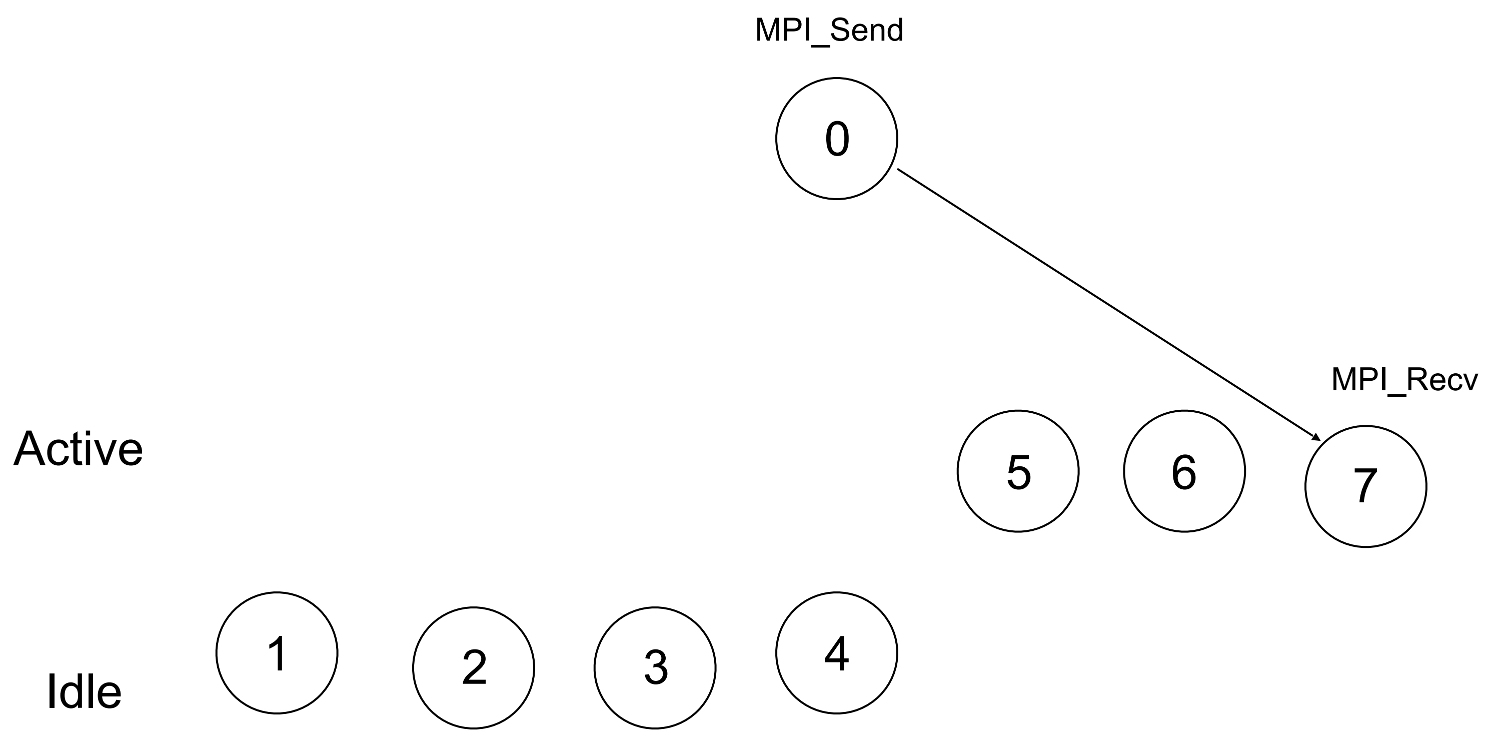
Blocking Type of Communications in Trapezoid Rule Program



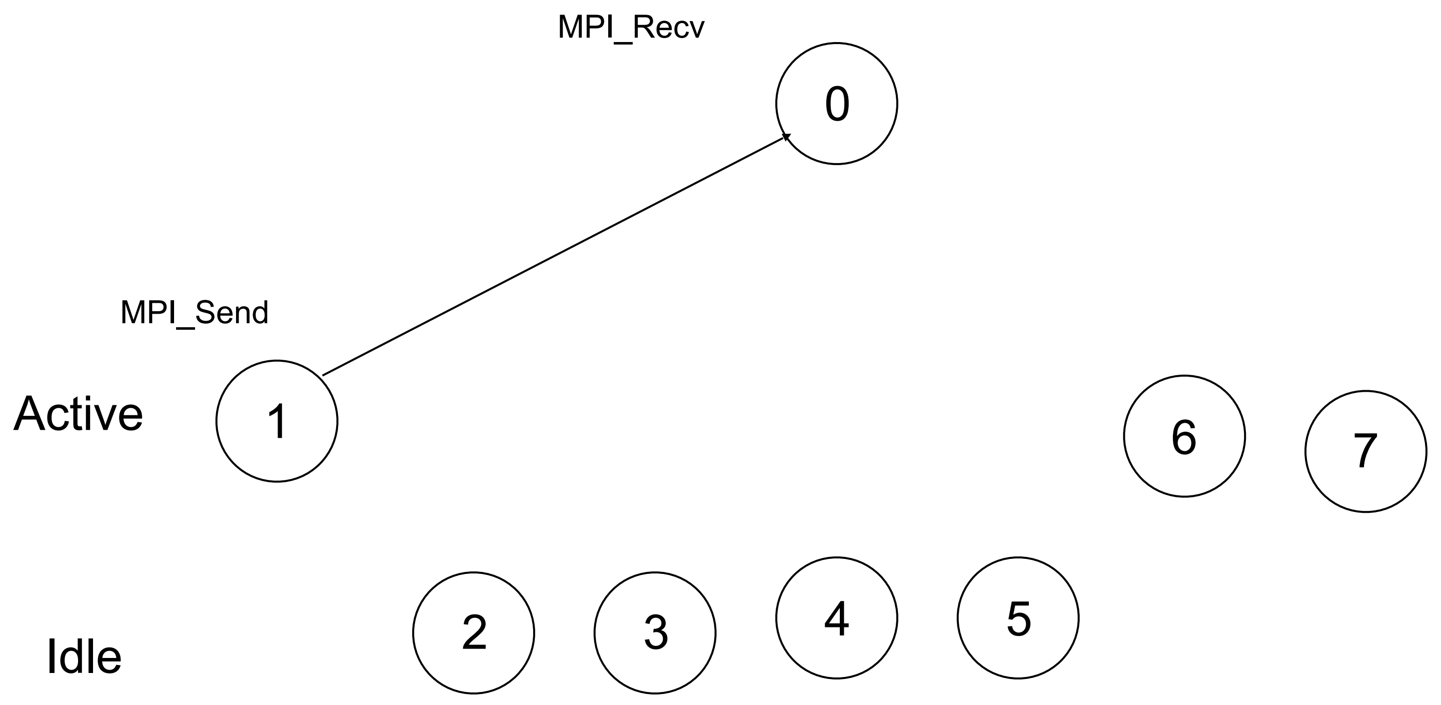
Blocking Type of Communications in Trapezoid Rule Program



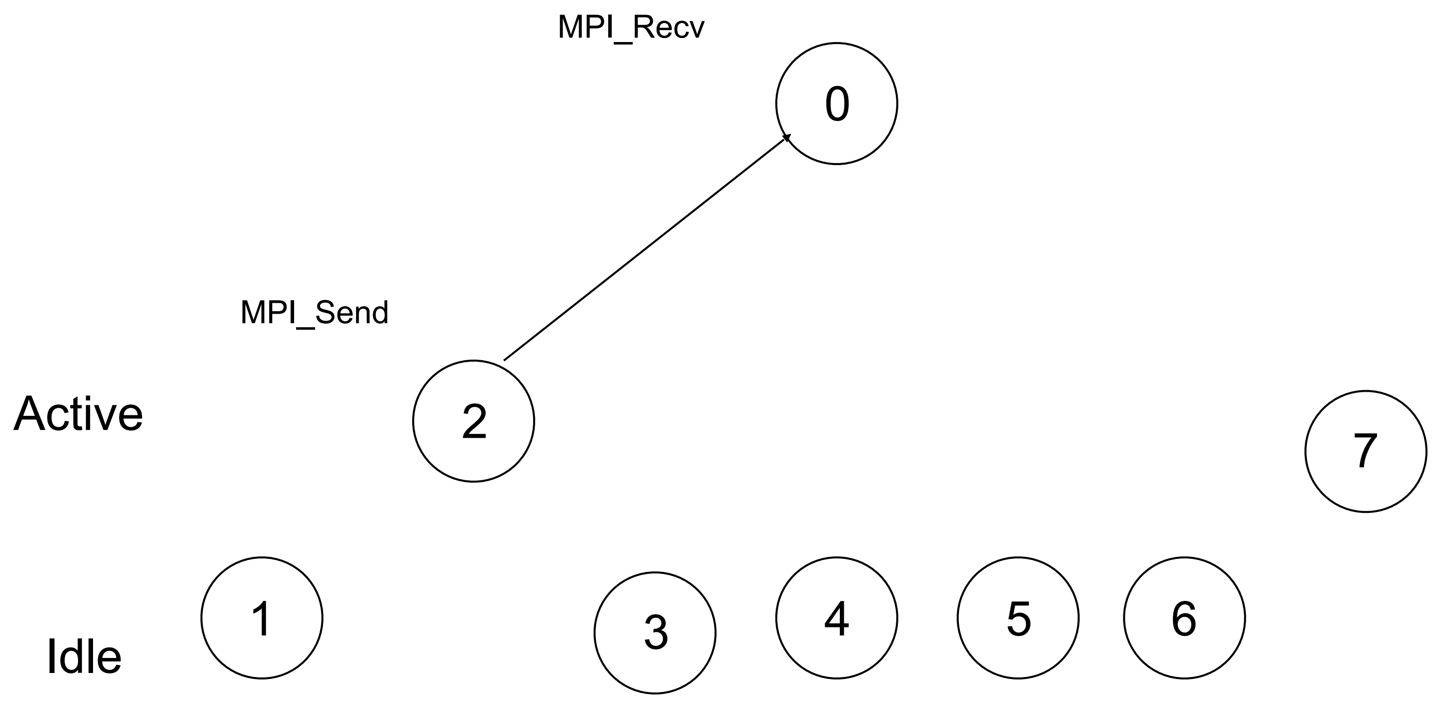
Blocking Type of Communications in Trapezoid Rule Program



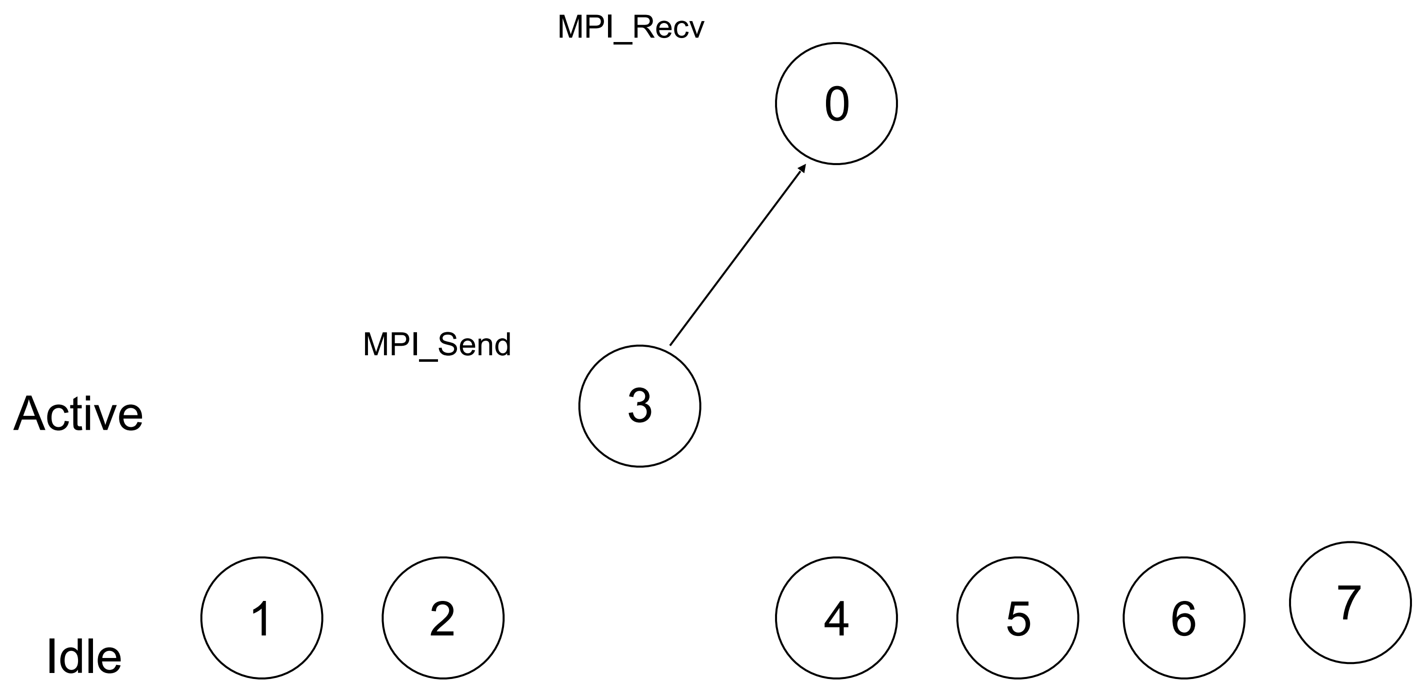
Blocking Type of Communications in Trapezoid Rule Program



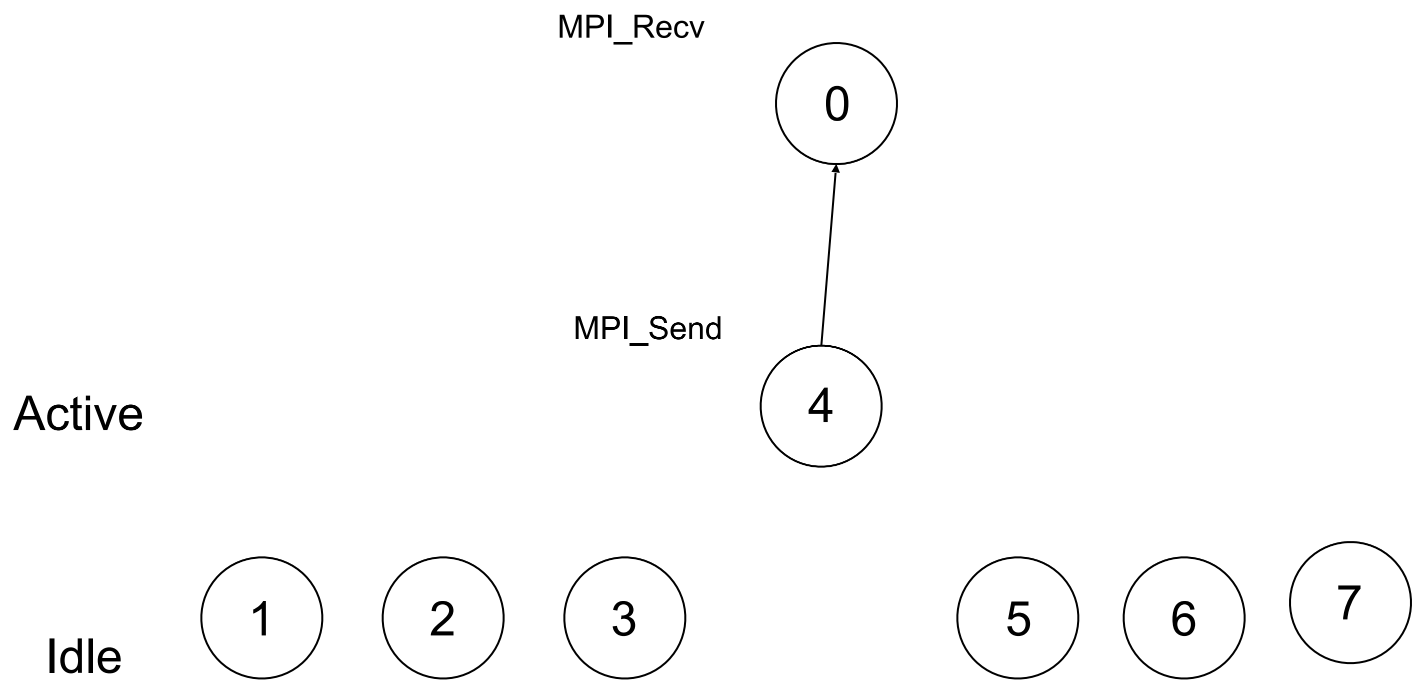
Blocking Type of Communications in Trapezoid Rule Program



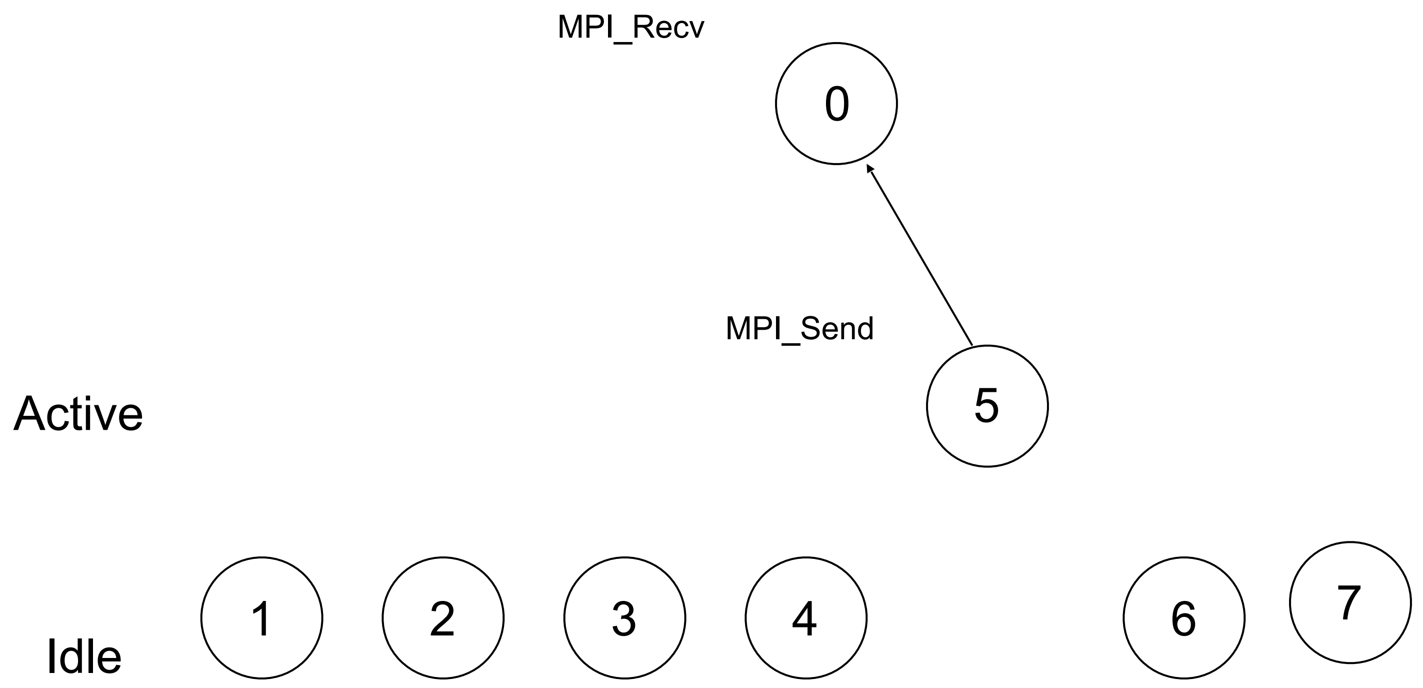
Blocking Type of Communications in Trapezoid Rule Program



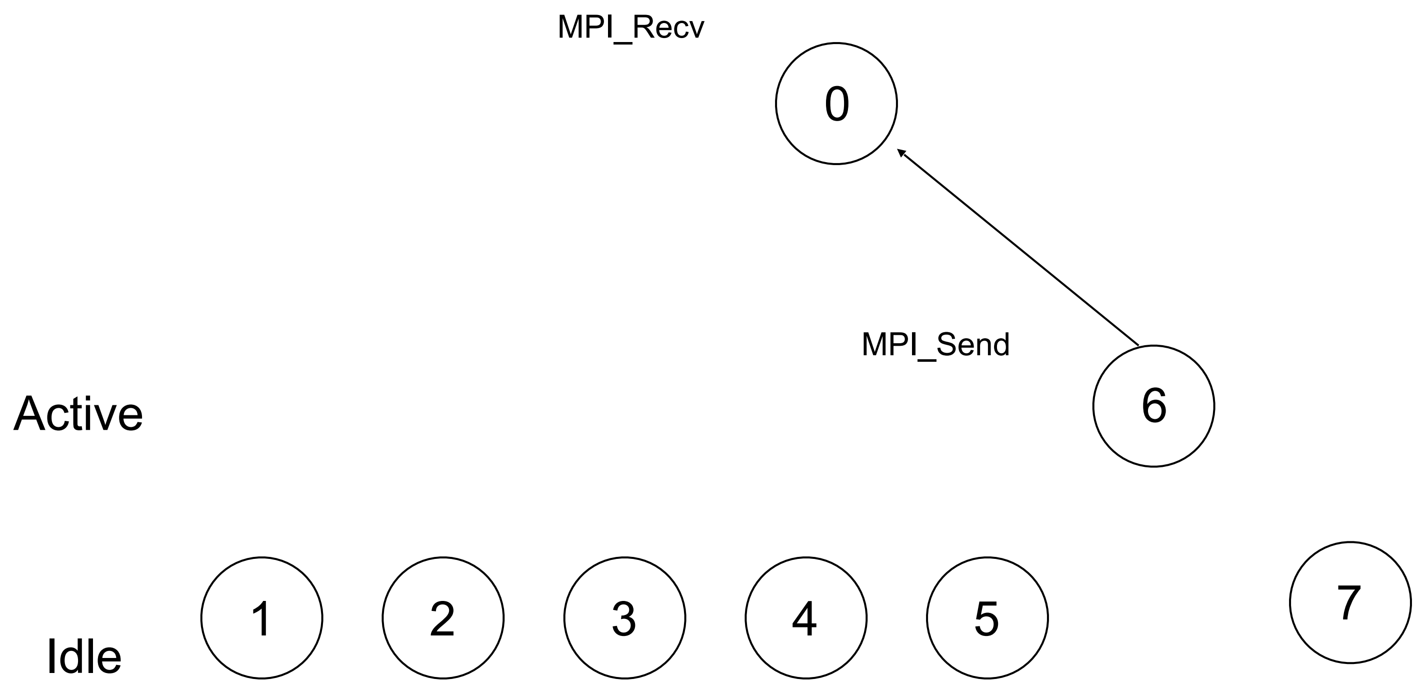
Blocking Type of Communications in Trapezoid Rule Program



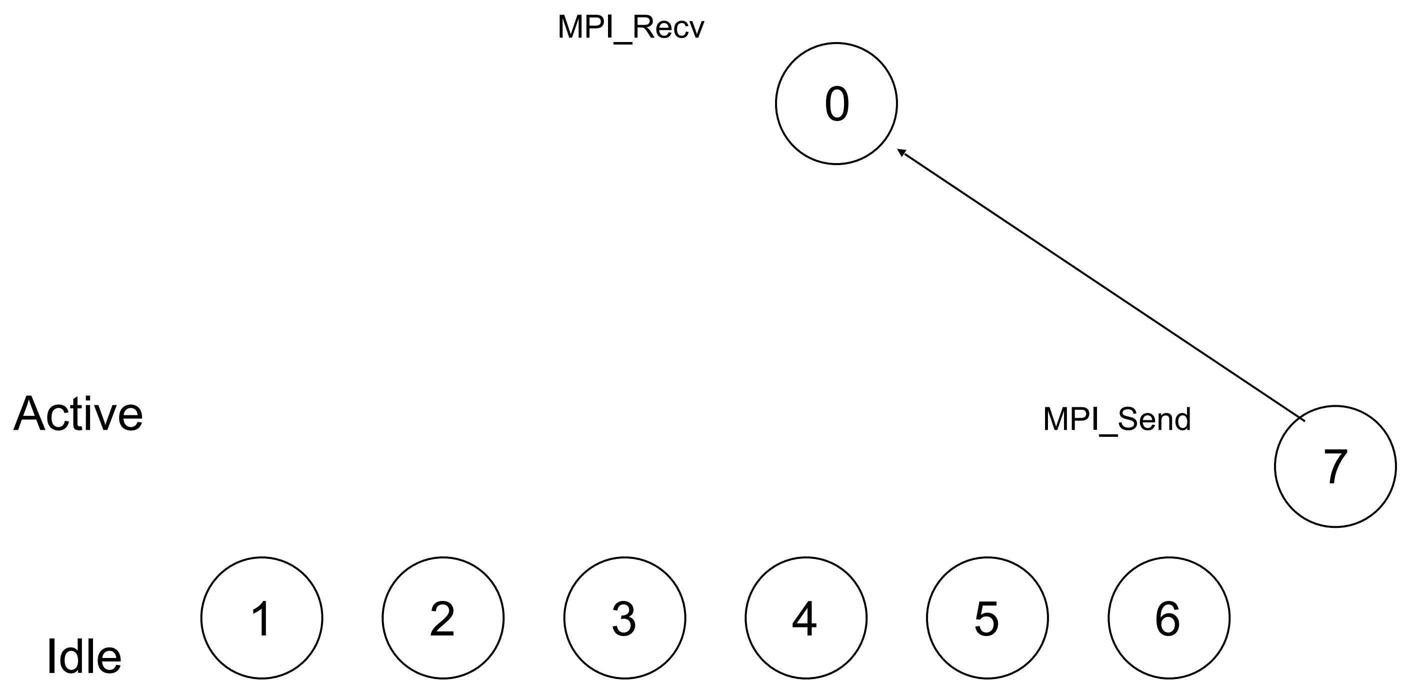
Blocking Type of Communications in Trapezoid Rule Program



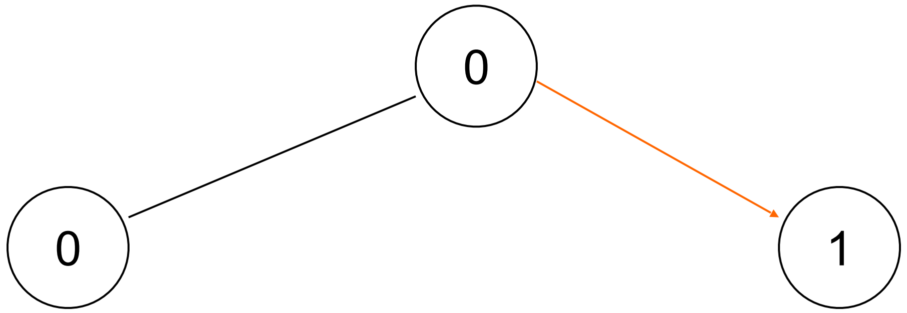
Blocking Type of Communications in Trapezoid Rule Program



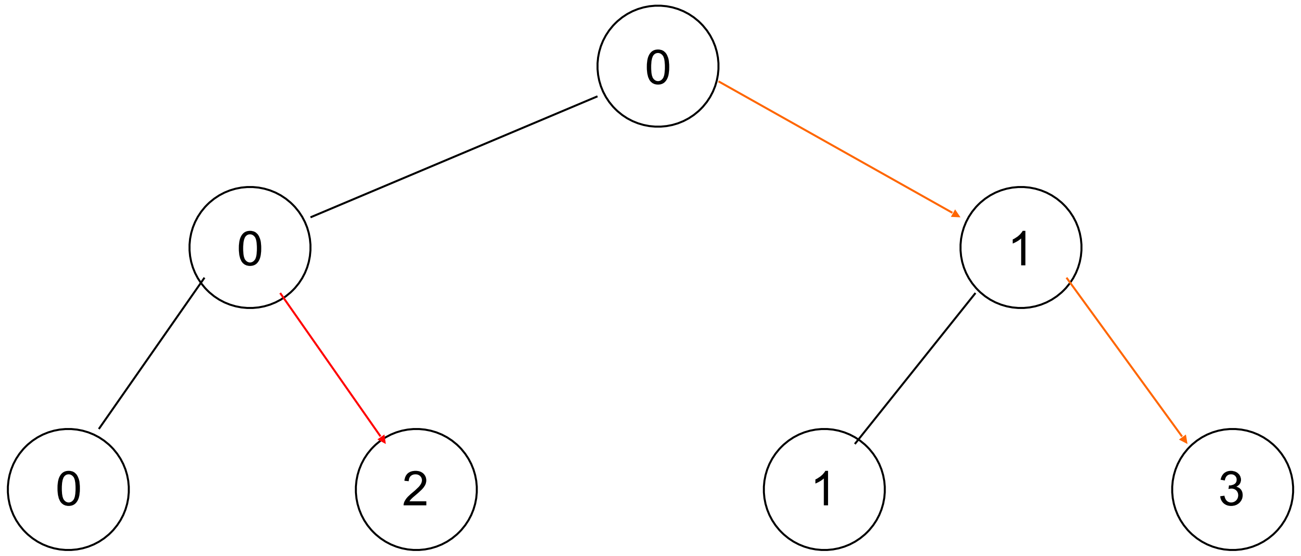
Blocking Type of Communications in Trapezoid Rule Program



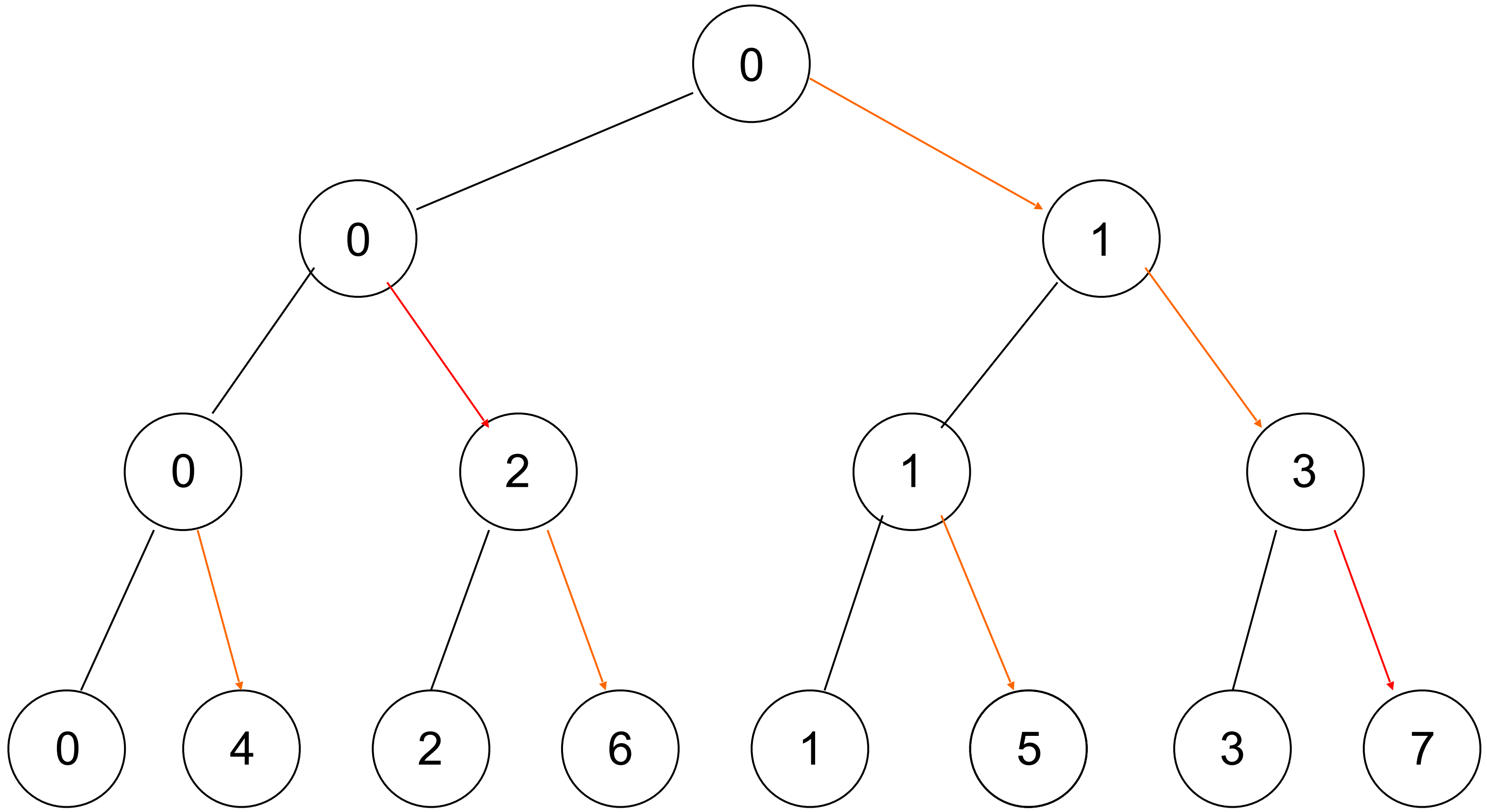
Processed Configured as a Tree



Processed Configured as a Tree



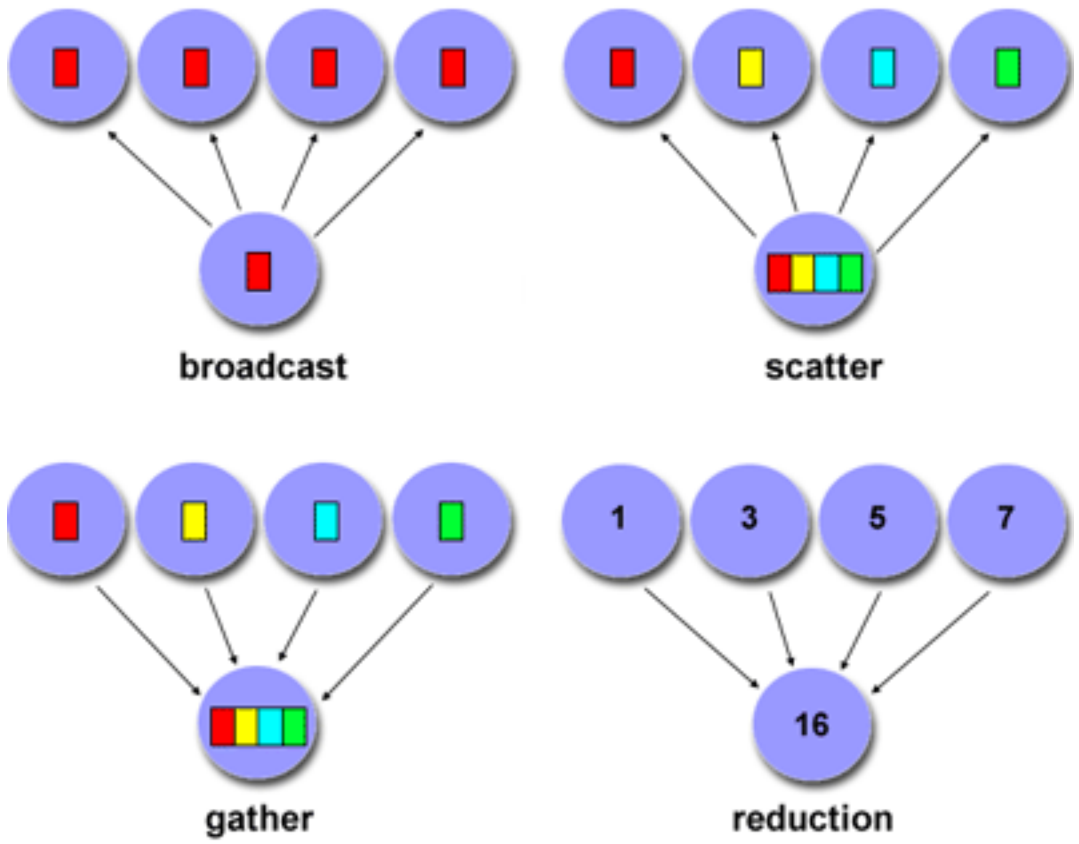
Processed Configured as a Tree



Collective Communication

A communication pattern that involves all the processes in a communicator is a Collective Communication.

Interactions



Broadcast

A Broadcast is a collective communication in which a single process sends data to every process in communicator.

Reducing

Reducing is the act of receiving data from all processes onto one process and performing a simple action on it to get a final result.

Broadcasting and Reducing in MPI

These are very common MPI operations and they have special functions

MPI_Bcast

MPI_Reduce

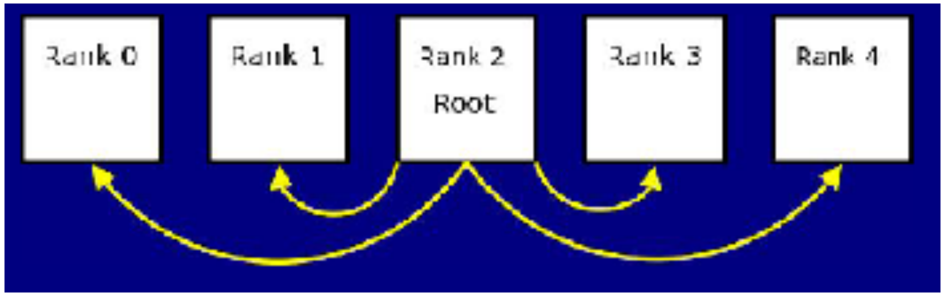
These can simplify things a lot!

MPI_Bcast

MPI_Bcast sends data from *root* process to all others

- **On root rank, *message* is send location**
- **On other ranks, *message* is receive location**

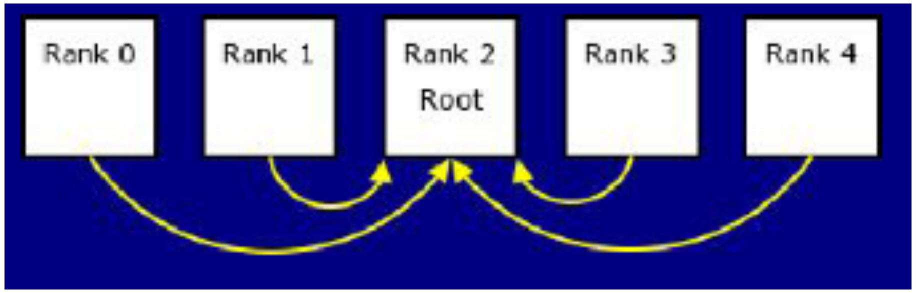
```
int MPI_Bcast(
        int count,                  /* in */
        void *message,              /* in/out  */
        MPI_Datatype datatype,      /* in */
        int root,                   /* in */
        MPI_Comm comm               /* in */)
```



MPI_Reduce

MPI_Reduce combines the operands stored in the memory referenced by *operand* using operation *operator* and stores the result in **result* on process *root*.

```
int MPI_Reduce(
        void*           operand,        /* in */
        void*           result,         /* out      */
        int         count,          /* in */
        MPI_Datatype    datatype,       /* in */
        MPI_Op          operator,/* in */
        int         root,           /* in */
        MPI_Comm    comm            /* in */)
```



MPI_Reduce

Both *operand* **and** *result* **refer to** *count* **memory locations with type** *datatype***.**

MPI_Reduce must be called by all processes in the communication *comm*

count, datatype, operator, and *root* must be the same on each process

MPI_Reduce

operator can take on one of the followed predefined values

Operation name	*Meaning*
MPI_MAX	**Maximum**
MPI_MIN	**Minimum**
MPI_SUM	**Sum**
MPI_PROD	**Product**
MPI_LAND	**Logical and**
MPI_BAND	**Bitwise and**
MPI_LOR	**Logical or**
MPI_BOR	**Bitwise or**
MPI_LXOR	**Logical exclusive or**
MPI_BXOR	**Bitwise exclusive or**
MPI_MAXLOC	**Maximum and location of maximum**
MPI_MINLOC	**Minimum and location of minimum**

MPI_Reduce

Example for Trapezoid Rule program

```
MPI_Reduce(&integral, &total, 1, MPI_DOUBLE,
              MPI_SUM, 0,  MPI_COMM_WORLD);
```

Sources of Overheads in Parallel Programs

Sources of Overheads in Parallel Programs

1. **Inter-process interactions: Processors working on any non-trivial parallel problem will need to talk to each other.**

Sources of Overheads in Parallel Programs

1. **Inter-process interactions: Processors working on any non-trivial parallel problem will need to talk to each other.**

2. **Idling: Processes may idle because of load imbalance, synchronization, or serial components.**

Sources of Overheads in Parallel Programs

1. Inter-process interactions: Processors working on any non-trivial parallel problem will need to talk to each other.

2. Idling: Processes may idle because of load imbalance, synchronization, or serial components.

3. Excess Computation: This is computation not performed by the serial version. This might be because the serial algorithm is difficult to parallelize, or that some computations are repeated across processors to minimize communication.

Synchronization in MPI

```
int MPI_Barrier( MPI_Comm comm /* in */)
```

This function causes each process in comm to block until every process in comm has called it.

Timing in MPI

double MPI_Wtime(void)

Returns an elapsed time on the calling processor in seconds since an arbitrary time in the past. If a process is interrupted by the system, the time it spends idle will be added into the elapsed time.

This is a function, declared as DOUBLE PRECISION MPI_WTIME() in Fortran.

Running on a supercomputer or cluster

- **Most supercomputers are batch oriented.**
- **You submit a job to a queue**
  - You specify the time you need
  - The resources required
  - Queues tend to have constraints
- **The scheduler then schedules a job and it will run when it's time slot is reached**
- **If you run over your time it will terminate you job**
- **Two common ones are SLURM and PBS/Torque**

Example PBS

```
#!/bin/sh
#PBS -N pbs_test
#PBS -l nodes=4:ppn=4:compute,walltime=1:00:00
#PBS -q small
# usmall is 32 items on the queue usmall2 is 8
#PBS -j oe

echo PBS_JOBNAME    = $PBS_JOBNAME
echo PBS_O_WORKDIR = $PBS_O_WORKDIR
echo PBS_TASKNUM    = $PBS_TASKNUM
echo PBS_NODENUM    = $PBS_NODENUM
echo PBS_NUM_NODES = $PBS_NUM_NODES
echo PBS_NUM_PPN    = $PBS_NUM_PPN
echo PBS_O_HOST     = $PBS_O_HOST
echo PBS_NP         = $PBS_NP
echo PBS_DEFAULT    = $PBS_DEFAULT
echo PBS_NODEFILE   = $PBS_NODEFILE
echo PBS_JOBID      = $PBS_JOBID

echo '--------'
cat $PBS_NODEFILE
echo '--------'

cd $PBS_O_WORKDIR
pwd

mkdir -p /scratch/kevin/temp
temp_file=$(mktemp -p /scratch/kevin/temp train_gw.XXXXXXXXXX)
```

Example SLURM

```
#!/bin/sh
# SLURM directives
#
#SBATCH --job-name=slurm_test
#SBATCH --time=01:00:00
#SBATCH --nodes=4
#SBATCH --tasks-per-node=4

echo SBATCH_MEM_BIND                 = $SBATCH_MEM_BIND
echo SBATCH_MEM_BIND_LIST            = $SBATCH_MEM_BIND_LIST
echo SBATCH_MEM_BIND_PREFER          = $SBATCH_MEM_BIND_PREFER
echo SBATCH_MEM_BIND_TYPE            = $SBATCH_MEM_BIND_TYPE
echo SBATCH_MEM_BIND_VERBOSE         = $SBATCH_MEM_BIND_VERBOSE
#echo SLURM_*_PACK_GROUP_#            = $SLURM_*_PACK_GROUP_#
echo SLURM_ARRAY_TASK_COUNT          = $SLURM_ARRAY_TASK_COUNT
echo SLURM_ARRAY_TASK_ID             = $SLURM_ARRAY_TASK_ID
echo SLURM_ARRAY_TASK_MAX            = $SLURM_ARRAY_TASK_MAX
echo SLURM_ARRAY_TASK_MIN            = $SLURM_ARRAY_TASK_MIN
echo SLURM_ARRAY_TASK_STEP           = $SLURM_ARRAY_TASK_STEP
echo SLURM_ARRAY_JOB_ID              = $SLURM_ARRAY_JOB_ID
```

mpiexec

Launch 16 processes on the local node:

mpiexec –n 16 ./test

Launch 16 processes on 4 nodes (each has 4 cores)

mpiexec –hosts h1:4,h2:4,h3:4,h4:4 –n 16 ./test

Runs the first four processes on h1, the next four on h2, etc.

mpiexec –hosts h1,h2,h3,h4 –n 16 ./test

Runs the first process on h1, the second on h2, etc., and wraps around

So, h1 will have the 1st, 5th, 9th and 13th processes

mpiexec

If there are many nodes, it might be easier to create a host file

```
cat hf
  h1:4
  h2:2
mpiexec –hostfile hf –n 16 ./test
```

Thanks

To my colleague Dr Slava Kitaeff for letting me “borrow” many of his slides