Introduction to Parallel Programming with MPI

Nikolai Sergueev

Center for Computational Research
and Cyber Infrastructure Support
(CCR-CIS)

UNIVERSITY OF SOUTH CAROLINA

Columbia, SC
July 29, 2011

sergueev@sc.edu

Overview

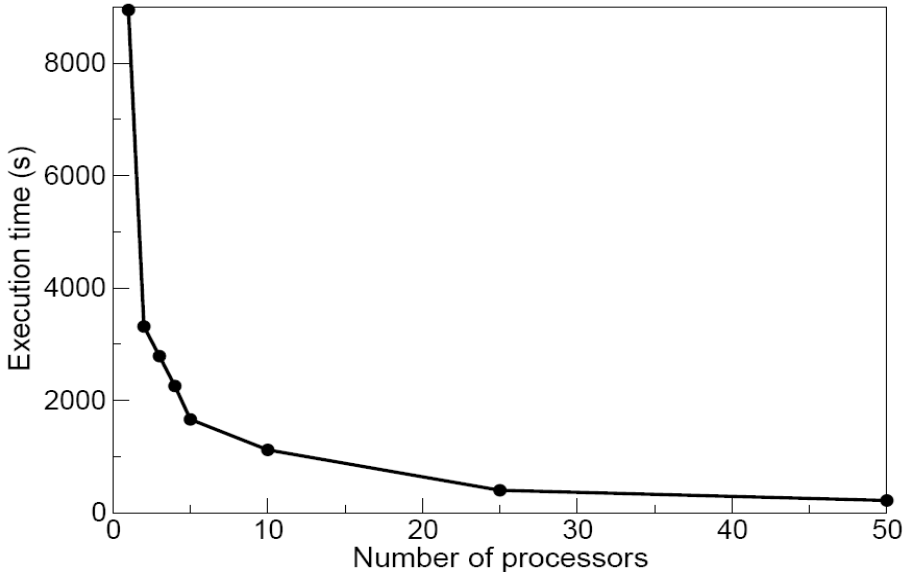
- Short intro to parallel computing
  - Parallel architecture
  - Parallel programming models
  - Hardware, software, etc
- Message Passing Model
- Getting started with MPI
  - MPI program structure
  - Compiling and executing MPI program
- Blocking Communications
- Non-Blocking Communications
- Collective Communications

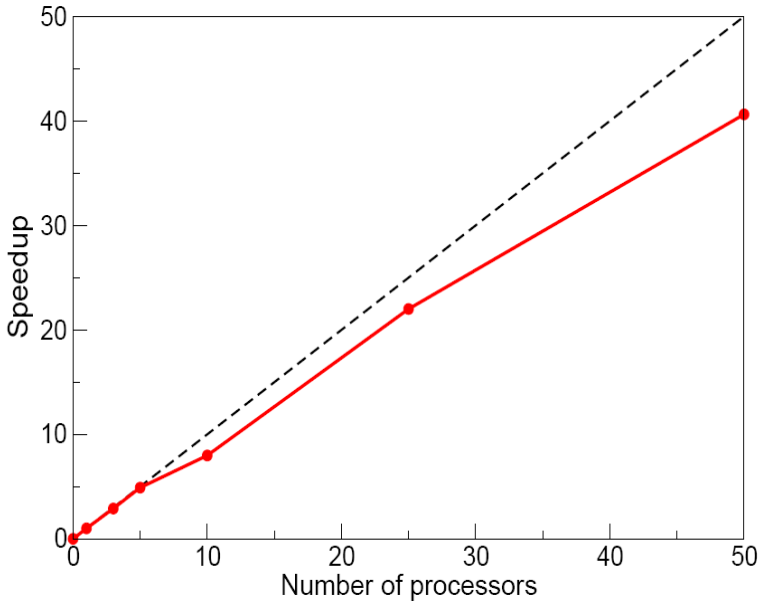
sergueev@sc.edu

What is the purpose of parallelization ?

Example: Atomistic electronic structure code McDCal

Benchmarking: AMD Opteron cluster, 10 Gigabit Ethernet network





decrease the computational time needed to solve problem

increase the size of the problem that can be solved

sergueev@sc.edu

Recipe for writing a parallel program:

1. Decompose algorithm or data into the parts
   (it is better if parts can be handled independently)

2. Distribute the parts over the multiple processors simultaneously

3. Coordinate work and communication of those processors

Intro to Parallel Computing

What do we need to run in parallel ?

- Bunch of processors (the workers)
- Network (linking those workers)
- Environment to create and handle parallel execution
  - Operating system
  - Programming paradigm

    -MPI
    -MPL
    -PVM
- Parallel algorithm and parallel program

Parallel programming models:

Shared Memory

Tasks share common address space, which they write and read
No data ownership – no need for data communication

Distributed Memory / Message Passing

Tasks use their own local memory
Tasks exchange data through communications by sending and receiving messages

Data Parallel

Parallelization is on a data set, tasks work together on the same data, though on its different partitions

Hybrid Models

Combination of the above models

Message Passing Model

What is MPI ?

MPI – Message Passing Interface

It is a specification for creating interface libraries
but it is not a library, not a product

- Designed for parallel computers and clusters
- Goal of MPI – interface needs to be:
  - practical
  - portable
  - efficient
  - flexible
- There are different MPI implementations:
  MPICH, MPICH2, OpenMPI

Moe info on MPI: http://www.mpi-forum.org, http://*www.mcs.anl.gov/**mpi**/*, *http://www.open-mpi.org*

Getting Started with MPI

How to program with MPI ?

All operations are performed by subroutine calls

MPI_Init, MPI_Send, MPI_Recv, MPI_Finalize, etc

All MPI subroutines can be divided into several categories:

- Subroutines used to initialize, manage, and terminate operations
- Subroutines for communications between pairs of processors
- Subroutines for communications among group of processors
- Subroutines for data type definition

What do you need to know for a simple MPI program ?

MPI_INIT	-initializes MPI
MPI_COMM_SIZE	-get number of processors
MPI_COMM_RANK	-get identification number
MPI_FINALIZE	-finalize MPI

Getting Started with MPI

First MPI program “Hello World”

Look for “**hello.c**”

C example

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


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

  MPI_Init (&argc, &argv);      /* starts MPI */
  MPI_Comm_rank (MPI_COMM_WORLD, &rank);        /* get current process id */
  MPI_Comm_size (MPI_COMM_WORLD, &size);        /* get number of processes */
  printf( "Hello world from process %d of %d\n", rank, size );
  MPI_Finalize();
  return 0;
}
```


sergueev@sc.edu

Getting Started with MPI

First MPI program “Hello World”

Look for “**hello.f**”

Fortran example

```
program hello
      include 'mpif.h'
      integer rank, size, ierror, tag, status(MPI_STATUS_SIZE)

      call MPI_INIT(ierror)
      call MPI_COMM_SIZE(MPI_COMM_WORLD, size, ierror)
      call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)
      print*, "Hello world from process",rank,"of",size
      call MPI_FINALIZE(ierror)
      end
```


sergueev@sc.edu

Getting Started with MPI

Now lets try this MPI code on your desktops …

These are quad-core machines with OpenMPI installed

To compile C program:

mpicc hello.c -o hello.exe

To compile Fortran program:

mpif77 hello.c -o hello.exe

To execute them:

mpirun –np NUMPROC ./hello.exe

sergueev@sc.edu

Getting Started with MPI

First MPI program “Hello World”

Look for a file “hello.c”

C example

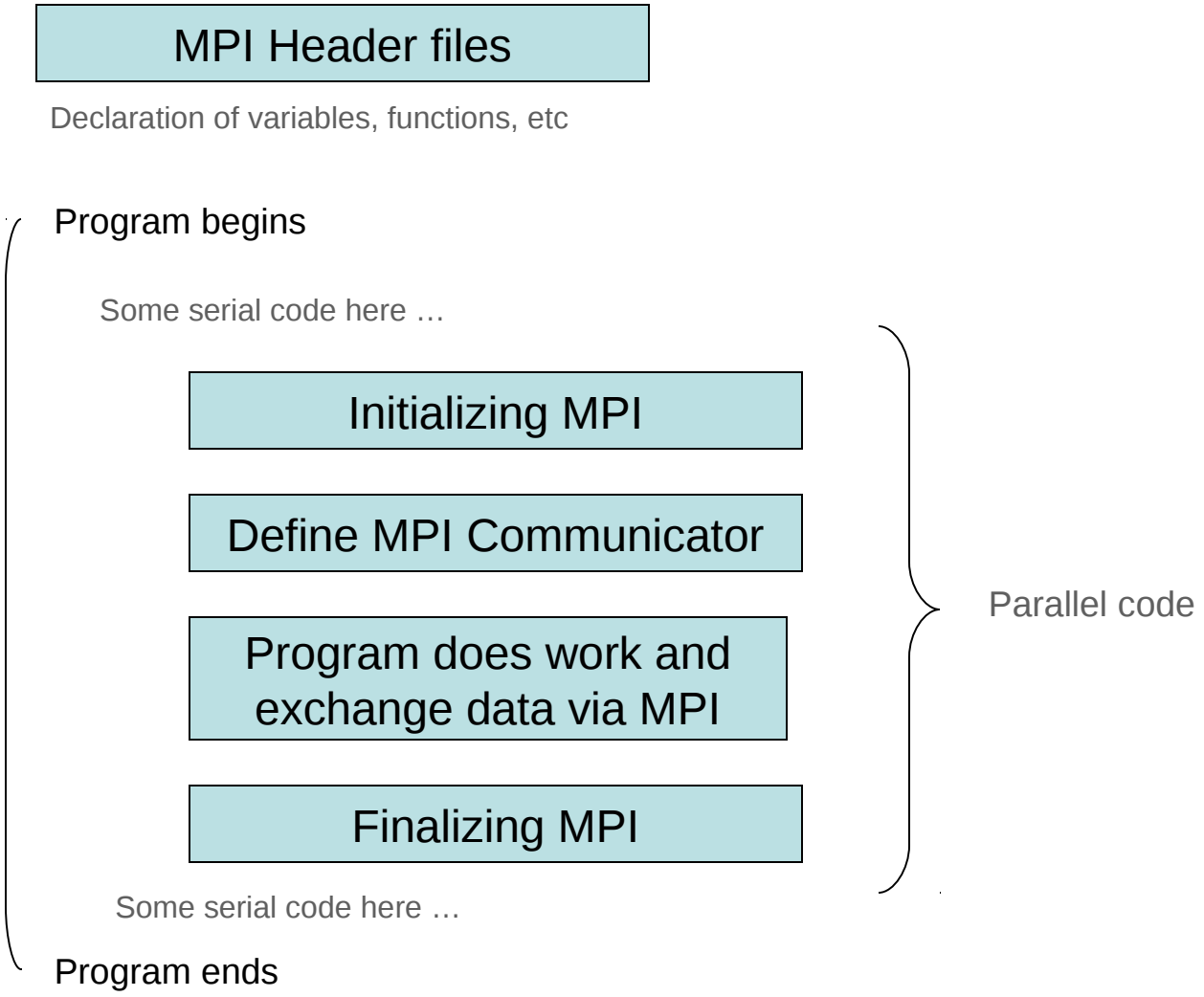
Output:

Hello world from process 1 of 4
Hello world from process 2 of 4
Hello world from process 3 of 4
Hello world from process 0 of 4

sergueev@sc.edu

Getting Started with MPI

Basic structure of MPI program



sergueev@sc.edu

Getting Started with MPI

MPI Header files

Both the main program and all subroutines should have a header file declaration

In C:

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

In Fortran:

```
include 'mpif.h'
```

Header files contain:

- MPI constants
- macros
- definitions
- function prototypes

sergueev@sc.edu

Getting Started with MPI

Initializing MPI

The very first thing Main program should do is to call MPI_INIT
It should be called just once !

In C:

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

In Fortran:

```
integer error
call MPI_INIT(error)
```

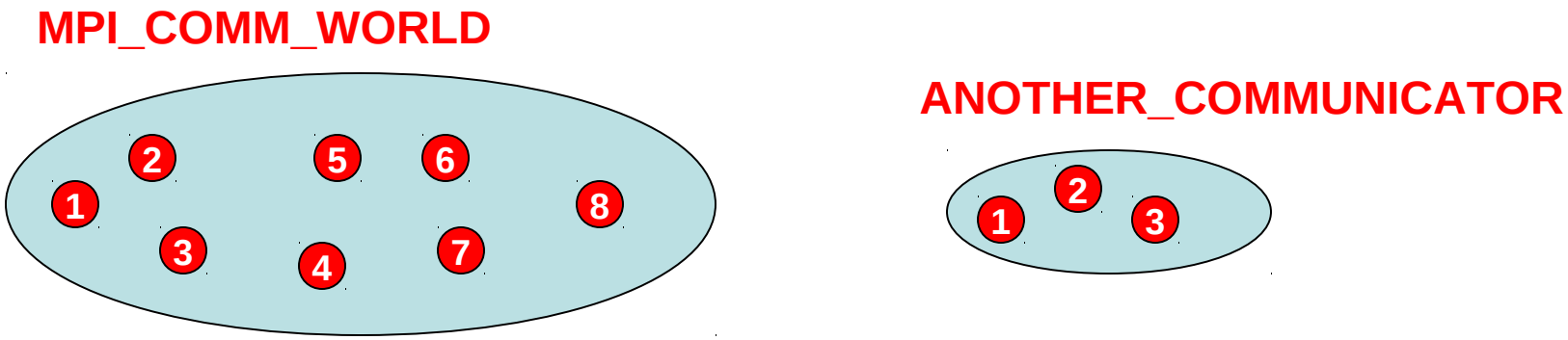

sergueev@sc.edu

Getting Started with MPI

MPI Communicator

What is MPI communicator ?

A handle that represents a group of processes communicating between each other



Here processes 1 2 3 are in both communicators

In those communicators they have different ranks

MPI_COMM_WORLD – is default communicator that contains all initial processes

Getting Started with MPI

MPI Communicator Size

It tells how many processes are in the communicator

In C:

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

In Fortran:

```
integer comm, size, error
call MPI_COMM_SIZE(comm, size, error)
```

number of processes = size

MPI Process Rank

Rank is the ID of a given process in the communicator

In C:

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

In Fortran:

```
integer comm, size, error
call MPI_COMM_RANK(comm, rank, error)
```

ID of a process = rank

sergueev@sc.edu

Getting Started with MPI

Finalizing MPI

Every MPI code needs to be terminated. The termination subroutine is called just once !
No MPI calls after MPI finalizing !

In C:

```
MPI_Finalize( );
```

In Fortran:

```
integer error;
call MPI_FINALIZE(error);
```

MPI Data Types

MPI Data Type	C Data Type
MPI_CHAR	signed char
MPI_SHORT	signed short in
MPI_INT	signed int
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_LONG	long

MPI Data Type	Fortran Data Type
MPI_CHARACTER	character
MPI_INTEGER	integer
MPI_REAL	real
MPI_DOUBLE_PRECISION	double precision
MPI_COMPLEX	complex
MPI_DOUBLE_COMPLEX	double complex
MPI_LOGICAL	logical

Communications: Blocking and Non-blocking

Two major communications: Send and Receive

Lets divide communication into 3 phases:

1. Initiate communication
2. Do some work
3. Wait for communication to complete

Definition: "**Completion**"
… means that the memory allocated for the data transfer can be accessed

If we Send: variable sent can be safely accessed and reused
If we Receive: variable received can be safely used now

Definition:
"**Blocking** communication" : MPI subroutine return guaranties completion
"**Non-blocking** communication" : MPI subroutine returns immediately right after it has been called; no guaranties that communication is completed; user must check for completion

Blocking communications

MPI Send Communication

In C:

MPI_Send(void *buffer, int count, MPI_Datatype data_type, int destination, int tag, MPI_Comm comm);

In Fortran:

call MPI_SEND(buffer, count, data_type, destination, tag, comm, error)

buffer - data that needs to be sent
count - number of elements to be sent
data_type – what is the type of data in a buffer
destination – rank of the receiver
tag – label of the message
comm – communicator
error – error code

Definition: "tag" – is the identification number of the message

Messages can be filtered at the receiver by specifying a certain tag
Wildcarding: tag=MPI_ANY_TAG means no filtering: any message will be accepted

Blocking communications

MPI Receive Communication

In C:

int **MPI_Recv(void *buffer, int count, MPI_Datatype data_type, int source, int tag, MPI_Comm comm, MPI_Status *status);**

In Fortran:

call MPI_RECV(buffer, count, data_type, source, tag, comm, status)

source – rank of the process that sent the message
status – array that contain information about transferred message

Wildcarding: source=MPI_ANY_SOURCE means that messages from any source will be accepted

Blocking communications

Status Array

Status – data structure that should be allocated in the program

In C:

```
int recvd_tag, recvd_from, recvd_count;
MPI_Status status;
recvd_tag = status.MPI_TAG
recvd_from = status.MPI_SOURCE
MPI_Get_count(&status, data_type, &recvd_count);
```

In Fortran:

```
integer recvd_tag, recvd_from, recvd_count, error
integer status(MPI_STATUS_SIZE)
recvd_tag = status(MPI_TAG)
recvd_from = status(MPI_SOURCE)
call MPI_GET_COUNT(status, data_type, recvd_count, error)
```

Information that we get:

recvd_tag – tag of the original message sent
recvd_from – where the message came from

sergueev@sc.edu

Blocking communications

Example: simple Send/Receive code

Look for "**send_receive.c**"

C example:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
/* Run with two processes */
int main(int argc, char *argv[]) {
        int rank, i, count;
        float sendbuf[100],recvbuf[100];
        MPI_Status status;
        MPI_Init(&argc,&argv);
        MPI_Comm_rank(MPI_COMM_WORLD,&rank);
        if(rank==1) {
                for(i=0;i<100;++i) sendbuf[i]=i;
                MPI_Send(sendbuf,100,MPI_FLOAT,0,55,MPI_COMM_WORLD);
        } else
        {

MPI_Recv(recvbuf,100,MPI_FLOAT,MPI_ANY_SOURCE,55,MPI_COMM_WORLD,&status);
                printf("Process:%d Got data from processor %d \n",rank, status.MPI_SOURCE);
                MPI_Get_count(&status,MPI_FLOAT,&count);
                printf("Process:%d Got %d elements with tag=%d \n",rank,count,status.MPI_TAG);
                printf("Process:%d recvbuf[5]=%f \n",rank,recvbuf[5]);
        }
        MPI_Finalize();
}
```


sergueev@sc.edu

Blocking communications

Example: simple Send/Receive code

Look for "**send_receive.f**"

Fortran example:

```
program main
include "mpif.h"

integer rank, i, count, ierror, status(MPI_STATUS_SIZE)
real*8 sendbuf(100), recvbuf(100)

call MPI_INIT(ierror)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, ierror)

if(rank.eq.1)  then
   do i=1,100
      sendbuf(i)=i
   enddo
   call MPI_SEND(sendbuf,100,MPI_REAL,0,55,MPI_COMM_WORLD,ierror)
else
   call MPI_RECV(recvbuf,100,MPI_REAL,MPI_ANY_SOURCE,55,
>           MPI_COMM_WORLD,status,ierror)
   print*, "Processor:",rank,"Got data from processor",
>                       status(MPI_SOURCE)
   call MPI_GET_COUNT(status,MPI_REAL,count,ierror)
   print*, "Processor:",rank,"Got",count,"elements with tag=",
>          status(MPI_TAG)
   print*, "Processor:",rank,"recvbuf[5]=",recvbuf(5)

endif
call MPI_FINALIZE(ierror)
end
```


sergueev@sc.edu

Blocking communications

Example: simple Send/Receive code

Look for "**send_receive.c**"

Running C example:

Compile: **mpicc send_receive.c -o send_receive.exe**

Execute: **mpirun -np 2 ./send_receive.exe**

Output:

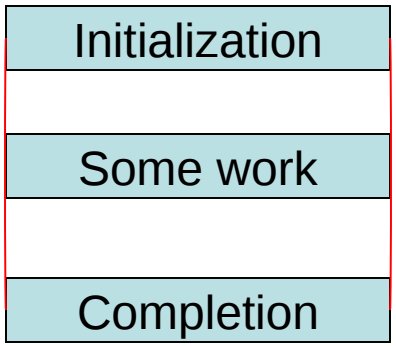
```
Process:0 Got data from processor 1
Process:0 Got 100 elements with tag=55
Process:0 recvbuf[5]=5.000000
```

Blocking versus Non-Blocking

MPI_Send and MPI_Recv are both **blocking** communications

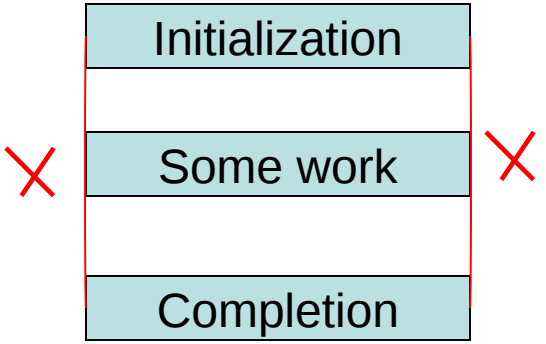
Code waits until the Completion occurs

Blocking communication



tightly linked

Non-Blocking communication



separated !

sergueev@sc.edu

Non-Blocking communications

Non-blocking MPI Send Communication

In C:

MPI_Isend(void *buffer, int count, MPI_Datatype data_type, int destination, int tag, MPI_Comm comm, MPI_Request *request);

In Fortran:

call MPI_ISEND(buffer, count, data_type, destination, tag, comm, request, error)

Non-blocking MPI Receive Communication

In C:

int **MPI_Irecv(void *buffer, int count, MPI_Datatype data_type, int source, int tag, MPI_Comm comm, MPI_Status *status, MPI_Request *request);**

In Fortran:

call MPI_IRECV(buffer, count, data_type, source, tag, comm, status, request, ierror)

"**I**" stands for "Immediate" as the subroutine returns immediately !

sergueev@sc.edu

Non-Blocking communications

Why use Blocking message passing ?

Advantage: Right after the Initialization the code does not wait for Completion; can do useful calculations in a meantime;

Disadvantage: The user needs to check whether the MPI subroutine is completed

How to check for Completion ?

In C:

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

In Fortran:

```
call MPI_TEST(request, flag, status, error)
```

Non-Blocking communications

How to synchronize Non-blocking communications ?

Use Wait subroutine: it causes the code to wait until the communication pointed by ***request*** is completed

In C:

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

In Fortran:

```
call MPI_WAIT(request, status, error)
```

Blocking communications

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
/* Run with two processes */
int main(int argc, char *argv[]) {
int rank, i, count;
float data1[100],data2[100];
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if(rank==0) {
        MPI_Send(data1,100,MPI_FLOAT,1,10,MPI_COMM_WORLD);
        MPI_Recv(data2,100,MPI_FLOAT,1,20,MPI_COMM_WORLD,&status);
        printf("Processor %d: Messages sent and received\n",rank);
}
if(rank==1)
{
        MPI_Send(data2,100,MPI_FLOAT,0,20,MPI_COMM_WORLD);
        MPI_Recv(data1,100,MPI_FLOAT,0,10,MPI_COMM_WORLD,&status);
        printf("Processor %d: Messages sent and received\n",rank);
}
MPI_Finalize();
}
```

This code **will not work**: both MPI_Send and MPI_Recv are blocking

Blocking communications

Look for "**deadblock.c**"

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
/* Run with two processes */
int main(int argc, char *argv[]) {
int rank, i, count;
float data1[100],data2[100];
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if(rank==0) {
        MPI_Ssend(data1,100,MPI_FLOAT,1,10,MPI_COMM_WORLD);
        MPI_Recv(data2,100,MPI_FLOAT,1,20,MPI_COMM_WORLD,&status);
        printf("Processor %d: Messages sent and received\n",rank);
}
if(rank==1)
{
        MPI_Ssend(data2,100,MPI_FLOAT,0,20,MPI_COMM_WORLD);
        MPI_Recv(data1,100,MPI_FLOAT,0,10,MPI_COMM_WORLD,&status);
        printf("Processor %d: Messages sent and received\n",rank);
}
MPI_Finalize();
}
```

Not working ...

sergueev@sc.edu

Non-Blocking communications

Solution: use Non-Blocking MPI routines

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
/* Run with two processes */
int main(int argc, char *argv[]) {
int rank, i, count;
float data1[100],data2[100];
MPI_Status status;
MPI_Request request;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if(rank==0) {
        MPI_Isend(data1,100,MPI_FLOAT,1,10,MPI_COMM_WORLD,&request);
        MPI_Recv(data2,100,MPI_FLOAT,1,20,MPI_COMM_WORLD,&status);
        MPI_Wait(&request, &status);
        printf("Processor %d: Messages sent and received\n",rank);
}
if(rank==1)
{
        MPI_Isend(data2,100,MPI_FLOAT,0,20,MPI_COMM_WORLD,&request);
        MPI_Recv(data1,100,MPI_FLOAT,0,10,MPI_COMM_WORLD,&status);
        MPI_Wait(&request, &status);
        printf("Processor %d: Messages sent and received\n",rank);
}
MPI_Finalize();
}
```

Look for "**deadblock_fixed1.c**"

This code **Works** !

Blocking communications

Another solution : swap Send and Receive for one of the processes

Look for "**deadblock_fixed2.c**"

This code **Works** !

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
/* Run with two processes */
int main(int argc, char *argv[]) {
int rank, i, count;
float data1[100],data2[100];
MPI_Status status;
MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if(rank==0) {
        MPI_Recv(data2,100,MPI_FLOAT,1,20,MPI_COMM_WORLD,&status);
        MPI_Ssend(data1,100,MPI_FLOAT,1,10,MPI_COMM_WORLD);
        printf("Processor %d: Messages sent and received\n",rank);
}
if(rank==1)
{
        MPI_Ssend(data2,100,MPI_FLOAT,0,20,MPI_COMM_WORLD);
        MPI_Recv(data1,100,MPI_FLOAT,0,10,MPI_COMM_WORLD,&status);
        printf("Processor %d: Messages sent and received\n",rank);
}
MPI_Finalize();
}
```

Blocking communications

Another solution : swap Send and Receive for one of the processes

Running either "**deadblock_fixed1.c**" or "**deadblock_fixed2.c**"

mpirun -np 2 deadblock_fixed1.exe

Output:

Processor 0: Messages sent and received
Processor 1: Messages sent and received

Collective communications

What is Collective communication ?

Communication that involves a group of processes and called by all the processes in the communicator

Several good features about Collective communications:

- they do not interfere with point-to-point communications
- they are non-blocking by definition
- no need for tags

Several limitations:

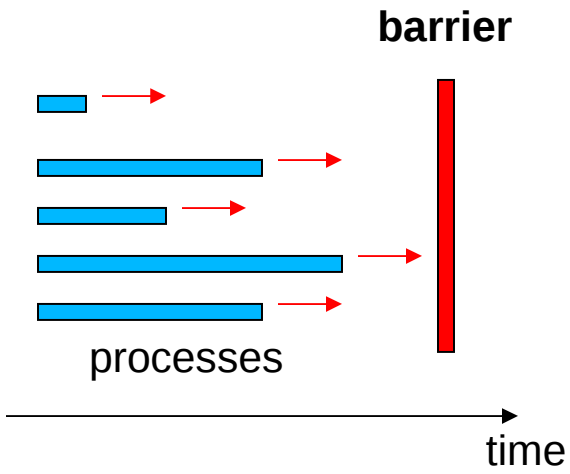
- all processes must call collective routines
- communications are not synchronized (except for Barrier)
- buffer for the receiver must be consistent with that for the sender

Collective communications

MPI Barrier synchronization

Code waits until all the processes are synchronized

Cause idle time for some processes



In C:

```
MPI_Barrier(MPI_Comm comm);
```

In Fortran:

```
call MPI_BARRIER(comm)
```

Collective communications

MPI Broadcast

The same data is sent from the root to all processes in the communicator

In C:

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

In Fortran:

```
call MPI_Bcast(buffer, count, data_type, root, comm, error)
```

MPI Scatter

Different data is sent to each process in the communicator

In C:

```
MPI_Scatter(void *sendbuffer, int sendcount, MPI_Datatype senddata_type,
            void *recvbuffer, int recvcount, MPI_Datatype recvdata_type,
            int root, MPI_Comm comm);
```

In Fortran:

```
call MPI_SCATTER(sendbuffer, sendcount, senddata_type,
                 recvbuffer, recvcount, recvdata_type,
                 root, comm, error);
```

Collective communications

Example: code demonstrating Broadcast subroutine

Look for "**broadcast.c**"

C example:

```
# include <mpi.h>
Int main (int argc, char *argv[])
{
      int rank;
      double param;
      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      if(rank==2) param=23.0;
      MPI_Bcast(&param,1,MPI_DOUBLE,2,MPI_COMM_WORLD);
      printf("P:%d after broadcast parameter is %f\n",rank,param);
      MPI_Finalize();
}
```


sergueev@sc.edu

Collective communications

Example: code demonstrating Broadcast subroutine

Fortran example:

Look for "**broadcast.f**"

```
program BROADCAST
include 'mpif.h'
integer error, rank, size
real param
call MPI_INIT(error)
call MPI_COMM_RANK(MPI_COMM_WORLD, rank, error)
if(rank.eq.5) param=23.0
call MPI_BCAST(param,1,MPI_REAL,5,MPI_COMM_WORLD,error)
print*,"P:", rank, "after broadcast param is ", param
call MPI_FINALIZE(error)
end
```


sergueev@sc.edu

Collective communications

Example: code demonstrating Broadcast subroutine

Running “broadcast.c” :

mpirun -np 4 ./broadcast.exe

Output:

P:0 after broadcast parameter is 23.000000
P:2 after broadcast parameter is 23.000000
P:1 after broadcast parameter is 23.000000
P:3 after broadcast parameter is 23.000000

Collective communications

Global Reduction Communications

What are these communications ?

Communications that involve computation operation on the data which is transmitted over the processes in the communicator

Advantage:

- collect all the date across the communicator
- reduce multiple data to a single one
- perform desired operation with the data
- store on root or share date over the processes

MPI_Reduce

In C:

```
MPI_Reduce(void *sendbuffer, void *recvbuffer, int count,
    MPI_Datatype data_type, MPI_Op operation, int root, MPI_Comm comm);
```

In Fortran:

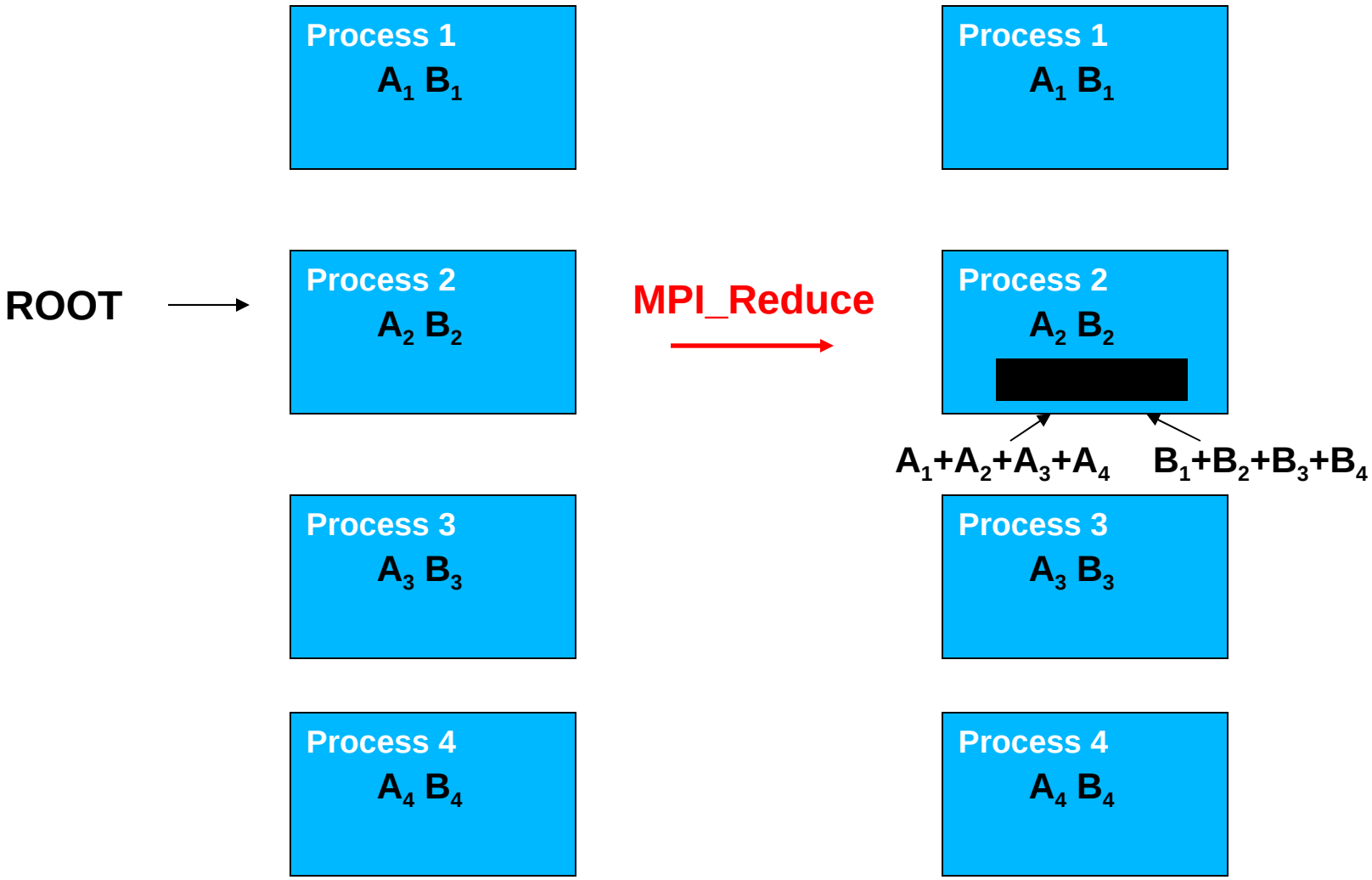
```
call MPI_Reduce(sendbuffer, recvbuffer, count,
            data_type, operation, root, comm, error);
```


sergueev@sc.edu

Collective communications

MPI_Reduce :

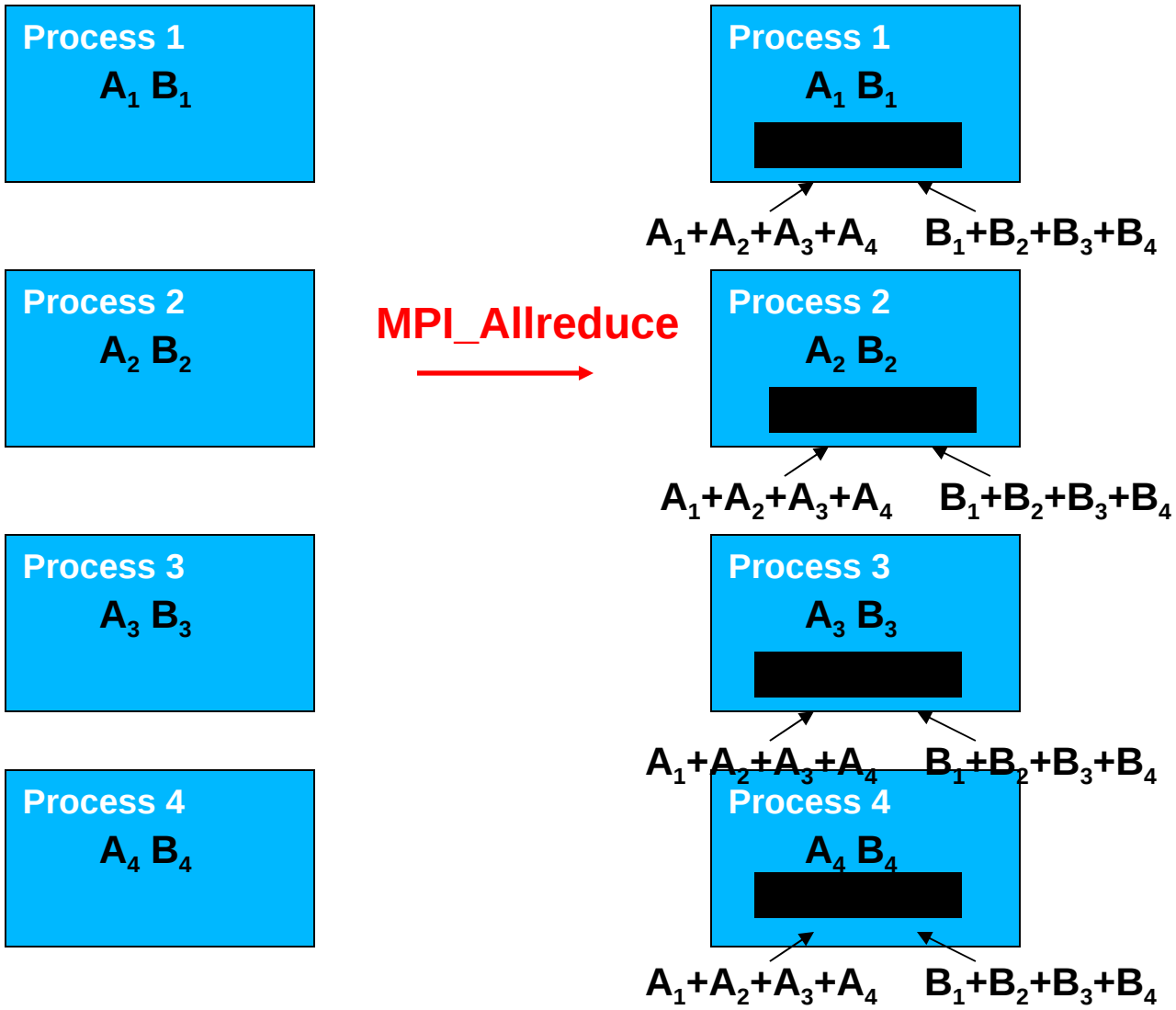
OPERATION = SUMMATION



sergueev@sc.edu

Collective communications

MPI_Allreduce : **OPERATION = SUMMATION**



There is no ROOT

sergueev@sc.edu

Collective communications

Reduction operations

MPI Name	Operation
MPI_SUM	summation
MPI_PROD	product
MPI_MIN	minimum
MPI_MAX	maximum
MPI_LAND	logical AND
MPI_LOR	logical OR

Other useful reduction subroutines:

MPI_REDUCE_SCATTER
MPI_SCAN

Other useful collective subroutines:

MPI_GATHER, MPI_ALLGATHER
MPI_ALLTOALL

Collective communications

Example demonstrating MPI_Reduce operation: Calculation of PI

Look for "**calculate_pi.c**"

```
#include "mpi.h"
#include <stdio.h>
#include <math.h>
int main( int argc, char *argv[] )
{
    int n, myid, numprocs, i;
    double PI25DT = 3.14159265358979323846264 3;
    double mypi, pi, h, sum, x;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    while (1) {
        if (myid == 0) {
            printf("Enter the number of intervals: (0 quits) "); fflush(stdout);
            scanf("%d",&n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0)
            break;
```


sergueev@sc.edu

Collective communications

Example demonstrating MPI_Reduce operation: Calculation of PI

Look for "**calculate_pi.c**"

```
else {
        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);
        if (myid == 0)
            printf("pi is approximately %.16f, Error is %.16f\n",
                pi, fabs(pi - PI25DT));
    }
  }
  MPI_Finalize();
  return 0;
}
```


sergueev@sc.edu

Collective communications

Example demonstrating MPI_Reduce operation: Calculation of PI

Running "**calculate_pi.c**"

mpirun -np 4 ./calculate_pi.exe

Output:

```
Enter the number of intervals: (0 quits) 100
pi is approximately 3.1416009869231249, Error is 0.0000083333333318
Enter the number of intervals: (0 quits) 500
pi is approximately 3.1415929869231269, Error is 0.0000003333333338
Enter the number of intervals: (0 quits) 1000
pi is approximately 3.1415927369231262, Error is 0.0000000833333331
```

Acknowledgement

- **Center for Computational Research and Cyber Infrastructure Support**
  - Professor Robert Sharpley
  - Dr. Jerry Ebalunode

- **Research Computing Team**
  - Andrew Yancey
  - Glenn Dufour
  - Frank Bhakit
  - Steven

- **Financial Support**
  - USC Vice President Office of Research and Graduate Education
    NSG EPSCoR RII Track-1 and Track-2

sergueev@sc.edu