PHYSM0032 Advanced Computational Physics
Lectures 5 & 6

Dr. Simon Hanna

October 12, 2021

7 Distributed memory programming using the message passing interface (MPI)

7.1 Introduction

- For large parallel systems, needing either:
  - more memory than on one node, or
  - more cpu power than on one node.
- Typically, use the SPMD model: single program, multiple data.
- Each processor runs the same program, but applies it to different data.
- The data might be different parts of the same array.
- Communication is arranged explicitly i.e. processors will broadcast/receive data to/from specified other nodes.
- Communication times need to be factored into algorithm design.
- Most commonly used method is the Message Passing Interface (MPI). This is a "standard" and there are many available implementations.
- Readily available interfaces include OpenMPI and MPICH.
- Python documentation (mpi4py) available at: *https://mpi4py.readthedocs.io/en/stable/tutorial.html*

7.2 Using MPI

- You will first need to install a version of MPI. OpenMPI is easy to obtain and works with gcc under Linux and on the Mac. MS-MPI is needed on Windows systems with MSVC (see installation notes).
- Don't confuse MPI or Open-MPI with OpenMP – they are different.
- If using Python/Cython, you need the mpi4py package which is available under Anaconda, but has to be installed (see Installation notes on Blackboard).
- See the Installation notes for information on compiling C or Fortran programs with MPI.
- Under MPI, programs are executed using, for example:

```
mpiexec -np 4 ./my_prog
mpirun -np 4 ./my_prog
```

- The command name (mpiexec or mpirun) varies with the version of MPI used and the operating system.
- The -np flag (sometimes -n) specifies the number of independent processes that will run, potentially on separate machines.
- To run a python program with mpi4py, use a command such as:

```
mpiexec -np 4 python my_prog.py
```

7.3 A simple example MPI program

```
—————————C/C++—————————
#include <stdio.h>
#include <mpi.h>
int main(){
int rank, size, length;
char name[MPI_MAX_PROCESSOR_NAME];
MPI_Init(NULL,NULL);
MPI_Comm_size(MPI_COMM_WORLD,&size);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Get_processor_name(name, &length);
printf("Hello, from process (rank) %d of %d,
    running on processor %s\n",rank,size,name);
MPI_Finalize();
}
```

```
—————————Fortran—————————
program mpi_hello
include 'mpif.h'
integer rank, size, length, ierror
character*(MPI_MAX_PROCESSOR_NAME) name
call MPI_INIT(ierror)
call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
call MPI_GET_PROCESSOR_NAME(name, length, ierror)
write(*,"('Hello, from process (rank)',I3,' of',
    I3, ' running on processor ',A)")rank,size,
    name(1:length)
```

```
 call MPI_FINALIZE(ierror)
end
```

```
---------------Python---------------
from mpi4py import MPI
rank = MPI.COMM_WORLD.Get_rank()
size = MPI.COMM_WORLD.Get_size()
name = MPI.Get_processor_name()
print("Hello,from process (rank) {:d} of {:d},
    running on processor {:s}".format(rank, size,
    name))
```

MPI_COMM_WORLD is a *communicator* that defines the group of processes that can speak to each other. In this case it is the *world* group i.e. *all* processes.

- All processes are essentially equal, but 0 is often regarded as the *master* process that can pass data, flags etc out to all other processes.
- The same code is executed on all processes – the *rank* is used to determine which processes communicate with each other, and what each of them actually do. In the above case, they all do the same thing.

7.4 More complex MPI example

7.4.1 C version

```
//---------------MPI Prime Number Counter
    ---------------
# include <math.h>
# include <mpi.h>
# include <stdio.h>
# include <stdlib.h>
# include <time.h>

int prime_number ( int n, int id, int p ){
  int i, j, prime, total=0;
  for ( i = 2 + id; i <= n; i = i + p ){
    prime = 1;
    for ( j = 2; j < i; j++ )   // main work
        function
      if ( ( i % j ) == 0 ){    // tackles
          different
        prime = 0;              // range for each
        break; }                // task rank
    total += prime; }
  return total; }

int main ( int argc, char *argv[] ){
  int i, id, ierr, n, n_factor=2, p;
  int n_lo=1, n_hi=65536, primes, primes_part;
  double wtime;          // Get size of world group
      and
  ierr = MPI_Init ( &argc, &argv );   // current
      rank
  ierr = MPI_Comm_size ( MPI_COMM_WORLD, &p );
  ierr = MPI_Comm_rank ( MPI_COMM_WORLD, &id );
```

```
  if ( id == 0 ){ // only master rank (0) does
      this
    printf ( "PRIME_MPI\n" );
    printf ( "  C/MPI version\n" );
    printf ( "  MPI  program to count primes.\n"
        );
    printf ( "  Number of processes is %d\n", p )
        ;
    printf ( "         N      Primes       Time\n"
        );
  }
  n = n_lo;
  while ( n <= n_hi ){
    if ( id == 0 ) wtime = MPI_Wtime ( );//MPI
        timer
    ierr = MPI_Bcast ( &n, 1, MPI_INT, 0,
        MPI_COMM_WORLD );// broadcast n to ranks
    primes_part = prime_number ( n, id, p );
    ierr = MPI_Reduce ( &primes_part, &primes, 1,
        MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD );
    if ( id == 0 ) { // reduction to count primes
      wtime = MPI_Wtime ( ) - wtime;
      printf ( "  %8d  %8d  %14f\n", n, primes,
          wtime ); }
    n = n * n_factor; }
  ierr = MPI_Finalize ( );
  if ( id == 0 ) {
    printf ( "\n PRIME_MPI - Master process:\n");
    printf ( "  Normal end of execution.\n\n"); }
  return 0; }
```

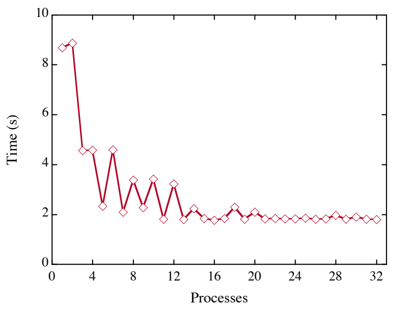
7.4.2 Python version

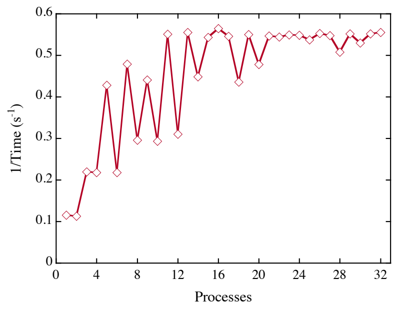
```
#---------------MPI Prime Number Counter
    ---------------
from mpi4py import MPI

def prime_number ( n, id, p ): # main work
    function
    total=0                    # tackles
        different
    for i in range(2+id, n+1, p): # range for
        each
        prime = 1              # task rank
        for j in range(2, i, 1):
            if ( ( i % j ) == 0 ):
                prime = 0
                break
        total += prime
    return total

n_factor=2
n_lo=1
n_hi=65536
primes=0
comm = MPI.COMM_WORLD # communicator for world
    group
p = comm.Get_size()   # number of tasks
id = comm.Get_rank()  # rank for this task
```

```
if ( id == 0 ):  # only master rank (0) does this
    print ( "PRIME_MPI:" )
    print ( "  Python/mpi4py version" )
    print ( "  MPI program to count primes" )
    print ( "  Number of processes is", p )
    print ( "    N   Primes     Time" )

n = n_lo;
while ( n <= n_hi ):
    if ( id == 0 ):
        wtime = MPI.Wtime()  # MPI timing routine
    comm.bcast(n, root=0)    # broadcast n to
        ranks
    primes_part = prime_number ( n, id, p )
    primes = comm.reduce (primes_part, None, root
        =0)
    if ( id == 0 ):  # reduction to count primes
        wtime = MPI.Wtime ( ) - wtime
        print ( "{:5d}  {:5d}   {:10.8f}  ".
            format(n, primes, wtime) )
    n = n * n_factor
if ( id == 0 ):
    print (" ")
    print ("PRIME_MPI:")
    print ("  Normal end of execution.")
```

7.4.3 Example timings

Example timings from the prime number program shown in previous section, spawning up to 32 processes, on a 4 core laptop:





This was with the C version, with `n_hi=262144`. The Python/mpi4pi version is somewhat slower but illustrates the principle.

8 A closer look at MPI

8.1 Introduction

The following notes were originally derived from the online tutorial at Lawrence Livermore National Laboratories:

https://computing.llnl.gov/tutorials/mpi/

More details on mpi4py can be found at:

https://mpi4py.readthedocs.io/en/stable/tutorial.html

8.2 A very basic program revisited

We start with the following "hello world" type of program and explain the various parts:

```
//——————————————C/C++——————————————
#include <stdio.h>
#include <mpi.h>
int main(){
    int rank, size, length;
    char name[MPI_MAX_PROCESSOR_NAME];
    MPI_Init(NULL,NULL);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Get_processor_name(name, &length);
    printf("Hello, from process (rank) %d of %d,
    running on processor %s\n",rank,size,name);
    MPI_Finalize(); }
```

```
# ——————————————Python——————————————
from mpi4py import MPI
rank = MPI.COMM_WORLD.Get_rank()
size = MPI.COMM_WORLD.Get_size()
name = MPI.Get_processor_name()
print("Hello,from process (rank) {:d} of {:d},
running on processor {:s}".format(rank, size,
    name))
# ————————————————————————————————
```

import mpi4py / #include <mpi.h> (mpif.h for Fortran):
File/module including all MPI function prototypes and data types.

MPI_Init(): MPI environment call. All C or Fortran programs must begin with this. *Not required in Python.*

MPI_MAX_PROCESSOR_NAME: Length of string for holding processor name. *Not needed in Python.*

MPI.COMM_WORLD / MPI_COMM_WORLD: A *communicator* that defines the set of all processes (tasks) that may communicate (see below). It is possible to define smaller groups if you need to partition your work. *In Python this is an object.*

size: the total number of processes / tasks in the group;

rank: the unique number for the current task.

Note: in C, all MPI routines take the format:

```
rc = MPI_Xxxxx([parameters]);
```

where `rc` is a return code that may rarely be of use.

In Python, all MPI functions take the format:

```
object = MPI.COMM_WORLD.[X/x]xxxx([parameters])
```

where `object` will be a useful parameter or control structure.

8.3 Communicators, Groups, Rank and Size

- MPI uses communicators and groups to define which collection of tasks may communicate with each other. The number of such tasks is known as the ***size***.
- Most MPI routines require a communicator to be specified.
- For now, use ***MPI.COMM_WORLD*** / ***MPI_COMM_WORLD*** whenever a communicator is required – it is predefined to include all MPI tasks.
- Within communicator, each task has a unique, integer ***rank*** assigned when the process initializes. Rank is sometimes also called "task ID". Ranks are contiguous from zero to `size-1`.
- Rank is used to specify source and destination of messages. Often used conditionally by the application to control program execution (`if rank==0` *do this* / `if rank==1` *do that*).

8.4 Environment Management Routines

Used for interrogating and setting the MPI execution environment. Most of the commonly used ones are described below. Details of calling these functions can be found in the documents linked above.

MPI_Init: Initializes the MPI execution environment. *Must be called in every MPI program,* ***before*** any other MPI functions, and must be called only ***once***. May be used to pass command line arguments to all processes. i.e. `MPI_Init(&argc,&argv);`. *MPI_Init() is called automatically on importing mpi4py and should not be needed in a Python program.*

MPI.COMM_WORLD.Get_size / MPI_Comm_size: Returns total number of MPI processes in specified communicator. If communicator is MPI_COMM_WORLD, it represents the total number of MPI tasks available to the application.

MPI.COMM_WORLD.Get_rank / MPI_Comm_rank: Returns rank (task ID) of calling MPI process within the specified communicator (e.g. between `0` and `size-1` within MPI_COMM_WORLD). A process associated with multiple communicators will have a unique rank within each.

MPI.COMM_WORLD.Abort / MPI_Abort: Terminates all MPI processes associated with communicator. Often actually terminates ALL processes regardless of the communicator specified.

MPI.Get_processor_name / MPI_Get_processor_name: Returns processor name. (In C/Fortran also returns number of characters – buffer for `name` should be at least MPI_MAX_PROCESSOR_NAME characters).

MPI.Get_version / MPI_Get_version: Returns the version of MPI standard implemented by the library.

MPI.Is_initialized / MPI_Initialized: Indicates whether `mpi4py` has been successfully loaded or whether MPI_Init has been successfully called. Returns integer as true (1) or false(0). In C/Fortran use MPI_Initialized to avoid calling MPI_Init more than once.

MPI.Wtime / MPI_Wtime: Returns elapsed wall-clock time in seconds (double precision) on calling processor. ***Always*** use this function when timing MPI code.

MPI.Wtick / MPI_Wtick: Returns resolution in seconds (double precision) of MPI_Wtime. Mainly for information.

MPI_Finalize: Terminates MPI execution environment. Should be the last MPI routine called in every MPI program. Not needed in Python.

Note: Routines MPI_init() and MPI_Finalize() are required in all C/C++ and Fortran programmes. These are called implicitly by `mpi4py` and should not be used in a Python program.

8.5 Communication

8.5.1 Introduction

- The need for communications between tasks depends upon your problem:
- Some problems can be decomposed and executed in parallel with virtually no need for tasks to share data. The data can easily be distributed to multiple tasks that then act independently of each other to do their portion of the work.

- These problems are often called embarrassingly parallel because they are so straight-forward. Very little inter-task communication is required. e.g. in the Pi calculation, each process could take a portion of the work, and they can simply combine results at the end.
- Most parallel applications are not so simple, and require tasks to share data with each other. E.g. a 3-D heat diffusion problem requires each task to know the temperatures calculated by the tasks handling neighbouring data.
- There are several factors to consider when designing your program's inter-task communications (see below)

8.5.2 Cost of communications

- Inter-task communication virtually always implies overhead.
- Resources that could be used for computation are instead used to handle data.
- Communications frequently require synchronization between tasks, resulting in tasks *waiting* instead of doing work.
- Competing communication traffic can saturate the available network bandwidth.
- Latency vs. Bandwidth:
  - *Latency* is the time it takes to send a minimal (0 byte) message from point A to point B.
  - *Bandwidth* is the amount of data that can be communicated per unit of time.

  Sending many small messages can cause latency to dominate. Often more efficient to package small messages into a single larger message, thus increasing the effective bandwidth.

8.5.3 Scope of communications

- Knowing which tasks must communicate with each other is critical when designing parallel code. Both of the scopings described below can be implemented synchronously or asynchronously.
- *Point-to-point* – involves two tasks, with one task acting as the sender of data, and the other acting as the receiver.
- *Collective* – involves data sharing between more than two tasks, often specified as being members in a common group. Some common variations (there are more):

  broadcast: One task sends same data to many tasks.

  scatter: One task sends different data to each of many tasks.

  gather: One task receives different data from each of many.

  reduction: One task receives partial results from each of many tasks, which are combined (reduced) into a single result.

8.5.4 Synchronous vs. asynchronous communications

- *Synchronous* communications require some type of *handshaking* between tasks that are sharing data.
- Synchronous communications often referred to as ***blocking*** communications since *other work must wait* until the communications have completed.
- *Asynchronous* communications allow tasks to transfer data independently. E.g. task 1 can prepare and send a message to task 2, and then immediately begin doing other work. When task 2 actually receives the data doesn't matter.
- Asynchronous communications often referred to as ***non-blocking*** since other work can be done while the communications are taking place.
- Interleaving computation with communication is the single greatest benefit of using asynchronous communications; however, use of asynchronous calls often requires status checks to verify when transfers are complete.
- *Efficiency of communications:* Generally use of asynchronous communication operations can improve overall program performance, but note the added requirement for periodic synchronisations e.g. at the end of a time-step in a simulation.

8.6 Point-to-point communication

8.6.1 Blocking Message Passing Routines

The more commonly used MPI blocking message passing routines are described:

MPI.COMM_WORLD.Send() or .send() / MPI_Send:
Basic blocking send operation. Routine returns only after the application buffer in the sending task is free for reuse. In `mpi4py`, `.Send()` version is used for sending NumPy arrays, and `.send()` is used for sending Python objects.

MPI.COMM_WORLD.Recv() or .recv() / MPI_Recv:
Receive a message and block until the requested data is available in the application buffer in the receiving task. In `mpi4py`, `.Recv()` version is used for receiving NumPy arrays, and `.recv()` version for receiving Python objects.

MPI_Get_count: Returns the source, tag and number of elements of datatype received. Can be used with both blocking and non-blocking receive operations. The source and tag are returned in the status structure as `status.MPI_SOURCE` and `status.MPI_TAG`. There appears to be no `mpi4py` equivalent.

8.6.2 Blocking example

```
// mpi_pingpong.c
// Simple blocking message passing demo.
// Play ping-pong between ranks 0 and 1.
// To run: mpiexec -np 2 ./mpi_pingpong
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

int main(){
  int size, rank;
  MPI_Status Stat; // required for receive
      routines
  MPI_Init(NULL,NULL);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  if (size != 2) {
    printf("Ping-pong only works with 2 ranks");
    MPI_Finalize();
    return 1; }
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  const int PING=0, PONG=1;
  int MESSAGE_ID = 5; // arbitrary unique choice
  int LENGTH = 1; // length of message (1 integer
      )
  int counter = 0;
```

```
  for (int i=0; i<10; i++) { // PING sends to
      PONG
    if (rank == PING) {       // and awaits reply
      printf("PING sending %d to PONG\n", counter
          );
      MPI_Send(&counter, LENGTH, MPI_INT, PONG,
          MESSAGE_ID, MPI_COMM_WORLD);
      MPI_Recv(&counter, LENGTH, MPI_INT, PONG,
          MESSAGE_ID, MPI_COMM_WORLD, &Stat);
      printf("PING receiving %d from PONG\n",
          counter);
    }
    // PONG waits for PING's message then replies
    else if (rank == PONG) {
      MPI_Recv(&counter, LENGTH, MPI_INT, PING,
          MESSAGE_ID, MPI_COMM_WORLD, &Stat);
      printf("PONG receiving %d from PING and
          incrementing\n", counter);
      counter++;
      printf("PONG sending %d to PING\n", counter
          );
      MPI_Send(&counter, LENGTH, MPI_INT, PING,
          MESSAGE_ID, MPI_COMM_WORLD);
    }
  }
  if (rank == PING) printf("Total length of rally
      : %d times there and back\n", counter);
  MPI_Finalize();
}
```

```
# Simple blocking message passing demo.
# A message is passed back and forth between
# a pair of processes.
#
# Run by typing:
# mpirun -np 2 python mpi_pingpong.py
#

from mpi4py import MPI

size = MPI.COMM_WORLD.Get_size()
if size != 2:
    print("Ping-pong only works with 2 ranks")
    exit(1)

comm = MPI.COMM_WORLD
rank = MPI.COMM_WORLD.Get_rank()
#
# Play ping-pong between ranks 0 and 1
#
PING = 0
PONG = 1
MESSAGE_ID = 5 # arbitrary unique choice
counter = 0
```

```
for i in range(10):
    # PING sends to PONG and waits to receive a
        return message
    if rank == PING:
        print("PING sending %d to PONG" % counter
            )
        comm.send(counter, dest=PONG, tag=
            MESSAGE_ID)
        counter = comm.recv(source=PONG, tag=
            MESSAGE_ID)
        print("PING receiving %d from PONG" %
            counter)
    # PONG waits for PING's message then returns
        it, incremented
    elif rank == PONG:
        counter = comm.recv(source=PING, tag=
            MESSAGE_ID)
        print("PONG receiving %d from PING and
            incrementing" % counter)
        counter += 1
        print("PONG sending %d to PING" % counter
            )
        comm.send(counter, dest=PING, tag=
            MESSAGE_ID)
if rank == PING:
    print("Total length of rally: %d times there
        and back" % counter)
```

8.6.3 Non-blocking Message Passing Routines

The more commonly used MPI non-blocking message passing routines are described.

MPI.COMM_WORLD.I(i)send / MPI_Isend: Identifies an area in memory (e.g. an array) to serve as a send buffer. Processing continues without waiting for the message to be copied out from the buffer. A communication request object is returned for handling the message status. The program should not modify the buffer until status checks (in C/Fortran using MPI_Wait or MPI_Test) indicate that the non-blocking send has completed.

MPI.COMM_WORLD.I(i)recv / MPI_Irecv: Identifies an area in memory (e.g. an array) to serve as a receive buffer. Processing continues without waiting for the message to be received and copied into the buffer. A communication request object is returned for handling the message status. The program must use calls to the `.wait()` method (Python) or MPI_Wait / MPI_Test (in C/Fortran) to determine when the requested message is available in the buffer.

.wait() methods / MPI_Wait??: These block until a specified non-blocking send or receive operation has completed. In C/Fortran, can specify any, all or some messages.

MPI_Test?? (C/Fortran only): Checks the status of a specified non-blocking send or receive. The flag returns 1 if the operation has completed, and 0 if not. Can choose any, all or some messages.

8.6.4 Non-blocking message example

```
// ———————————————C/C++———————————————
MPI_Isend (&buffer,count,datatype,dest,tag,comm,&
     request)
MPI_Irecv (&buffer,count,datatype,source,tag,comm
     ,&request)

MPI_Wait (&request,&status)
MPI_Waitany (count,&array_of_requests,&index,&
     status)
MPI_Waitall (count,&array_of_requests,&
     array_of_statuses)
MPI_Waitsome (incount,&array_of_requests,&
     outcount,
 &array_of_offsets, &array_of_statuses)

MPI_Test (&request,&flag,&status)
MPI_Testany (count,&array_of_requests,&index,&
     flag,&status)
MPI_Testall (count,&array_of_requests,&flag,&
     array_of_statuses)
MPI_Testsome (incount,&array_of_requests,&
     outcount,
 &array_of_offsets, &array_of_statuses)
// ————————————————————————————————————
```

```
# ———————————————Python———————————————
from mpi4py import MPI
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
```

```
# Non-blocking send
request = comm.isend(buffer, dest=1, tag=11)
request.wait()  # Returns when send is complete
# Non-blocking receive
request = comm.irecv(source=0, tag=11)
buffer = request.wait()  # Returns with data in
                         # buffer when receive
                         # is complete
# ——————————————————————————————————————
```

8.7 Collective Communication Routines

Collectives calls are blocking – only one can be active at one time. They are generally more efficient than point-to-point communications, because they use a tree-like structure for propagating the messages:

MPI.COMM_WORLD.Barrier / MPI_Barrier: Synchronization operation. Creates a barrier synchronization in a group. Each task, when reaching the MPI barrier call, blocks until all tasks in the group reach the same call. Then all are free to proceed.

MPI.COMM_WORLD.B(b)cast / MPI_Bcast: Data movement operation. Broadcasts (sends) a message from the process with rank *"root"* to all other processes in the group.

MPI.COMM_WORLD.S(s)catter / MPI_Scatter: Data movement operation. Distributes distinct messages from a single source task to each task in the group.

MPI.COMM_WORLD.G(g)ather / MPI_Gather: Data movement operation. Gathers distinct messages from each task in the group to a single destination task. This is the reverse of MPI...Scatter.

MPI.COMM_WORLD.A(a)llgather / MPI_Allgather: Data movement operation. Concatenation of data to all tasks in a group. Each task, in effect, performs a one-to-all broadcast operation.

MPI.COMM_WORLD.R(r)educe / MPI_Reduce: Collective computation operation. Applies a reduction operation on all tasks in the group and places the result in one task.

```
//
// mpi_scatter.c
//
// Simple demonstration of scatter method.
//
#include <mpi.h>
#include <stdio.h>
#define SIZE 4

int main(){
  int numtasks, rank, sendcount, recvcount,
      source;
  double sendbuf[SIZE][SIZE];
  double recvbuf[SIZE];
  MPI_Init(NULL,NULL);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
```

```
  if (numtasks == SIZE) {
    // define source and array to send/receive,
    // then perform collective scatter
    source = 2;
    sendcount = SIZE;
    recvcount = SIZE;
    // only the source task initialises sendbuf
    if (rank==source)
      for (int i=0; i<SIZE; i++)
        for (int j=0; j<SIZE; j++)
          sendbuf[i][j] = i*SIZE + j+1;
    // all ranks issue MPI_Scatter command
    MPI_Scatter(sendbuf,sendcount,MPI_DOUBLE,
        recvbuf,recvcount,MPI_DOUBLE,source,
        MPI_COMM_WORLD);
    printf("rank= %d  Results: %f %f %f %f\n",
        rank,recvbuf[0],recvbuf[1],recvbuf[2],
        recvbuf[3]);
    }
  else
    printf("Must specify %d processors.
        Terminating.\n",SIZE);
  MPI_Finalize();
  }
```

```
#
# mpi_scatter.py
#
# Simple demonstration of scatter method.
#
# Run by typing:
# mpirun -np 4 python mpi_scatter.py
#

from mpi4py import MPI
import numpy as np

SIZE=4
sendbuf=np.zeros((SIZE,SIZE))
recvbuf=np.zeros(SIZE)
comm = MPI.COMM_WORLD
rank = comm.Get_rank()
numtasks = comm.Get_size()
```

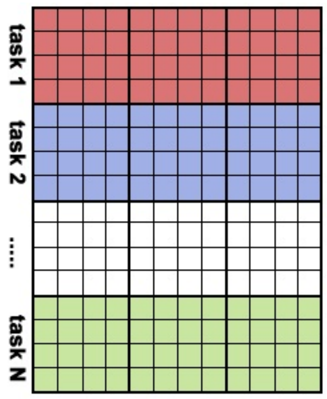
```
if numtasks==SIZE:
    # define source task and elements to send/
        receive,
    # then perform collective scatter
    source = 2
    # only the source task initialises sendbuf
    if rank==source:
        for i in range(SIZE):
            for j in range(SIZE):
                sendbuf[i,j] = i*SIZE + j+1
    # all ranks issue MPI Scatter command
    recvbuf=comm.scatter(sendbuf,root=source)
```

```
    print("rank= {:2d} Results: {:4.1f} {:4.1f}
        {:4.1f} {:4.1f}".
      format(rank,recvbuf[0],recvbuf[1],recvbuf
        [2],recvbuf[3]))
else:
    print("Must specify",SIZE," processors.
        Terminating.")
```

8.8 Decomposition examples

There follow a number of examples showing different ways that mathematical problems might be decomposed and shared between processes.

8.8.1 Array Processing Soln. 1: Direct allocation



- An example of data decomposition.
- Calculation of elements is independent – embarrassingly parallel solution; no communication or synchronization.
- Each process owns portion of the array (subarray).
- Distribution scheme is chosen for efficient memory access. In C or Python distribute by rows; in Fortran distribute by columns.
- Independent calculation so no communication or synchronization.
- No load balance concerns.
- Each task executes the portion of the loop corresponding to the data it owns.

```
for (i=mystart; i<myend; i++)
  for (j=0; j<n; j++)
    a[i,j] = fcn[i,j];
```

```fortran
DO i = mystart, myend
    DO j = 0, n
        a(j,i) = fcn(i,j)
    ENDDO
ENDDO
```

```python
for i in range(mystart, myend):
  for j in range(n):
      a[i,j] = fcn[i,j]
```

Possible Solution:

```
if I am MASTER
   initialize the array
   send each WORKER info on part of array it owns
   send each WORKER its portion of initial array
   receive results from each WORKER

if I am WORKER
   receive from MASTER info on part of array I own
   receive from MASTER my portion of initial array
   calculate my portion of array
   send results to MASTER
```

See program mpi_array.py / mpi_array.c / mpi_array.for for example.

8.8.2 Array Processing Solution 2: Pool of Tasks

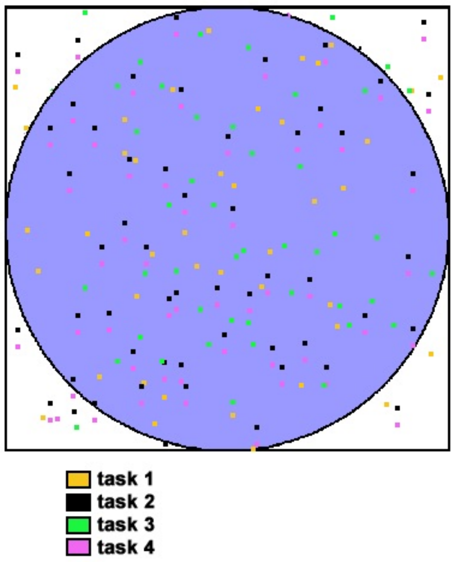
- Two process types are employed:

  Master Process: Holds pool of tasks for worker processes (could be a list of rows, or start/end points):
    - Sends worker a task when requested (needs non-blocking communication)
    - Collects results from workers

  Worker Process: repeatedly does the following:
    - Gets task from master process
    - Performs computation
    - Sends results to master

- Workers do not know beforehand which tasks they will perform.
- Dynamic load balancing at run time: processors running faster tasks will be assigned more work.
- In array processing example, a task could be a row or group of rows.
- The task size could reduce towards end of calculation to allow more flexibility in load balance.

8.8.3 PI Calculation using dartboard method



- Randomly generate points in the square
- Find ratio of number of points in the square to number in the circle
- Pi found from ratio of areas ($\propto$ ratio of numbers)
- Each task has same function but acts completely independently
- One task acts as the master to collect results and compute the value of Pi

See program mpi_pi_reduce.py / mpi_pi_reduce.c / mpi_pi_reduce.for for example.

8.8.4 2d Heat Equation

Solving the equation:

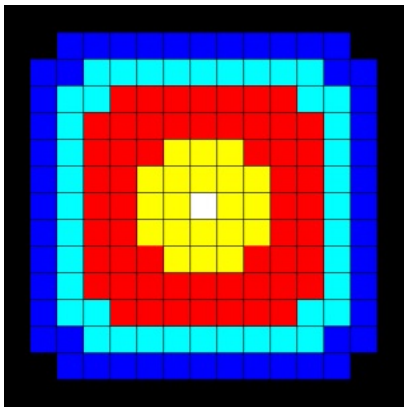
$$\dot{u} - k\nabla^2 u = 0$$

i.e. in 2-d:

$$\boxed{\frac{\partial u}{\partial t} - k\left[\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right] = 0} \tag{1}$$

subject to boundary conditions:

1. $u_0(x, y) = f(x, y)$: initial temperature distribution;
2. $u_t(0, y) = u_t(L_x, y) = u_t(x, 0) = u_t(x, L_y) = 0$.

On a lattice:  derivatives given by:

$$\frac{\partial u}{\partial t} = \frac{u_{t+\Delta t}(x,y) - u_t(x,y)}{\Delta t}$$

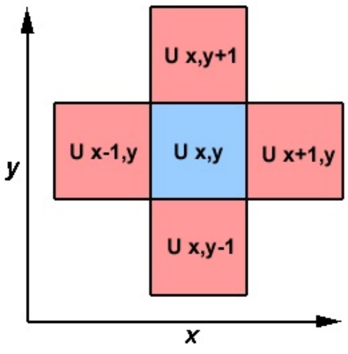
$$\frac{\partial^2 u}{\partial x^2} = \frac{u_t(x+\Delta x, y) - 2u_t(x,y) + u_t(x-\Delta x, y)}{(\Delta x)^2}$$

$$\frac{\partial^2 u}{\partial y^2} = \frac{u_t(x, y+\Delta y) - 2u_t(x,y) + u_t(x, y-\Delta y)}{(\Delta y)^2}$$

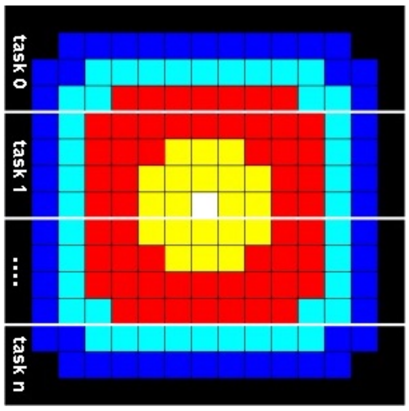
Substituting in Eq. (1) and rearranging, gives time-stepping equation:

$$\begin{aligned} u_{t+\Delta t}(x,y) &= u_t(x,y) \\ &+ c_x[u_t(x+\Delta x, y) - 2u_t(x,y) + u_t(x-\Delta x, y)] \\ &+ c_y[u_t(x, y+\Delta y) - 2u_t(x,y) + u_t(x, y-\Delta y)] \end{aligned}$$

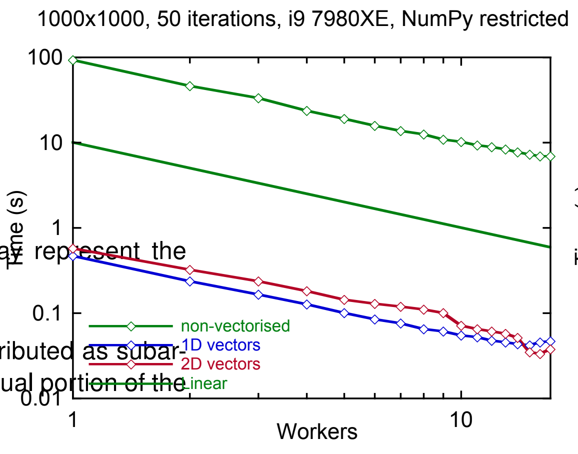
where $c_x = k\Delta t/(\Delta x)^2$ and $c_y = k\Delta t/(\Delta y)^2$. So each update of $u(x,y)$ involves 4 neighbours:

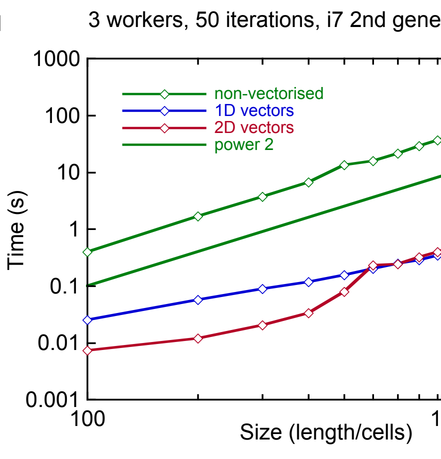


- The elements of a 2-dimensional array represent the temperature at points on the square.
- The entire array is partitioned and distributed as subarrays to all tasks. Each task owns an equal portion of the total array.
- In C or Python, distribute as blocks of rows (columns for Fortran):



- Because the amount of work is ~equal, load balancing should not be too much concern.
- Data dependencies:
  - interior elements belonging to a task are independent of other tasks
  - border elements are dependent upon a neighbour task's data, necessitating communication and synchronisation
- Implement as an SPMD model:
  - Master task sends initial info to workers, and then waits to collect results from all workers
  - Worker tasks calculate solution within specified number of time steps, communicating as necessary with neighbour tasks
- Example program mpi_heat2D.py / mpi_heat2D.c / mpi_heat2D.for.
- Results shown for mpi_heat2D.py program, plus vectorised versions of the code (see later).
- The code execution time scales inversely as the number of workers:





- and also scales as the square of the side-length of the square lattice, as expected.

- Each data point is the average of five runs. Methods of vectorisation will be discussed in a later lecture.

8.9 Overview of MPI send modes

Adapted from:
http://www.mcs.anl.gov/research/projects/mpi/sendmode.html

- MPI has different ***send*** modes.
- These represent different choices of buffering (where is the data kept until it is received) and synchronization (when does a send complete).
- Here, let *send buffer* be the user-provided buffer (e.g. an array) to send.

MPI_Send / MPI.COMM_WORLD.Send MPI_Send will not return until you can use the send buffer. It may or may not block (it is allowed to buffer, either on the sender or receiver side, or to wait for the matching receive).

MPI_Bsend / MPI.COMM_WORLD.Bsend May buffer; returns immediately and you can use the send buff late add-on to the MPI specification. Should be used only when absolutely necessary.

MPI_Ssend / MPI.COMM_WORLD.Ssend will not return until matching receive posted

MPI_Rsend / MPI.COMM_WORLD.Rsend May be used ONLY if matching receive already posted. User responsible for writing a correct program.

MPI_Isend / MPI.COMM_WORLD.Isend Nonblocking send. But not necessarily asynchronous. You can NOT reuse the send buffer until either a successful, wait/test or you KNOW that the message has been received (see MPI_Request_free). Note also that while the I refers to immediate, there is no performance requirement on MPI_Isend. An immediate send must return to the user without requiring a matching receive at the destination. An implementation is free to send the data to the destination before returning, as long as the send call does not block waiting for a matching receive. Different strategies of when to send the data offer different performance advantages and disadvantages that will depend on the application.

MPI_Ibsend / MPI.COMM_WORLD.Ibsend buffered nonblocking

MPI_Issend / MPI.COMM_WORLD.Issend Synchronous nonblocking. Note that a Wait/Test will complete only when the matching receive is posted.

MPI_Irsend / MPI.COMM_WORLD.Irsend As with MPI_Rsend, but nonblocking.

Note that *nonblocking* refers ONLY to whether the data buffer is available for reuse after the call. No part of the MPI specification, for example, mandates concurent operation of data transfers and computation.

Recommendations: The best performance is likely if you can write your program so that you could use just MPI_Ssend; in that case, an MPI implementation can completely avoid buffering data. Use MPI_Send instead; this allows the MPI implementation the maximum flexibility in choosing how to deliver your data. If nonblocking routines are necessary, then try to use MPI_Isend or MPI_Irecv. Use MPI_Bsend only when it is too inconvienent to use MPI_Isend. The remaining routines, MPI_Rsend, MPI_Issend, etc., are rarely used but may be of value in writing system-dependent message-passing code entirely within MPI.

mpi4py send modes: For a full list of methods available in mpi4py type `dir(MPI.COMM_WORLD)` at a Python prompt. For an argument list of a given function, and really not much else, use the help command e.g. `help(MPI.COMM_WORLD.send)`. Here is a list of the send modes available in mpi4py:

```
B(b)send, I(i)bsend, Irsend, I(i)send, I(i)ssend, Rsend, S(s)end, S(s)sen
```

Exercise Week 3

Exercise Week 3

- Download and install MPI on your computer. There are various versions available. Installation is a little complex – instructions for Windows and Mac, and for Python, C and Fortran, are included in the course installation notes on Blackboard. If you are using Python, `mpi4py` can be installed as part of Anaconda.
- If you are using Windows, you will also need to install MS-MPI for this to work. If you are programming with C/C++, you will need MS-MPI on Windows; OpenMPI or MPI-CH on other operating systems.
- Test your installation using the programs included in this lecture – these can be downloaded separately from Blackboard. Experiment with different numbers of tasks and compare timings for the prime number program.
- Upload the MPI program mpi_pi_reduce.[py/c/for] to BlueCrystal 4 and experiment with running programs in this environment. The notes on Blackboard cover use of C/Fortran and Python on BlueCrystal phase 4, and should help you to get started. (Hint: you can do very quick test runs at the terminal, but anything longer than about 1s should be submitted to the job queue.)

- Produce a timing graph mpi_pi_reduce.[py/c/for] for up to 28 MPI tasks (one node).