Parallel Programming with Message Passing Interface (MPI) – Day 1

SHARCNET Summer School

May 30, 2017

Jemmy Hu

SHARCNET
(519) 888-4567 x37854
jemmyhu@sharcnet.ca

Day 1 Contents

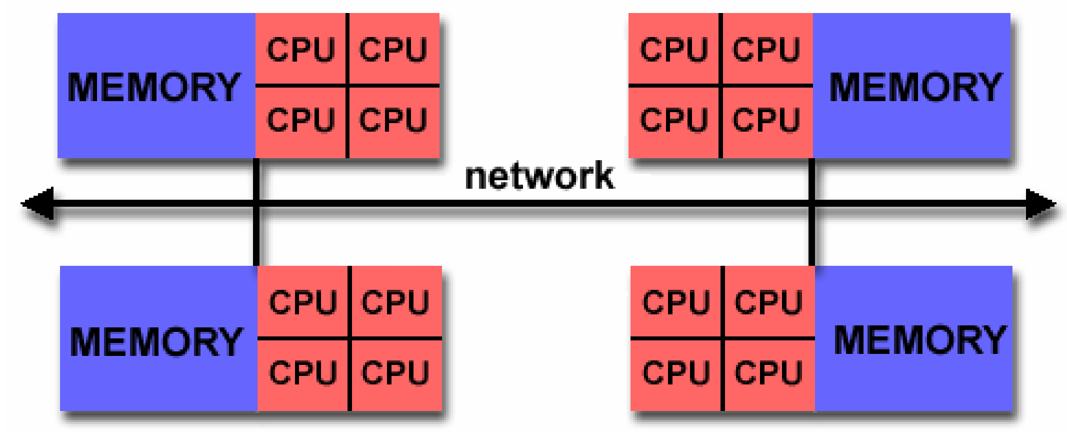
- Parallel Computing Concept
- MPI Basics
- MPI Communications
  - Point-to-point (Send & Receive)
  - Collectives
- Examples and hand-on exercises through the lecture
- References

HPC - High Performance Computing

- Definition is nebulous (CS vs Application)
  - resource (processing) intensive computation
  - computing where the need for speed is compelling
    - computing nearer the limit of what is feasible (clock time, …)
  - parallel computing (this is too strict): use as many cpus as possible for an application, speed, speed, speed …
- In reality, HPC is concerned with varied issues involving:
  - Hardware (cpu speed, networking)
    - pipelining, instruction sets, multi-processors, inter-connects
  - algorithms
    - efficiency, techniques for concurrency
  - software
    - compilers (optimization/parallelization), libraries

Parallel Computer Memory Architectures

Distributed Memory (Cluster solution)



- Employ both shared and distributed memory architectures

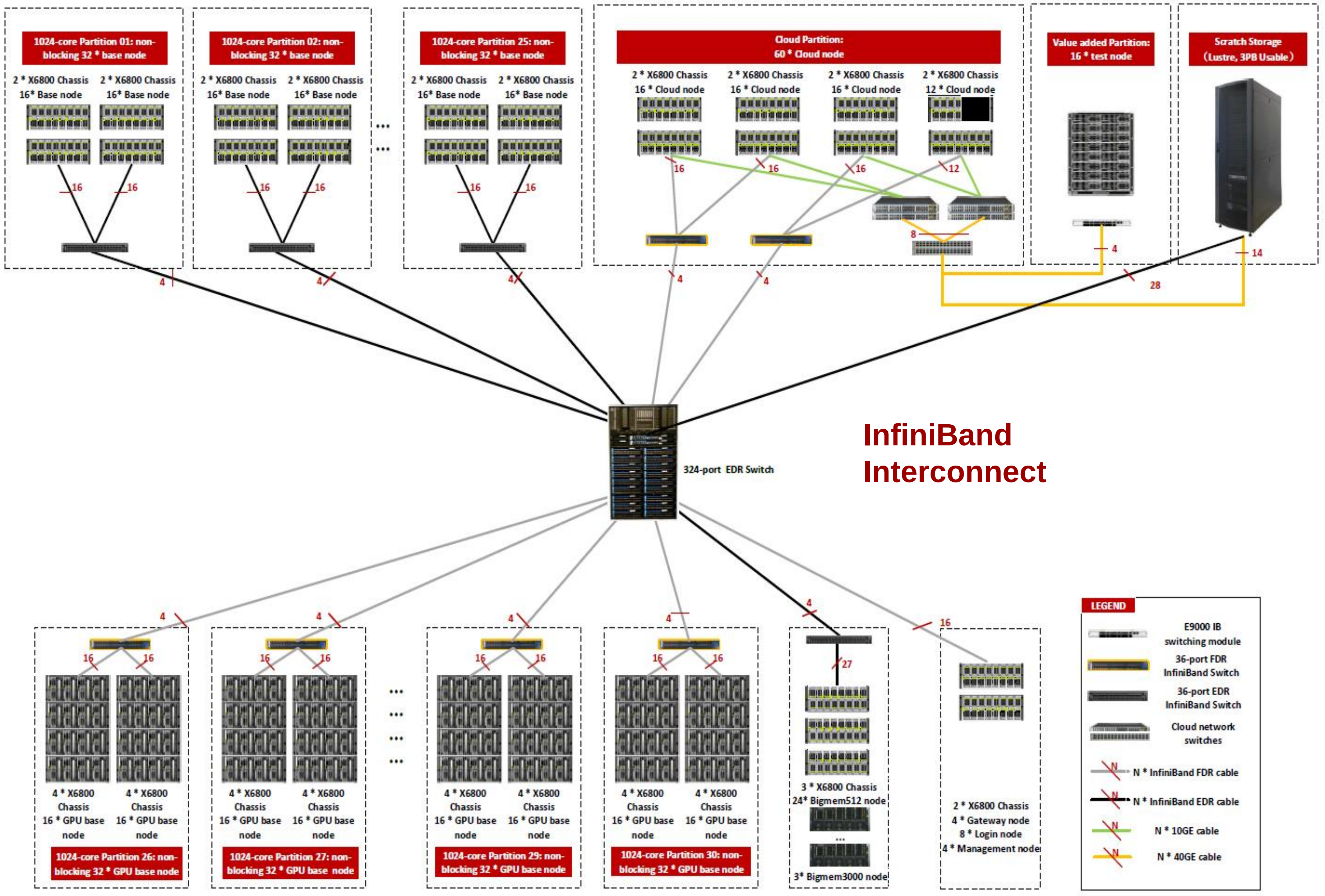
- The shared memory component is usually a cache coherent SMP machine. Processors on a given SMP can address that machine's memory as global.

- The distributed memory component is the networking of multiple SMPs. SMPs know only about their own memory - not the memory on another SMP. Therefore, network communications are required to move data from one SMP to another.

Example: SHARCNET cluster systems

System	No of Nodes	CPU/Node	Node type
saw	336	8	Intel Xeon
orca	392 (320 + 72)	16, 24	AMD Opteron + Intel Xeon
graham	~1400?	32, mix?	Intel Xeon +GPU

Example: Cluster Overview


1024-core Partition 01: non-blocking 32 * base node
2 * X6800 Chassis 16* Base node
2 * X6800 Chassis 16* Base node
1024-core Partition 02: non-blocking 32 * base node
2 * X6800 Chassis 16* Base node
2 * X6800 Chassis 16* Base node
1024-core Partition 25: non-blocking 32 * base node
2 * X6800 Chassis 16* Base node
2 * X6800 Chassis 16* Base node
Cloud Partition: 60 * Cloud node
2 * X6800 Chassis 16 * Cloud node
2 * X6800 Chassis 16 * Cloud node
2 * X6800 Chassis 16 * Cloud node
2 * X6800 Chassis 12 * Cloud node
Value added Partition: 16 * test node
Scratch Storage (Lustre, 3PB Usable)
16
16
4
4
4
12
8
28
14
324-port EDR Switch
27
1024-core Partition 26: non-blocking 32 * GPU base node
4 * X6800 Chassis 16 * GPU base node
4 * X6800 Chassis 16 * GPU base node
1024-core Partition 27: non-blocking 32 * GPU base node
1024-core Partition 29: non-blocking 32 * GPU base node
1024-core Partition 30: non-blocking 32 * GPU base node
3 * X6800 Chassis
24* Bigmem512 node
3* Bigmem3000 node
2 * X6800 Chassis
4 * Gateway node
8 * Login node
4 * Management node
LEGEND
E9000 IB switching module
36-port FDR InfiniBand Switch
36-port EDR InfiniBand Switch
Cloud network switches
N * InfiniBand FDR cable
N * InfiniBand EDR cable
N * 10GE cable
N * 40GE cable

InfiniBand Interconnect

Parallel Computing: What is it?

- **Parallel computing is when a program uses concurrency to either:**

  - decrease the runtime for the solution to a problem.

  - Increase the size of the problem that can be solved.

Gives you more performance to throw at your problems.

- **Parallel programming is not generally trivial, 3 aspects:**
  - specifying parallel execution
  - communicating between multiple procs/threads
  - synchronization

  tools for automated parallelism are either highly specialized or absent

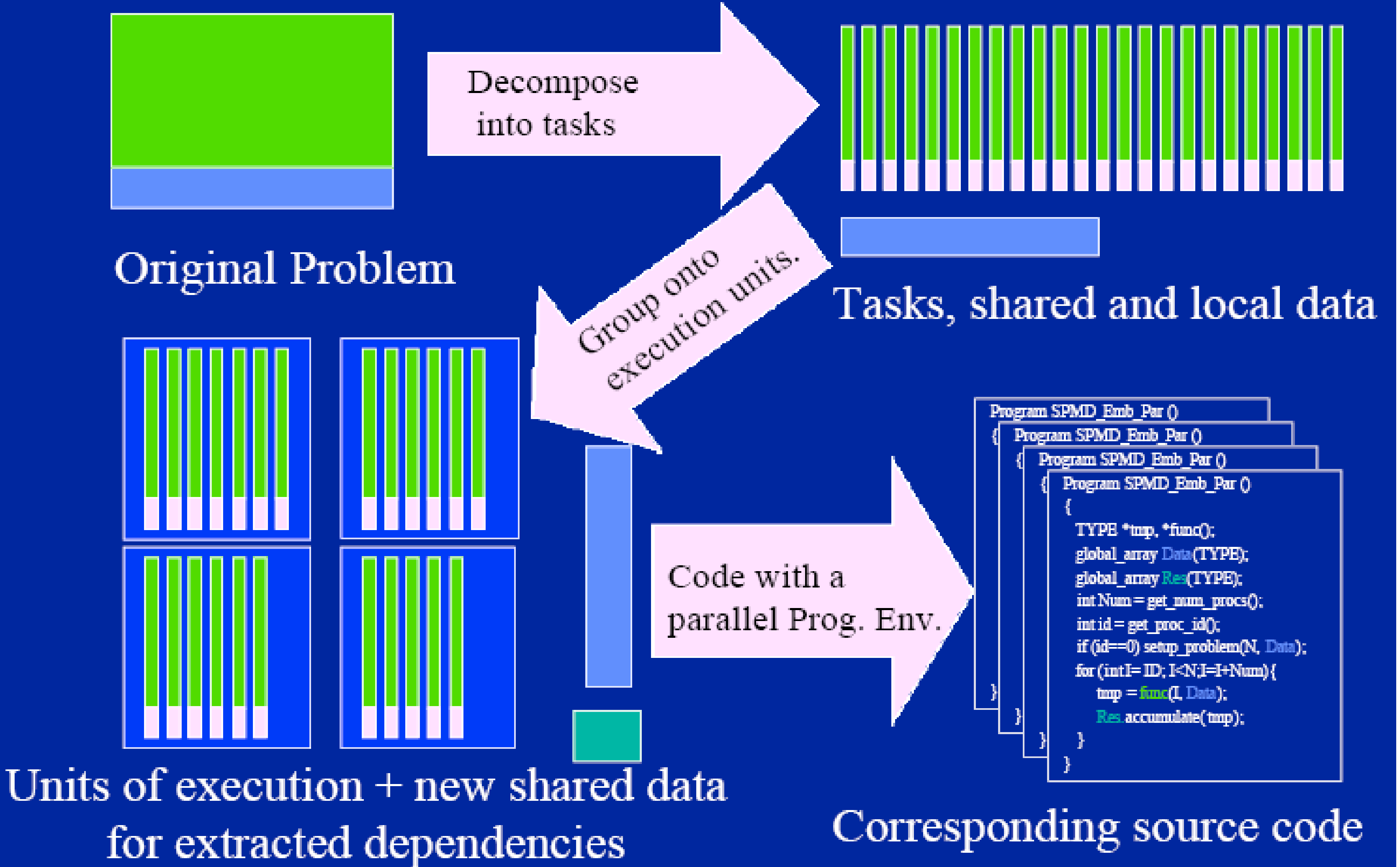
- **Many issues need to be considered, many of which don’t have an analog in serial computing**
  - data vs. task parallelism
  - problem structure
  - parallel granularity

Parallel Computing – problem faced

- <u>Hardware</u>
  - in order to facilitate processors working together they must be able to communicate
  - interconnect hardware is complex
    - sharing memory is easy to say, harder to realize as system scales
    - communication over any kind of network is still a challenge compared to bus speed --- overhead can be significant

- <u>Software</u>
  - parallel algorithms are actually fairly well understood
  - the realization of algorithms in software is non-trivial
  - compilers
    - automated parallelism is difficult
  - design
    - portability and power are typically at odds with each other

Parallel Computing:
Writing a parallel application.


Original Problem
Decompose into tasks
Tasks, shared and local data
Group onto execution units.
Units of execution + new shared data for extracted dependencies
Code with a parallel Prog. Env.
Corresponding source code

```
Program SPMD_Emb_Par ()
{
  TYPE *tmp, *func();
  global_array Data(TYPE);
  global_array Res(TYPE);
  int Num = get_num_procs();
  int id = get_proc_id();
  if (id==0) setup_problem(N, Data);
  for (int I= ID; I<N;I=I+Num){
      tmp = func(I, Data);
      Res.accumulate( tmp);
  }
}
```

Parallel Computing – software programming model

- Distributed memory systems
  - For processors to share data, the programmer must explicitly arrange for communication -**“Message Passing”**
  - Message passing libraries:
    - MPI (“Message Passing Interface”)

- Shared memory systems
  - “Thread” based programming (pthread, …)
  - Compiler directives (OpenMP)
  - Can also do explicit message passing, of course

Programming Models

- languages

  C, C++, Fortran

- threaded (shared memory)

  OpenMP, pthreads

- message passing (distributed memory)

  MPI, PGAS (UPC, Coarray Fortran)

- accelerator (GPU, Cell, MIC, FPGA)

  CUDA, OpenCL, OpenACC

Task (function, control) Parallelism

Work to be done is decomposed across processors

- e.g. divide and conquer
- each processor responsible for some part of the algorithm
- communication mechanism is significant
- must be possible for different processors to perform different tasks

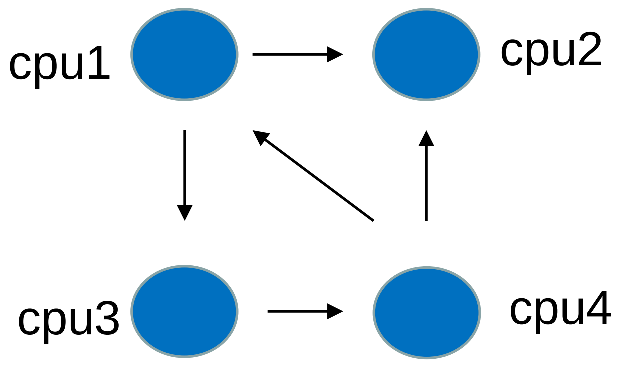
MPI Basics

Distributed Memory: Clusters

- Each cpu is independent!

- Parallel code consists of programs running on separate cpus, communicating with each other.

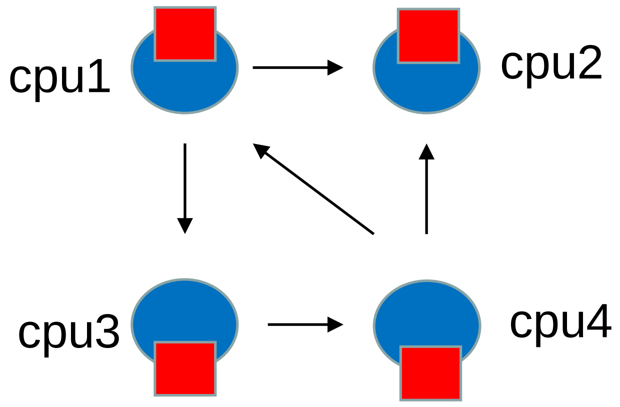
- Each processor could run entirely different programs.



Distributed Memory: Clusters

- Each cpu/node has own memory!

- Whenever it needs data from another region, requests it from that CPU.
- Usual model: “message passing"



Hardware:

Easy to build (Harder to build well)
Can build larger and larger clusters relatively easily

Software:

Every communication has to be hand-coded: hard to program

MPI - what is it?

- An open standard library interface for message passing ratified by the MPI Forum: https://www.open-mpi.org/
- primarily addresses the ***message-passing parallel programming model:*** data is moved from the address space of one process to that of another process through cooperative operations on each process
- The MPI standard has gone through a number of revisions, with the most recent version being MPI-3.x
- Interface specifications have been defined for C/C++ and Fortran language bindings
- Actual MPI library implementations differ in which version and features of the MPI standard they support, e.g.:

  openmpi v2.1.0

  intelmpi 2017

  mpich 3.x

Reasons for Using MPI

- **Standardization** - MPI is the only message passing library that can be considered a standard. It is supported on virtually all HPC platforms. Practically, it has replaced all previous message passing libraries.

- **Portability** - There is little or no need to modify your source code when you port your application to a different platform that supports (and is compliant with) the MPI standard.

- **Performance Opportunities** - Vendor implementations should be able to exploit native hardware features to optimize performance. Any implementation is free to develop optimized algorithms.

- **Functionality** - There are ***over 430*** routines defined in MPI-3, which includes the majority of those in MPI-2 and MPI-1.

- **Availability** - A variety of implementations are available, both vendor and public domain.

MPI: a **Library** for Message-Passing

- Not built in to compiler
- Function calls that can be made from any compiler, many languages
- Just link to it
- Header files: mpi.h, mpif.h
- Compiler Wrappers: mpicc, mpiCC, mpif77, mpif90

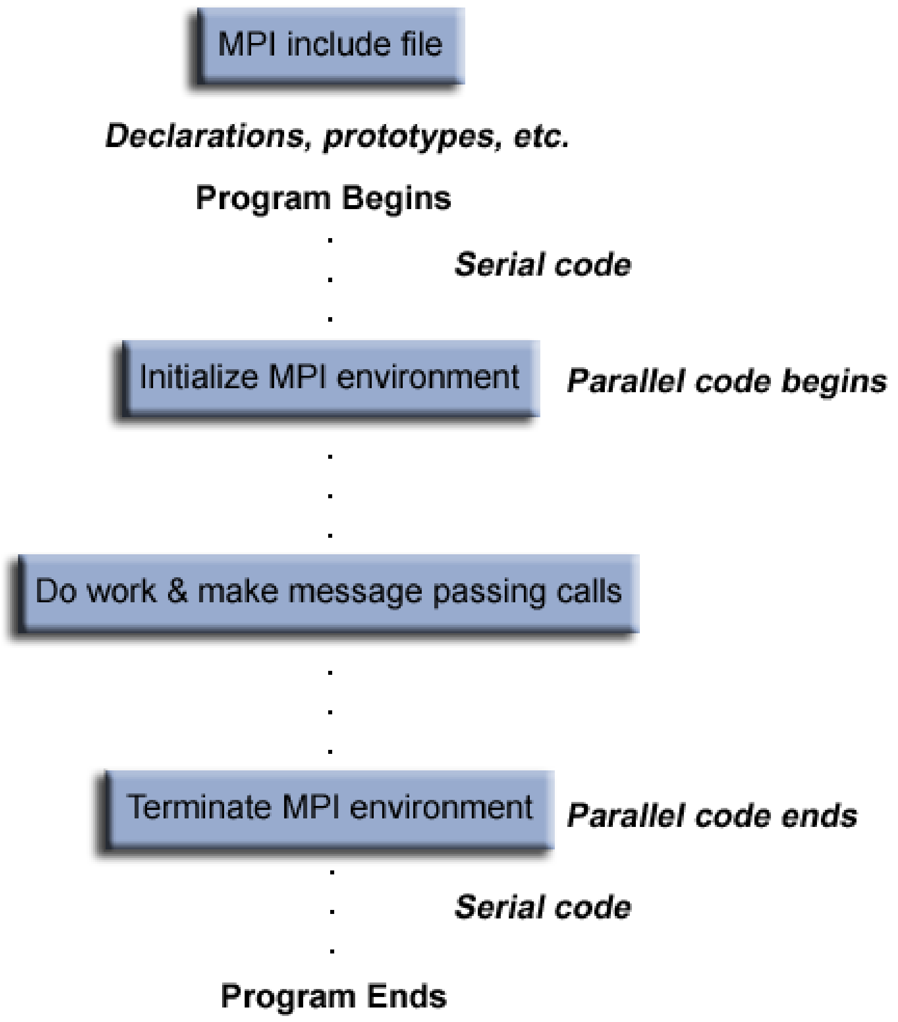
wrappers for the system C, Fortran compilers that have the various –I (header files), -L (libs) clauses in there automatically.

--showme (OpenMPI) shows which options are being used.

```
mpicc --showme -o mpi_comm_1 mpi_comm_1.c
```

```
icc -o mpi_comm_1 mpi_comm_1.c -I/opt/sharcnet/openmpi/1.6.2/intel/include
-pthread -L/opt/sharcnet/torque/current/lib -Wl,--rpath -Wl,
/opt/sharcnet/torque/current/lib -L/opt/sharcnet/openmpi/1.6.2/intel/lib -lmpi -
ltorque -ldl -lm -lnuma -Wl,--export-dynamic -lrt -lnsl -lutil
```

General MPI Program Structure:



MPI include file
Declarations, prototypes, etc.
Program Begins
Serial code
Initialize MPI environment
Parallel code begins
Do work & make message passing calls
Terminate MPI environment
Parallel code ends
Serial code
Program Ends

MPI - Environment Management Routines

Initializing and finalizing

MPI_Init: Initializes the MPI execution environment. This function must be called in every MPI program, must be called before any other MPI functions and must be called only once in an MPI program.

```
MPI_Init (&argc, &argv)
MPI_INIT (ierr)
```

MPI_Finalize: Terminates the MPI execution environment. This function should be the last MPI routine called in every MPI program - no other MPI routines may be called after it.

```
MPI_Finalize ()
MPI_FINALIZE (ierr)
```

Simplest example – helloworld.c

```
#include <stdio.h>

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

    printf("Hello World\n");

}
```

compile: `cc -o hello hello.c`

List files: `ls`

```
hello  hello.c
```

Run the compiled code:

```
./hello
Hello World
```

MPI – helloworld_1.c

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

Header file

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

    MPI_Init(&argc, &argv);
```

Initializes the MPI execution environment

```
    printf("Hello World\n");

    MPI_Finalize();
```

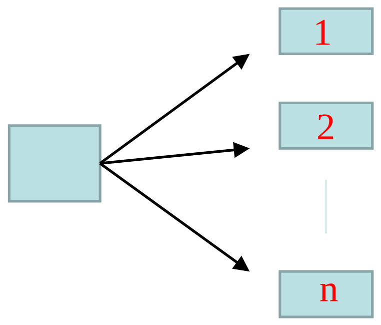
Terminates the MPI execution environment

```
}
```

Compile: `mpicc -o hello_mpi_1 hello_mpi_1.c`

mpirun

- Launches n processes, assigns each an MPI rank and starts the program
- For multi-node run, has a list of nodes, ssh's to each node and launches the program



- Number of processes to use is almost always equal to the number of processors (-np #)
- On your nodes, what happens when you run this?

```
mpirun –np 4 hello_mpi_1
```

```
$ mpirun -np 4 ./hello_mpi_1

Hello World
Hello World
Hello World
Hello World
```

```
$ hostname
saw331
$ mpirun -np 4 hostname
saw331
saw331
saw331
saw331
```

mpirun runs *any* program

- mpirun will start that process-launching procedure for any progam
- Sets variables somehow that mpi programs recognize so that they know which process they are

```
$ hostname
$ mpirun -np 4 hostname
$ ls
$ mpirun -np 4 ls
```

```
$ ls
hello  hello.c  hello_mpi_1  hello_mpi_1.c
$ mpirun -np 4 ls
hello  hello.c  hello_mpi_1  hello_mpi_1.c
hello  hello.c  hello_mpi_1  hello_mpi_1.c
hello  hello.c  hello_mpi_1  hello_mpi_1.c
hello  hello.c  hello_mpi_1  hello_mpi_1.c
```

Identifying the separate processes

How many processes are involved?

MPI_Comm_size: Returns the total number of MPI processes in the specified communicator, such as MPI_COMM_WORLD. If the communicator is MPI_COMM_WORLD, then it represents the number of MPI tasks available to your application.

```
MPI_Comm_size (comm, &size)
MPI_COMM_SIZE (comm, size, ierr)
```

Which process "am I" (the rank)

MPI_Comm_rank: Returns the rank of the calling MPI process within the specified communicator. Initially, each process will be assigned a unique integer rank between 0 and number of tasks - 1 within the communicator MPI_COMM_WORLD. This rank is often referred to as a task ID.

```
MPI_Comm_rank (comm, &rank)
MPI_COMM_RANK (comm, rank,ierr)
```

Hello_mpi_2.c

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

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

    int num_procs, my_id;

    MPI_Init(&argc, &argv);

    /* find out MY process ID, and how many processes were started. */
    MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

    printf("Hello world! I'm process %d out of %d processes\n", my_id, num_procs);

    MPI_Finalize();

}
```

```
mpicc -o hello_mpi_2 hello_mpi_2.c

mpirun -np 4 ./hello_mpi_2
```

```
Hello world! I'm process 3 out of 4 processes
Hello world! I'm process 0 out of 4 processes
Hello world! I'm process 2 out of 4 processes
Hello world! I'm process 1 out of 4 processes
```

Fortran example

```
program hello_mpi_f90
include 'mpif.h'

integer ierr, my_id, num_procs

call mpi_init(ierr)
call mpi_comm_rank(MPI_COMM_WORLD,my_id,ierr)
call mpi_comm_size(MPI_COMM_WORLD,num_procs,ierr)

! print rank and size to screen
print*, "Hello World! I am process ", my_id, "out of processors ", num_procs

call mpi_finalize(ierr)
end
```

```
mpif90 -o hello_mpi_f90 hello_mpi_f90.f90

mpirun -np 4 ./hello_mpi_f90
 Hello World! I am process           0 out of           4 processors
 Hello World! I am process           1 out of           4 processors
 Hello World! I am process           3 out of           4 processors
 Hello World! I am process           2 out of           4 processors
```

Each process can perform a different task, hello_play.c

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

main(int argc, char **argv) {
   int my_id, num_procs;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
   MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

   if ( my_id == 0 ) {
      /* do some work as process 0 */
   }
   else if ( my_id == 1 ) {
      /* do some work as process 1 */
   }
   else if ( my_id == 2 ) {
      /* do some work as process 2 */
   }
   else {
      /* do this work in any remaining processes */
   }
   /* Stop MPI processes */
   MPI_Finalize();
}
```

Hello_mpi_3.c

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

main(int argc, char **argv) {
   int my_id, num_procs;

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_id);
   MPI_Comm_size(MPI_COMM_WORLD, &num_procs);

   if ( my_id == 0 ) {
     printf("Hello! I'm the master with process ID %d\n",my_id);
   }
   else if ( my_id == 1 ) {
     printf("Hi, my process ID is %d\n",my_id);
   }
   else if ( my_id == 2 ) {
     printf("My process ID is %d\n",my_id);
   }
   else {
     printf("What! I'm the others, with process ID %d\n",my_id);
}
```

```
mpicc -o hello_mpi_3 hello_mpi_3.c


mpirun -np 8 ./hello_mpi_3

What! I'm the others, with process ID 7
What! I'm the others, with process ID 6
What! I'm the others, with process ID 4
Hello! I'm the master with process ID 0
What! I'm the others, with process ID 3
Hi, my process ID is 1
What! I'm the others, with process ID 5
My process ID is 2
```

Hands-on exercise 1- getting started

- login to saw
- ssh to one of saw-dev1, saw-dev2, saw-dev3, saw-dev4, saw-dev5, saw-dev6
- Go to your /home or /work directory
- Copy codes all over

  ```
  cp –rf /work/jemmyhu/ss2017/MPI_2017 .
  ```
- Go to your own

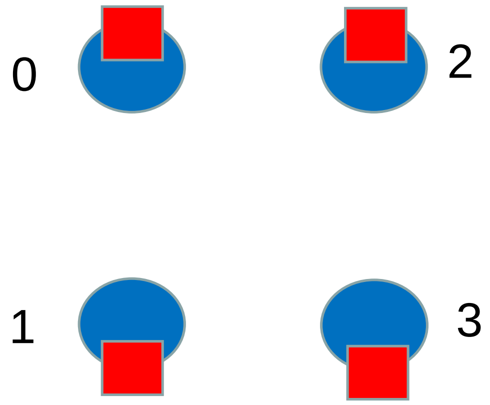
  ```
  cd ~/MPI_2017/codes/hello
  ```
- Run the hello world examples
- Modify hello_play.c and run

MPI – communicator

MPI groups processes into communicators.

- Each communicator has some size
  -- number of tasks.
- Each task has a rank 0 … size-1
- Every task in your program belongs to

MPI_COMM_WORLD :
  Global Communicator
MPI_Comm_rank(MPI_COMM_WORLD, &rank) :
  Get current tasks rank
MPI_Comm_size(MPI_COMM_WORLD, &size) :
  Get communicator size



MPI_COMM_WORLD :
Size = 4, ranks = 0 … 3

- Can create our own communicators over the same tasks
- May break the tasks up into subgroups
- May just re-order them for some reason

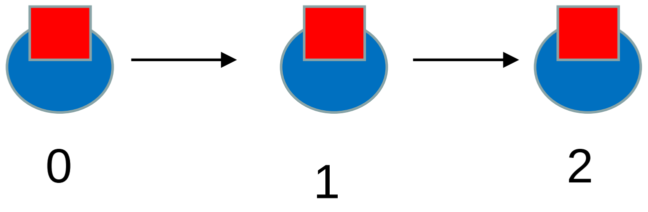
MPI – point-to-point communication routines

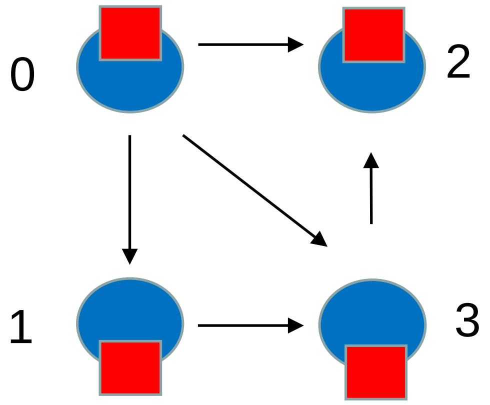
•MPI_Send: to send a message to another process

The syntax of MPI_Send is:

```
MPI_Send(void *data_to_send, int send_count, MPI_Datatype send_type,
                  int destination_ID, int tag, MPI_Comm comm);
```

•data_to_send: variable of a C type that corresponds to the send_type supplied below
•send_count: number of data elements to be sent (nonnegative int)
•send_type: datatype of the data to be sent (one of the MPI datatype handles)
•destination_ID: process ID of destination (int)
•tag: message tag (int)
•comm: communicator (handle)





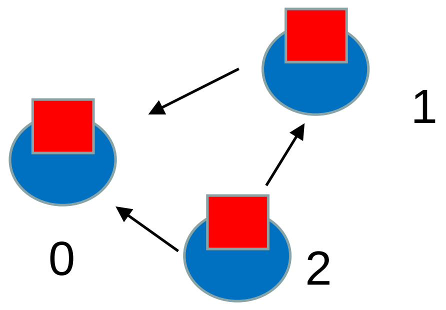
Send / Recv pair

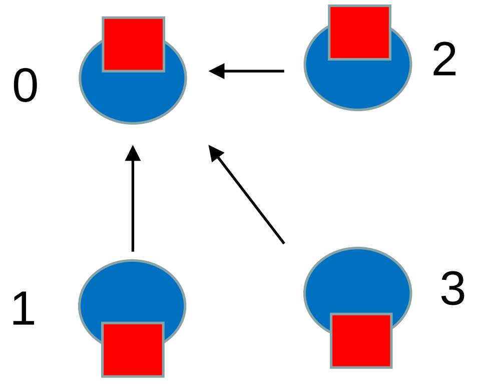
- MPI_Recv: to receive a message from another process

The syntax of MPI_Recv is:

MPI_Recv(void *received_data, int receive_count, MPI_Datatype receive_type,
int sender_ID, int tag, MPI_Comm comm, MPI_Status *status);

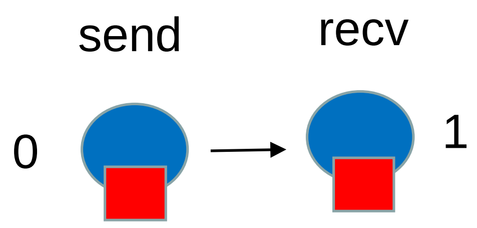
- received_data: variable of a C type that corresponds to the receive_type supplied below
- receive_count: number of data elements expected (int)
- receive_type: datatype of the data to be received (one of the MPI datatype handles)
- sender_ID: process ID of the sending process (int)
- tag: message tag (int)
- comm: communicator (handle)
- status: status struct (MPI_Status)





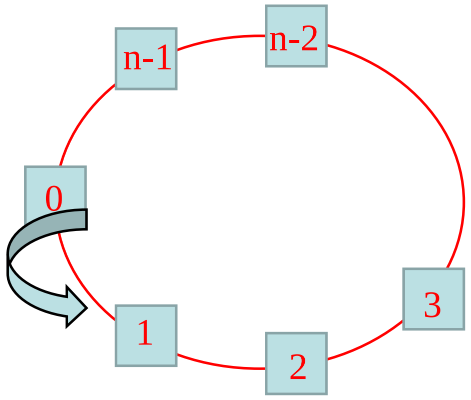
Example_1 – send_recv.c

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char* argv[]){
  MPI_Init(&argc,&argv);
  // Find out rank, size, set up a number
  int my_rank, num_procs, number = -1;
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
  if (my_rank == 0) {
    // If I am rank 0, send number to process 1
    MPI_Send(&number, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  } else if (my_rank == 1) {
    MPI_Recv(&number, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process 1 received number %d from process 0\n", number);
  }
  // We are assuming at least 2 processes for this task
  if (num_procs < 2) {
    fprintf(stderr, "World size must be greater than 1 for %s\n", argv[0]);
            MPI_Abort(MPI_COMM_WORLD, 1);
  }
  MPI_Finalize();
}
```



p0 send number to p1,
p1 recv it and print it out

Example_2 – ring_mpi.c



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

int main(int argc, char* argv[]) {
  MPI_Init(&argc,&argv);
  int my_rank, num_procs, token;
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
  /* Receive from the lower process and send to the higher process. Take care
   of the special case when you are the first process to prevent deadlock.*/
  if (my_rank != 0) {
    MPI_Recv(&token, 1, MPI_INT, my_rank-1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process %d received token %d from process %d\n", my_rank, token, my_rank-1);
  }
  else {
    token = -1;   //Set the token's value if you are process 0
  }
  MPI_Send(&token, 1, MPI_INT, (my_rank+1)%num_procs, 0, MPI_COMM_WORLD);

  /* Now process 0 can receive from the last process. */
  if (my_rank == 0) {
    MPI_Recv(&token, 1, MPI_INT, num_procs-1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    printf("Process %d received token %d from process %d\n", my_rank, token, num_procs-1);
  }
  MPI_Finalize();
}
```

```
mpicc -o ring_mpi ring_mpi.c

mpirun -np 2 ./ring_mpi
Process 0 received token -1 from process 1
Process 1 received token -1 from process 0

mpirun -np 4 ./ring_mpi
Process 1 received token -1 from process 0
Process 2 received token -1 from process 1
Process 3 received token -1 from process 2
Process 0 received token -1 from process 3

mpirun -np 8 ./ring_mpi
Process 1 received token -1 from process 0
Process 2 received token -1 from process 1
Process 3 received token -1 from process 2
Process 4 received token -1 from process 3
Process 5 received token -1 from process 4
Process 6 received token -1 from process 5
Process 7 received token -1 from process 6
Process 0 received token -1 from process 7
```

Example_3: Send/Recv pairs

```
#include "mpi.h"
#include <stdio.h>
main(int argc, char *argv[]) {
  int numtasks, rank, dest, source, rc, count, tag=1;
  char inmsg, outmsg=`hello`;
  MPI_Status Stat; // required variable for receive routines
  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  // task 0 sends to task 1 and waits to receive a return message
  if (rank == 0) {
  dest = 1; source = 1;
  MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat); }

// task 1 waits for task 0 message then returns a message
else if (rank == 1) {
  dest = 0; source = 0;
  MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD, &Stat);
  MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD); }

// query recieve Stat variable and print message details
  MPI_Get_count(&Stat, MPI_CHAR, &count);
  printf("Task %d: Received %d char(s) from task %d with tag %d \n",
              rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);

  MPI_Finalize();
}
```

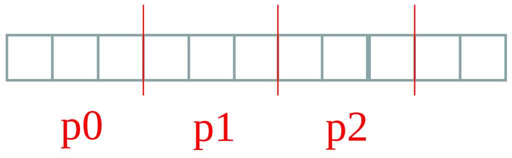
```
mpirun -np 2 ./send_recv_2
Task 0: Received 1 char(s) from task 1 with tag 1
Task 1: Received 1 char(s) from task 0 with tag 1
```

Example_4 – array_sum.c

```
#include <stdio.h>
#define max_rows 10000

   int array[max_rows];

   int main(int argc, char *argv[])
   {
      int i, num_rows;
      long int sum;
      printf("please enter the number of numbers to sum: ");
      scanf("%d", &num_rows);
      if(num_rows > max_rows) {
         printf("Too many numbers.\n");
         exit(1);
      }

      /* initialize an array */
      for(i = 0; i < num_rows; i++) {
         array[i] = i;
      }

       /* compute sum */
      sum = 0;
      for(i = 0; i < num_rows; i++) {
         sum += array[i];
      }
      printf("The grand total is: %ld\n", sum);
   }
```

Serial: one loop to sum

Parallelize – array_sum



MPI: each process does partial sum

Workload division

- each element is independent, sum of partial elements are independent as well
- work (sum) can be done separately by several independent processors

Implement approach

```
/* determine how many rows per process to sum. */
    avg_rows_per_process = num_rows/num_procs;

/* distribute a portion of the bector to each child process */
    for(an_id=1; an_id<num_procs; an_id++) {
      start_row=an_id*avg_rows_per_process;
      end_row=(an_id + 1)*avg_rows_per_process-1;

      num_rows_to_send = end_row-start_row + 1;
```

Example:

20/4 =5

an_id=1
start_row=5
end_row= 9

9-5+1 = 5

Data Send from Master

A common pattern of interaction among parallel processes is for one, the master, to allocate work to a set of slave processes and collect results from the slaves to synthesize a final result.

In this fragment, the master program sends a contiguous portion of array1 to each slave using MPI_Send. In practice, the master does not have to send an array; it could send a scalar or some other MPI data type, and it could construct array1 from any components to which it has access.

The master process will execute program statements like:

```
/* distribute portions of array1 to slaves. */
for(an_id = 1; an_id < num_procs; an_id++) {
   start_row = an_id*num_rows_per_process;
  MPI_Send(&num_rows_to_send, 1, MPI_INT, an_id, send_data_tag, MPI_COMM_WORLD);

  MPI_Send(&array1[start_row], num_rows_per_process, MPI_FLOAT, an_id, send_data_tag,
                MPI_COMM_WORLD);
}
```

Data Recv by slaves

The slave processes to work with the master would resemble:

```
/* Receive an array segment, here called array2 */

MPI_Recv( &num_rows_to_receive, 1 , MPI_INT, root_process, MPI_ANY_TAG,
                    MPI_COMM_WORLD, &status);

MPI_Recv( &array2, num_rows_to_receive, MPI_FLOAT, root_process,
                   MPI_ANY_TAG, MPI_COMM_WORLD, &status);
```

Workflow

The master process (my_id==0) will execute program statements like:

```
/* distribute portions of array1 to slaves. */
 MPI_Send ……

 /* and, then collect the results from the slave processes,
  * here in a variable called array2, and do something with them. */
for(an_id = 1 an_id < num_procs; an_id++) {
 MPI_Recv(&array2, num_rows_returned, MPI_FLOAT, MPI_ANY_SOURCE, MPI_ANY_TAG,
                MPI_COMM_WORLD, &status);

/* do something with its part*/
  ……
```

The slave processes (others) to work with:

```
/* Receive an array segment, here called array2 */
  MPI_Recv ……

/* Do its own partial calculation with array2 here
   ……

/* placing the result in partial_sum, and send partial_sum to the root process. */

MPI_Send( &partial_sum, num_rows_to_return, MPI_FLOAT, root_process,
                 return_data_tag, MPI_COMM_WORLD);
```

The slave processes to work with this master would resemble:

```
/* Receive an array segment, here called array2 */

MPI_Recv( &num_rows_to_receive, 1 , MPI_INT, root_process, MPI_ANY_TAG,
                   MPI_COMM_WORLD, &status);
MPI_Recv( &array2, num_rows_to_receive, MPI_FLOAT, root_process,
                  MPI_ANY_TAG, MPI_COMM_WORLD, &status);

/* Do something with array2 here, placing the result in partial_sum,
  * and send partial_sum to the root process. */

MPI_Send( &partial_sum, num_rows_to_return, MPI_FLOAT, root_process,
                  return_data_tag, MPI_COMM_WORLD);
```

Hands-on exercise 2 – parallel array sum in MPI

1) Run the examples

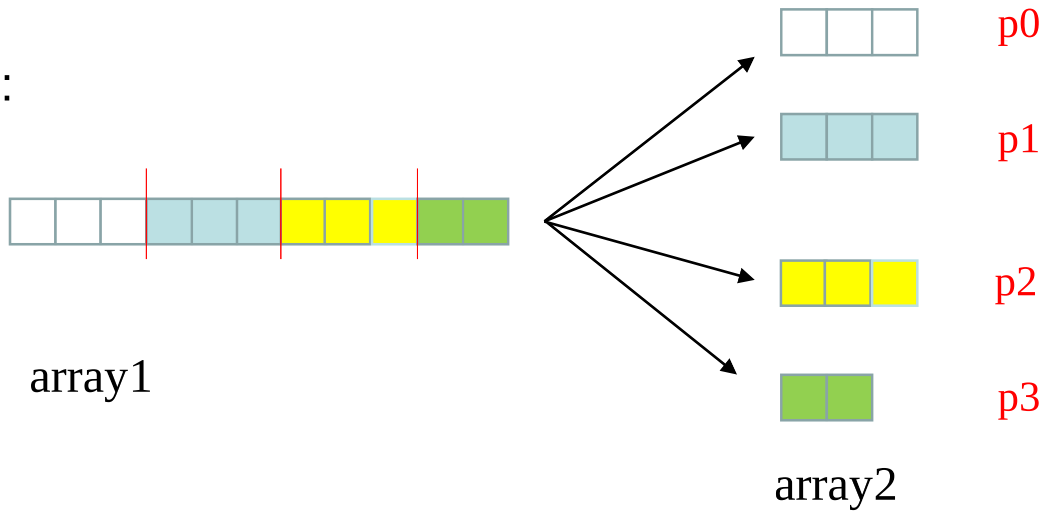
~/MPI_2017/codes/send_recv

2) work on:
array_sum_pseudo.c (for advance programmer)
or
array_sum_mpiAlmost.c (fill in a few lines)

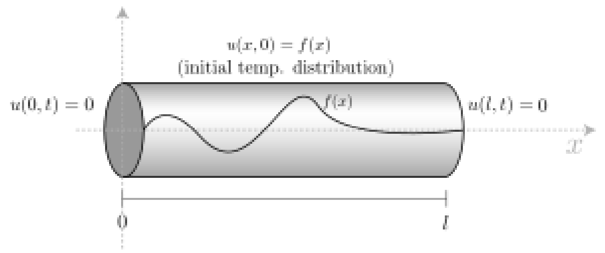
in
~/MPI_2017/codes/exercise

Before see the complete code:
array_sum_mpi.c

In
~/MPI_2017/codes/source



Example_5: 1d heat diffusion



- **Model 1d heat diffusion:**

  - diffusion of heat along an infinitely narrow pipe

- **Initial state:**
  - The whole pipe is at a stable and fixed temperature
  - At time 0, set both ends to different temperatures, which remain fixed throughout the computation

- **Calculate how temperature change in the rest of the pipe over time**
  - Expect the temperature will converge to a smooth gradient from one end of the pipe to the other

Heat Equation

- Mathematically, the one-dimensional heat conduction equation is

$$\frac{\partial U}{\partial t} = \kappa \frac{\partial^2 U}{\partial x^2}.$$

- Derivatives are approximated by finite differences

$$\partial u/\partial t \approx \frac{u_i^{k+1} - u_i^k}{\Delta t}$$

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i-1}^k - 2u_i^k + u_{i+1}^k}{\Delta x^2}$$

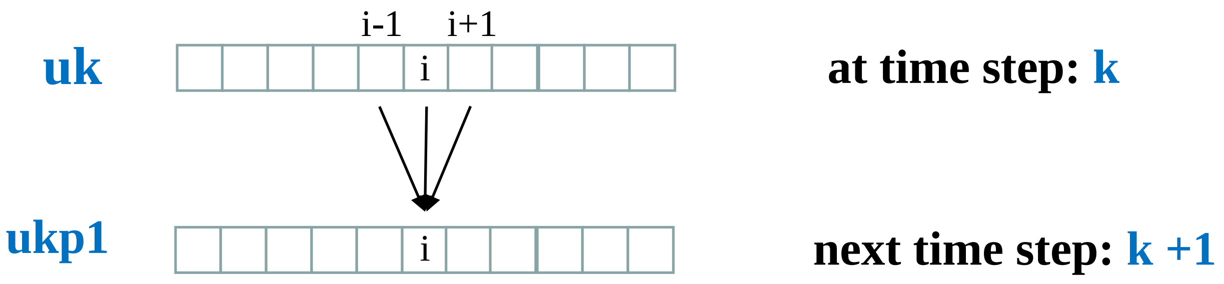
- An explicit numerical scheme for computing u^{k+1} based on u^k:

$$u_i^{k+1} = u_i^k + \frac{\Delta t}{\Delta x^2}(u_{i-1}^k - 2u_i^k + u_{i+1}^k)$$

Serial implementation

- Two data arrays: **uk** refers to u at timestep k, **ukp1** refers to u at timestep k+1

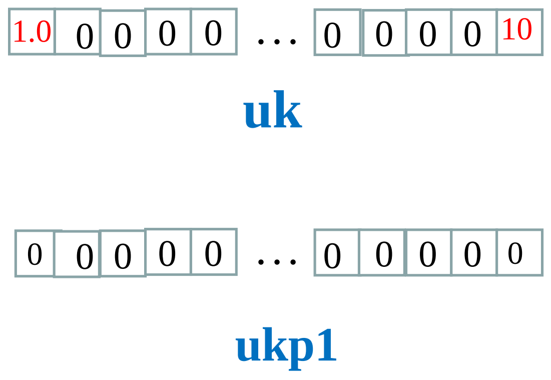
  ukp1[i] = **uk**[i] +(dt/dx*dx)(**uk**[i+1] – 2**uk**[i] + **uk**[i-1])



Initial condition

initialize(uk, ukp1, nx);

```
void initialize(double uk[], double ukp1[], int nx) {
    int i;
    uk[0] = LEFTVAL; uk[nx-1] = RIGHTVAL;
    for (i = 1; i < nx-1; ++i)
        uk[i] = 0.0;
    for (i = 0; i < nx; ++i)
        ukp1[i] = uk[i];
}
```



The main computation is a time-step process

When to stop?
- Use a converge threshold
- Iteration steps

```
for (step = 0; (step < maxsteps) && (maxdiff >= threshold); ++step) {
      double diff;
      /* compute new values */
      for (i = 1; i < nx-1; ++i) {
        ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
      }
      /* check for convergence */
      maxdiff = 0.0;
      for (i = 1; i < nx-1; ++i) {
        diff = fabs(uk[i] - ukp1[i]);
        if(diff > maxdiff)
          maxdiff = diff;
      }
     /* "copy" ukp1 to uk by swapping pointers */
     temp = ukp1; ukp1 = uk; uk = temp;
}
```

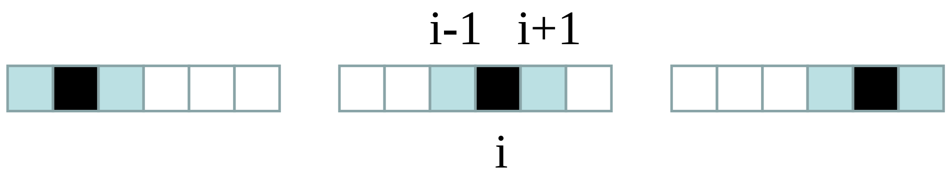
~/ss2017/MPI_2017/codes/exercise/ser_heat_1d.c

Parallelism

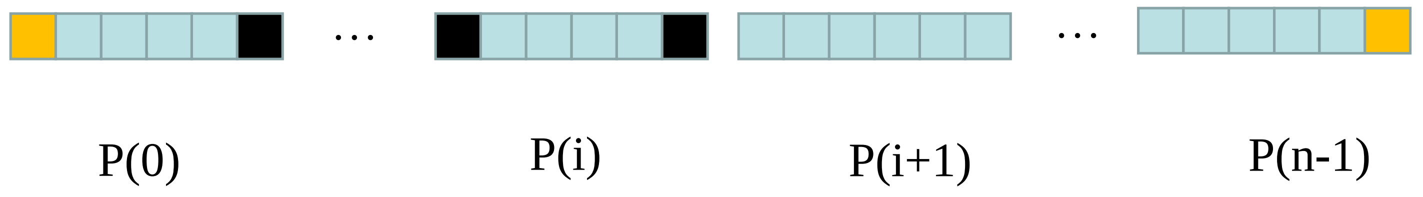
- **Work division**
  - Observation: ukp1[i] only depends on uk[i-1], uk[i], uk[i+1]
  - So, computations of ukp1[i] and ukp1[j] are independent of each other
  - We can divide the work of computing ukp1 among several processors

    numPoints per proc = totalPoints(nx)/numProcs

  - Calculate inner points using data within the chunk owned by the processor



- **Work division, data communication**
  - Most left and right points need data belongs to another processors
  - Two one-to-one data exchanges are needed on every processor per time step

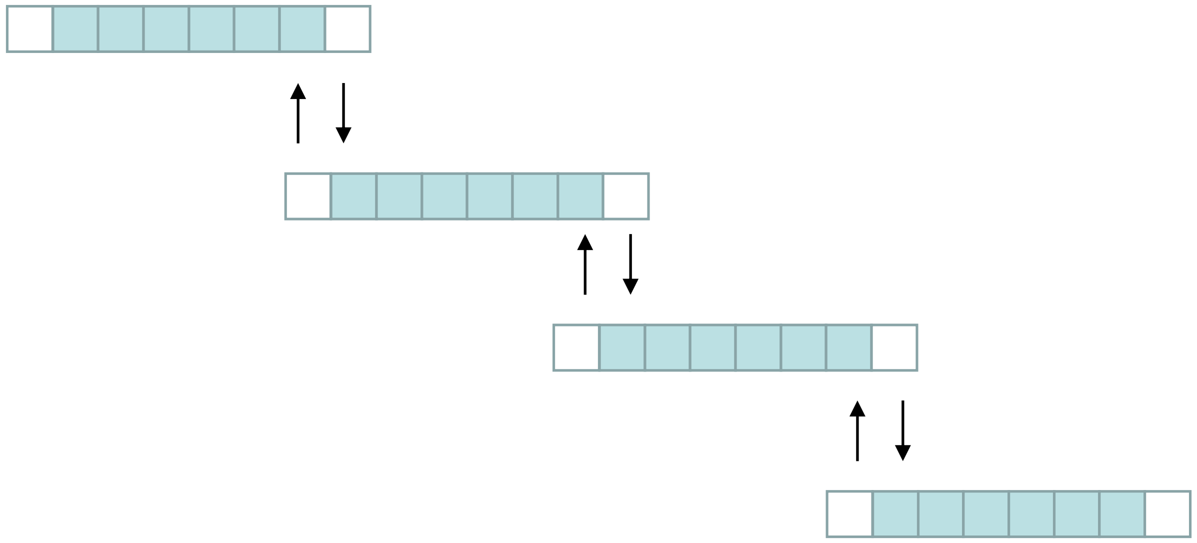


p0: rightmost point needs data from the leftmost point of p1

pi: leftmost point needs data from the rightmost point of p(i-1)
rightmost point needs data from the leftmost point of p(i+1)
…

p(n-1): leftmost point needs data from the rightmost point of p(n-2)

Data structure, use of ghost cells



Define “neighbors”

```
/* leftNbr, rightNbr, assume evenly divide */

leftNbr = myID - 1; /* ID of left "neighbor" process */

rightNbr = myID + 1; /* ID of right "neighbor" process */

numPoints = (nx / numProcs);


/* uk, ukp1 include a "ghost cell" at each end */

uk = malloc(sizeof(double) * (numPoints+2));

ukp1 = malloc(sizeof(double) * (numPoints+2));
```

Initialization per processor

```
/* uk, ukp1 include "ghost cell" at each end */
void initialize(double uk[], double ukp1[], int numPoints,
        int numProcs, int myID) {
    int i;
    for (i = 1; i <= numPoints; ++i)
        uk[i] = 0.0;

    /* left endpoint */
    if (myID == 0)
        uk[1] = LEFTVAL;

    /* right endpoint */
    if (myID == numProcs-1)
        uk[numPoints] = RIGHTVAL;

    /* copy values to ukp1 */
    for (i = 1; i <= numPoints; ++i)
        ukp1[i] = uk[i];
}
```

Exchange boundary information

```
if (myID != 0)
 MPI_Send(&uk[1], 1, MPI_DOUBLE, leftNbr, 0, MPI_COMM_WORLD);

if (myID != numProcs-1)
 MPI_Send(&uk[numPoints], 1, MPI_DOUBLE, rightNbr, 0, MPI_COMM_WORLD);

if (myID != 0)
 MPI_Recv(&uk[0], 1, MPI_DOUBLE, leftNbr, 0, MPI_COMM_WORLD, &status);

if (myID != numProcs-1)
MPI_Recv(&uk[numPoints+1],1, MPI_DOUBLE, rightNbr, 0, MPI_COMM_WORLD,
&status);
```

Compute new values

```
/* compute new values for interior points */
 for (i = 2; i < numPoints; ++i) {
        ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
 }

 /* compute new values for boundary points
  * (no real need to do these separately, but it would allow us to
  * understand the concept better) */

 if (myID != 0) {
         int i=1;
         ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
 }

 if (myID != numProcs-1) {
         int i=numPoints;
         ukp1[i]=uk[i]+ (dt/(dx*dx))*(uk[i+1]-2*uk[i]+uk[i-1]);
 }
```

Check for convergence, data swapping

```
/* check for convergence */
maxdiff_local = 0.0;

for (i = 1; i <= numPoints; ++i) {
    diff = fabs(uk[i] - ukp1[i]);
    if(diff > maxdiff_local)
    maxdiff_local = diff;
}

/* collect diff */
MPI_Allreduce(&maxdiff_local, &maxdiff, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);

/* "copy" ukp1 to uk by swapping pointers */
     temp = ukp1; ukp1 = uk; uk = temp;
```

Hands-on exercise 3
– implement 1d heat diffusion in MPI?

~/ss2017/MPI_2017/codes/exercise

ser_heat_1d.c

mpi_heat_1d_pseudo.c

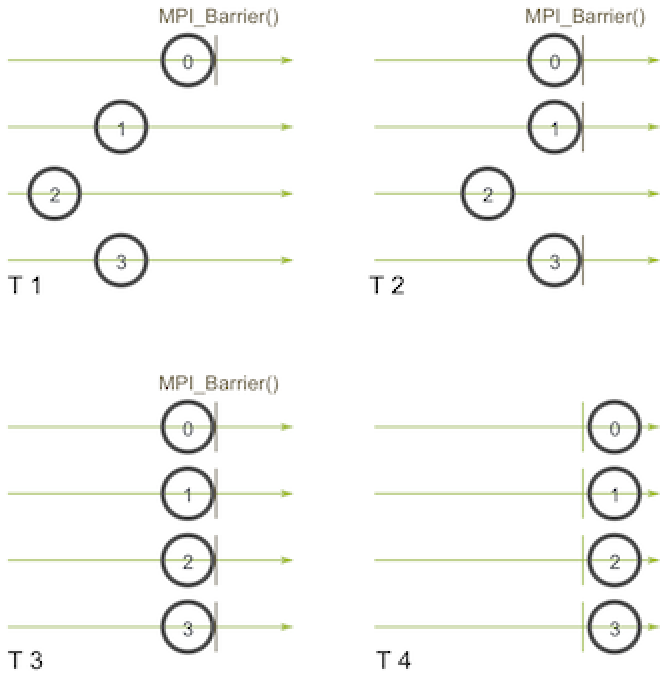
Collective communication

Collective communication is a method of communication which involves participation of **all** processes in a communicator.

One of the things to remember about collective communication is that it implies a *synchronization point* among processes. This means that all processes must reach a point in their code before they can all begin executing again.

MPI has a special function that is dedicated to synchronizing processes:

MPI_Barrier(MPI_Comm communicator)



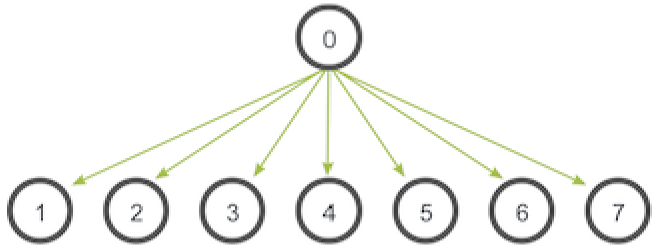
Broadcasting with MPI_Bcast

A *broadcast* is one of the standard collective communication techniques.

During a broadcast, one process sends the same data to all processes in a communicator.

One of the main uses of broadcasting is to send out user input to a parallel program, or send out configuration parameters to all processes.

The communication pattern of a broadcast looks like this:



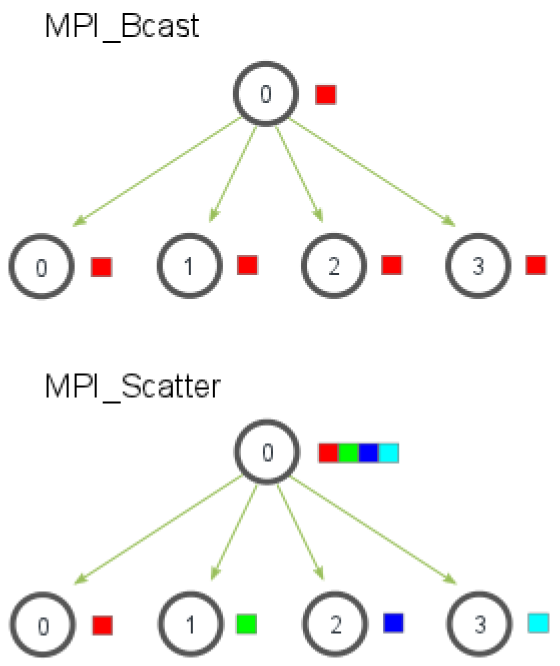
```
MPI_Bcast(
  void* data,
  int count,
  MPI_Datatype datatype,
  int root,
  MPI_Comm communicator
)
```

MPI_Scatter

MPI_Scatter involves a designated root process sending data to all processes in a communicator.

The primary difference between MPI_Bcast and MPI_Scatter is small but important. MPI_Bcast sends the *same* piece of data to all processes

while
MPI_Scatter sends *chunks of an array* to different processes.



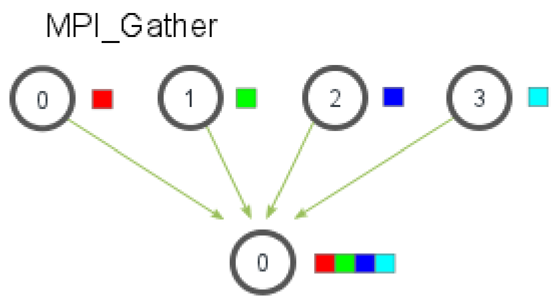
```
MPI_Scatter (&sendbuf,sendcnt,sendtype,&recvbuf,
             recvcnt,recvtype,root,comm)
```

MPI_Gather

MPI_Gather is the inverse of MPI_Scatter.

Instead of spreading elements from one process to many processes, MPI_Gather takes elements from many processes and gathers them to one single process.

This routine is highly useful to many parallel algorithms, such as parallel sorting and searching.



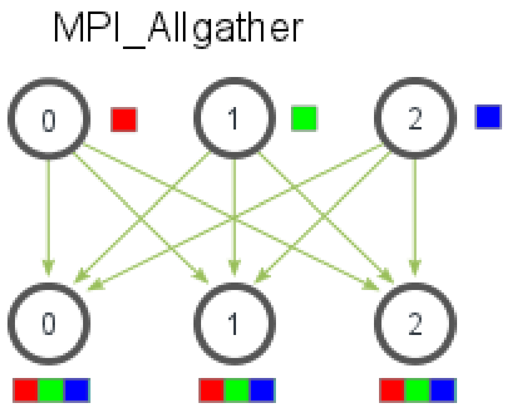
```
MPI_Gather (&sendbuf,sendcnt,sendtype,&recvbuf,
            recvcnt,recvtype,root,comm)
```

MPI_Allgather

Given a set of elements distributed across all processes

MPI_Allgather will gather all of the elements to all the processes.

In the most basic sense, MPI_Allgather is an MPI_Gather followed by an MPI_Bcast.



many-to-many communication pattern

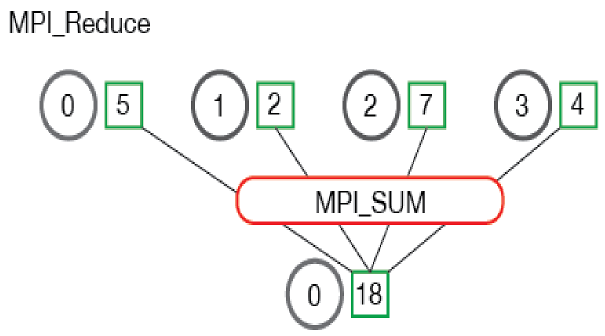
```
MPI_Allgather (&sendbuf,sendcnt,sendtype,&recvbuf,
               recvcnt,recvtype,root,comm)
```

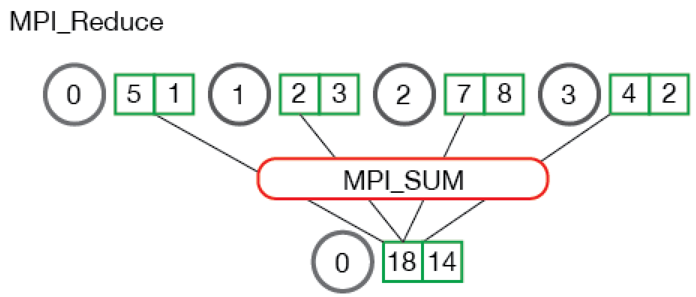
MPI Scatter/Gather: Average numbers in an array

```
if (my_rank == 0) {
    rand_nums = create_rand_nums(elements_per_proc * num_procs);
}
// Create a buffer that will hold a subset of the random numbers
float *sub_rand_nums = malloc(sizeof(float) * elements_per_proc);
// Scatter the random numbers to all processes
MPI_Scatter(rand_nums, elements_per_proc, MPI_FLOAT, sub_rand_nums, elements_per_proc, MPI_FLOAT, 0,
MPI_COMM_WORLD);

// Compute the average of your subset
float sub_avg = compute_avg(sub_rand_nums, elements_per_proc);
// Gather all partial averages down to the root process
float *sub_avgs = NULL;


if (my_rank == 0) {
    sub_avgs = malloc(sizeof(float) * num_procs);
}
MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
// Compute the total average of all numbers.
if (my_rank == 0) {
    float avg = compute_avg(sub_avgs, num_procs);
}
```

MPI_Reduce

Similar to MPI_Gather, MPI_Reduce takes an array of input elements on each process and returns an array of output elements to the root process.

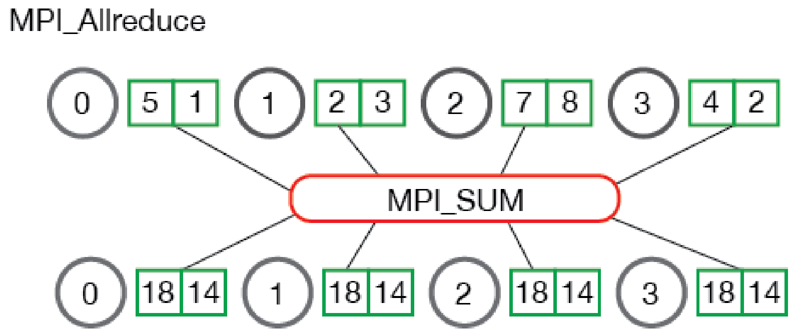




MPI_Allreduce

Many parallel applications will require accessing the reduced results across all processes rather than the root process

MPI_Allreduce is identical to MPI_Reduce with the exception that it does not need a root process id (since the results are distributed to all processes).



```
MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)
```

Example: Calculating π

- **Numerical integration**

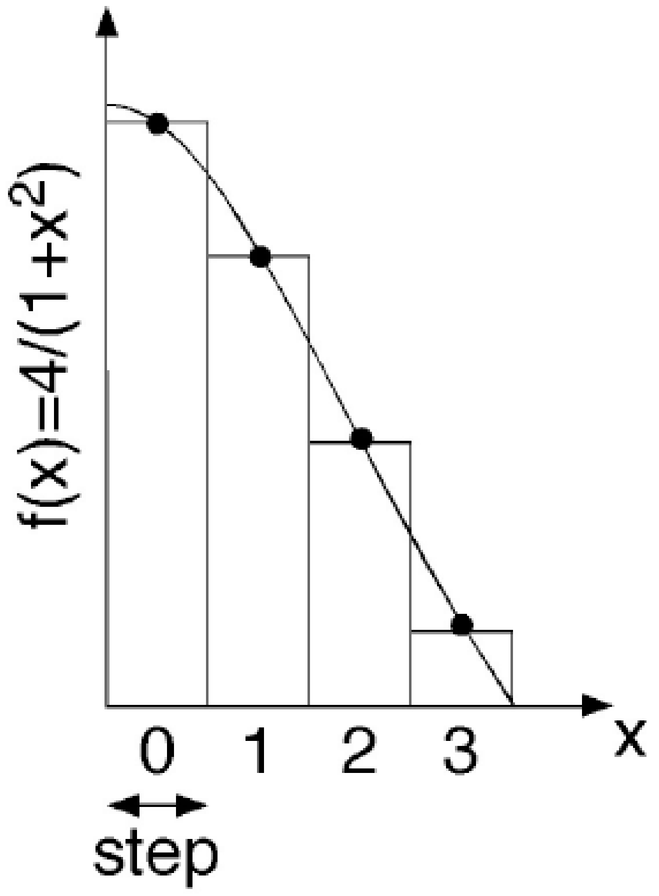
$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

- **Discretization:**

  $\Delta = 1/N$: `step = 1/NBIN`
  $x_i = (i+0.5)\Delta \; (i = 0,\ldots,N\text{-}1)$

$$\sum_{i=0}^{N-1} \frac{4}{1+x_i^2}\Delta \cong \pi$$

```
#include <stdio.h>
#define NBIN 100000
void main() {
  int i; double step,x,sum=0.0,pi;
  step = 1.0/NBIN;
  for (i=0; i<NBIN; i++) {
    x = (i+0.5)*step;
    sum += 4.0/(1.0+x*x);}
  pi = sum*step;
  printf("PI = %f\n",pi);
}
```



pi-serial.c

```c
/* compute pi by approximating the area under the curve f(x) = 4 / (1 + x*x)
 * between 0 and 1.
 *
 */
#include <stdio.h>
#define NBIN 100000

int main() {
	int i;
	double step,x,sum=0.0,pi;

	step = 1.0/NBIN;

	for (i=0; i<NBIN; i++) {
		x = (i+0.5)*step;
		sum += 4.0/(1.0+x*x);
	}
	pi = sum*step;
	printf("PI = %f\n",pi);
}
```

Hands-on exercise 4
– calculate pi using MPI collective communications?

Work on pi_mpiAlmost.c
in
~/MPI_2017/codes/exercise

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        sum += 4.0 / (1.0 + x*x);
    }
    mypi = h * sum;

MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
MPI_COMM_WORLD);
```

MPI datatypes

MPI datatype	C equivalent
MPI_SHORT	short int
MPI_INT	int
MPI_LONG	long int
MPI_LONG_LONG	long long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_BYTE	char

References

Online tutorials:

https://computing.llnl.gov/tutorials/mpi/
http://mpitutorial.com/tutorials/
http://condor.cc.ku.edu/~grobe/docs/intro-MPI-C.shtml

https://agenda.infn.it/getFile.py/access?contribId=3&sessionId=0&resId=0&materialId=slides&confId=9446

http://people.sc.fsu.edu/~jburkardt/c_src/mpi/mpi.html

http://www.uio.no/studier/emner/matnat/ifi/INF3380/v11/undervisningsmateriale/inf3380-week11.pdf

http://web.mst.edu/~ercal/387/MPI/cpi.c

http://www.new-npac.org/projects/cdroms/cewes-1999-06-vol2/cps615course/mpi-c.html