Parallel Programming with MPI on the Odyssey Cluster

Plamen Krastev

Office: Oxford 38, Room 204

Email: plamenkrastev@fas.harvard.edu

FAS Research Computing

Harvard University

Objectives:

To introduce you to the major concepts and ideas in parallel computing

To give you the basic knowledge to write simple parallel MPI programs

To provide the information required for running your applications efficiently on the Odyssey cluster

Outline:

- □ Parallelizing your program
- □ Parallelizing DO / FOR loop
- □ Collective communication
- □ Point to point communication
- □ Communication groups
- □ Parallel I / O
- □ Python MPI
- □ Debugging and optimization

Parallelizing your program:

You parallelize your program to run faster, and to solve larger problems.

How much faster will the program run?

Speedup:

$$S(n) = \frac{T(1)}{T(n)}$$

Time to complete the job on **one** process

Time to complete the job on **n** process

Efficiency:

$$E(n) = \frac{S(n)}{n}$$

Tells you how efficiently you parallelize your code

Oversimplified example:

p → fraction of program that can be parallelized

1 - p → fraction of program that cannot be parallelized

n → number of processors

Then the time of running the parallel program will be

$1 - p + p/n$ of the time for running the serial program

80% can be parallelized

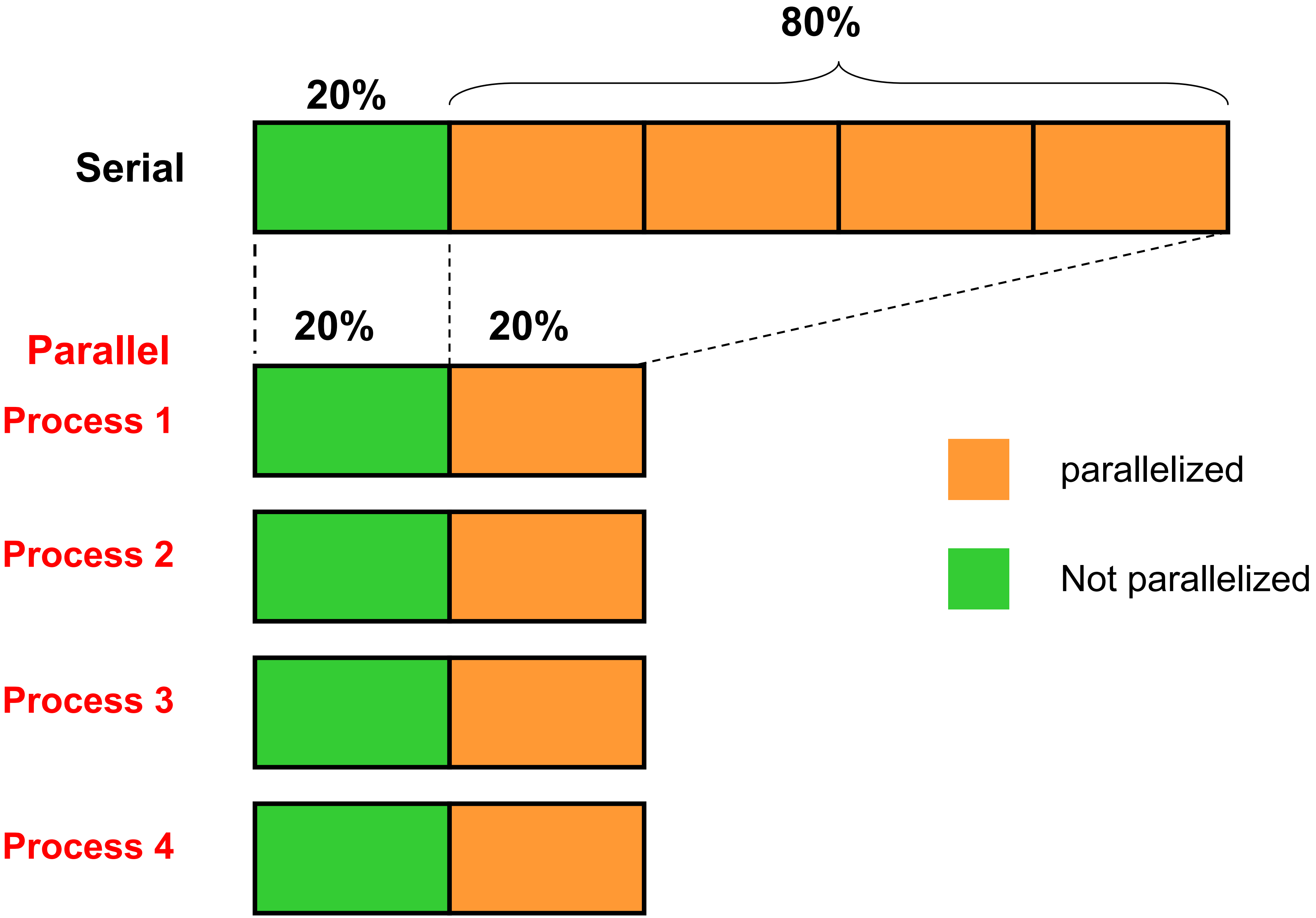
20 % cannot be parallelized

n = 4

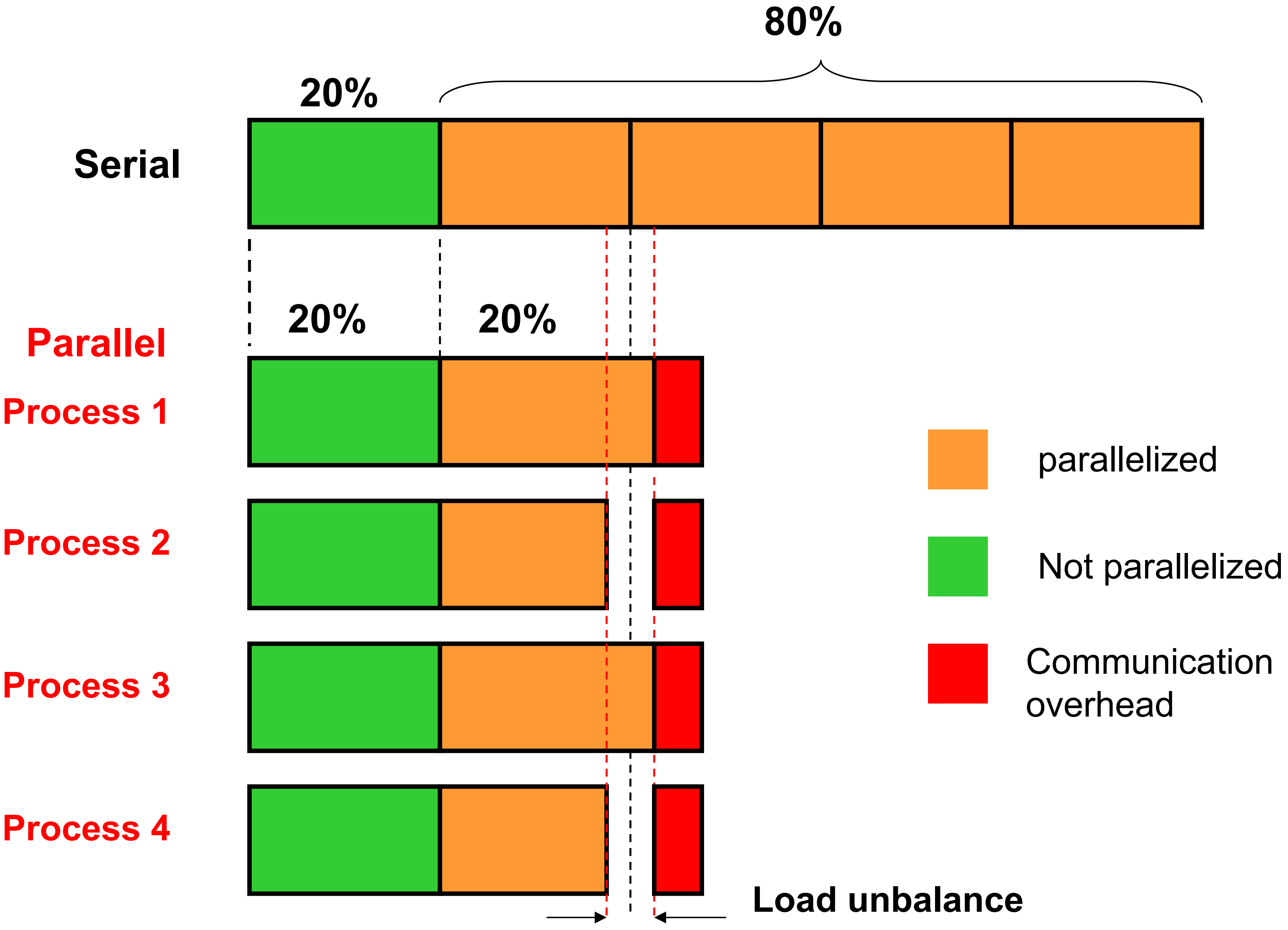
1 - 0.8 + 0.8 / 4 = 0.4 i.e., 40% of the time for running the serial code

You get 2.5 speed up although you run on 4 cores since only 80% of your code can be parallelized (assuming that all parts in the code can complete in equal time)

Oversimplified example, cont'd:



More realistic example:

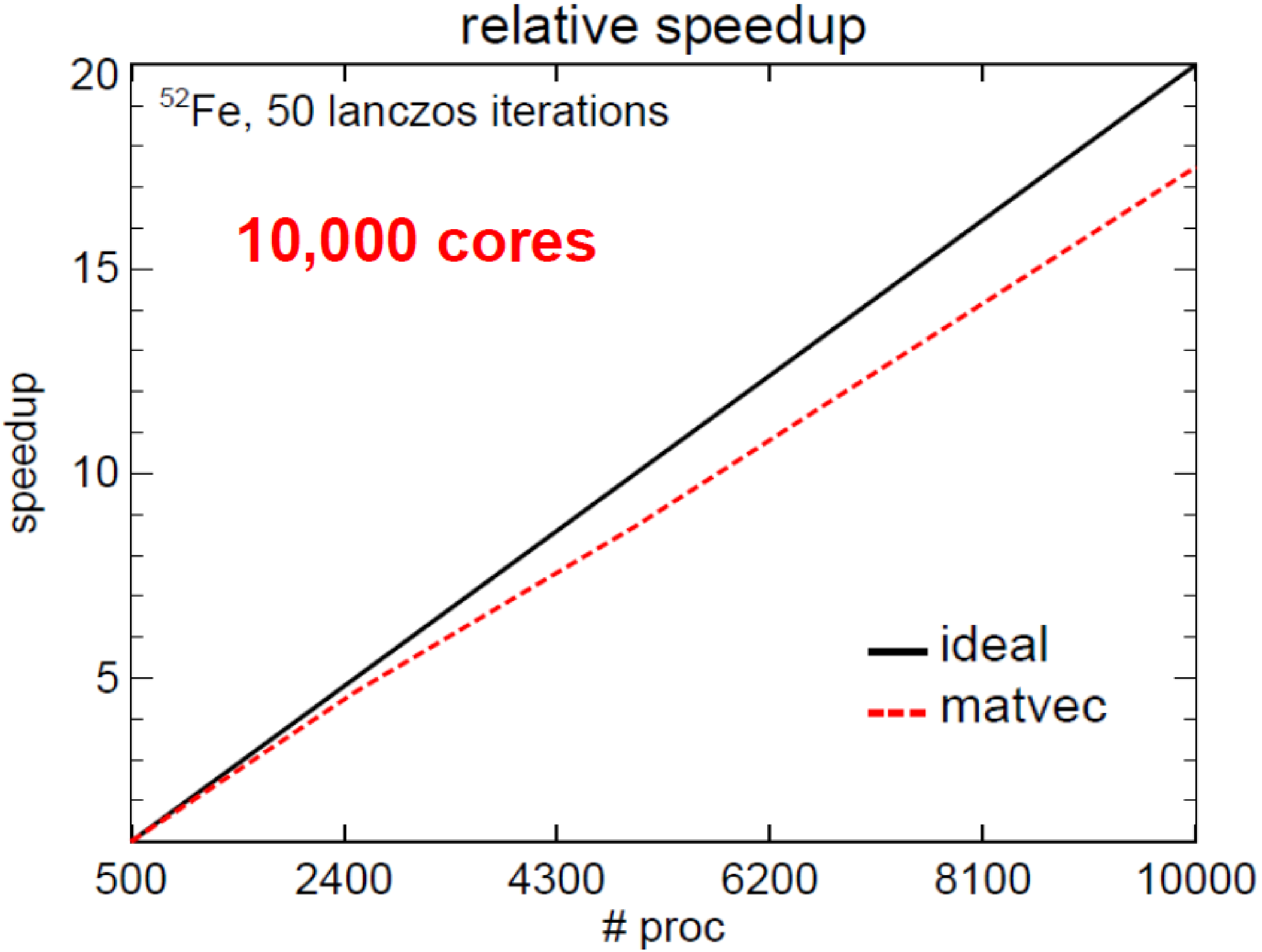


🟧 parallelized

🟩 Not parallelized

🟥 Communication overhead

Realistic example: Speedup of matrix vector multiplication in large scale shell-model calculations



Basic guidance for efficient parallelization:

(1) Increase the fraction of your program that can be parallelized (identify the most time consuming parts of your program and parallelize them). This could require modifying your intrinsic algorithm and code's organization

(2) Balance parallel workload

(3) Minimize time spent in communication

(4) Use simple arrays instead of user defined derived types

(5) Partition data. Distribute arrays and matrices – allocate specific memory for each MPI process

Parallelizing DO / FOR loops:

In almost all of the scientific and technical programs, the hot spots are likely to be found in DO / FOR loops.

Thus parallelizing DO / FOR loops is one of the most important challenges when you parallelize your program.

The basic technique of parallelizing DO / FOR loops is to distribute iterations among processors and to let each processor do its portion in parallel.

Usually, the computations within a DO / FOR loop involve arrays whose indices are associated with the loop variable. Therefore distributing iterations can often be regarded as dividing arrays and assigning chunks (and computations associated with them) to processors.

Block distribution:

p → number of processors

n → number of iterations

$n = p \times q + r$

reminder

quotient

Example:

n = 14, p = 4, q = 3, r = 2

Processors 0 r-1 are assigned q+1 iterations

the rest are assigned q iterations

$$n = r(q + 1) + (p - r)q$$

Iteration	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Rank	0	0	0	0	1	1	1	1	2	2	2	3	3	3

The para_range subroutine:

Computes the iteration range for each MPI process

FORTRAN implementation

```
subroutine para_range(n1, n2, nprocs, irank, ista, iend)
  integer(4) :: n1        ! Lowest value of iteration variable
  integer(4) :: n2        ! Highest value of iteration variable
  integer(4) :: nprocs ! # cores
  integer(4) :: irank     ! Iproc (rank)
  integer(4) :: ista      ! Start of iterations for rank iproc
  integer(4) :: iend      ! End of iterations for rank iproc
  iwork1 = ( n2 - n1 + 1 )  /  nprocs
  iwork2 = MOD(n2 - n1 + 1, nprocs)
  ista = irank * iwork1 + n1 + MIN(irank, iwork2)
  iend = ista + iwork1 - 1
  if ( iwork2 > irank ) iend = iend + 1
  return
end subroutine para_range
```

The para_range subroutine, con'd:

Computes the iteration range for each MPI process

C / C++ implementation

```
void para_range(int n1, int n2, int &nprocs, int &irank, int &ista, int &iend){
  int iwork1;
  int iwork2;
  iwork1 = ( n2 - n1 + 1 )  /  nprocs;
  iwork2 = ( ( n2 - n1 + 1 ) %  nprocs );
  ista = irank * iwork1 + n1 + min(irank, iwork2);
  iend = ista + iwork1 - 1;
  if ( iwork2 > irank ) iend = iend + 1;
}
```

Simple example: Sum up elements of an array (serial code)

Fortran

```
program main
implicit none
integer(4) :: i, sum
integer(4), parameter :: n = 1000
integer(4) :: a(n)
do i = 1, n
   a(i) = i
end do
sum = 0.0
do i = 1, n
   sum = sum + a(i)
end do
write(6,*) ’sum =’,sum
end program main
```

C++

```
#include <iostream>
#include <math.h>
using namespace std;
int main(){
  int i;
  int n = 1000;
  int a[n];
  int sum;
  for ( i = 1; i <= n; i++ ){
    a[i] = i;
  }
  sum = 0;
  for ( i = 1; i <= n; i++ ){
    sum = sum + a[i];
  }
  cout << "sum = " << sum <<
endl;
  return 0 ;
}
```

Sum up elements of an array (parallel code, Fortran)

```
program main
  implicit none
  include 'mpif.h'
  integer(4), parameter :: n = 1000
  integer(4) :: a(n)
  integer(4) :: i, ista, iend, sum, ssum, ierr, iproc, nproc
  call MPI_INIT(ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, iproc, ierr)
  call para_range(1, n, nproc, iproc, ista, iend)
  do i = ista, iend
     a(i) = i
  end do
  sum = 0.0
  do i = ista, iend
     sum = sum + a(i)
  end do
  call MPI_REDUCE(sum, ssum, 1, MPI_INTEGER,MPI_SUM, 0,MPI_COMM_WORLD, ierr)
  sum = ssum
  if ( iproc == 0 ) write(6,*)'sum =', sum
  call MPI_FINALIZE(ierr)
end program main
```

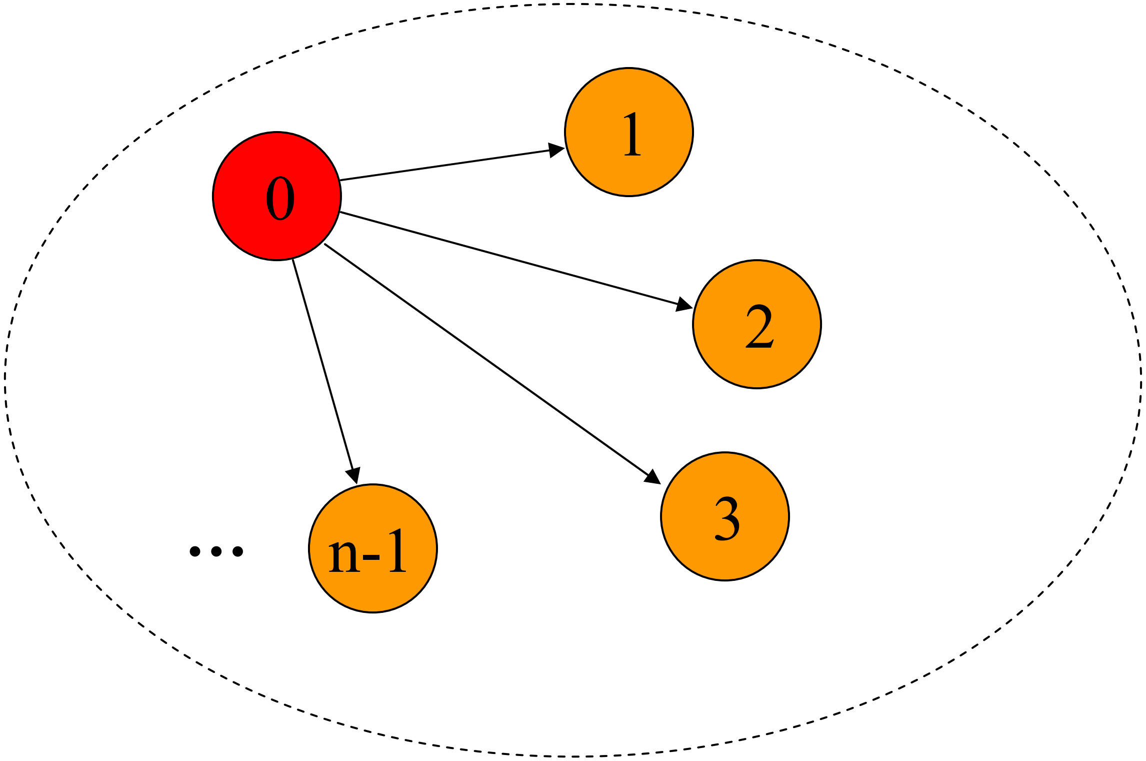
Sum up elements of an array (parallel code, C++)

```
#include <iostream>
#include <mpi.h>
using namespace std;
int main(int argc, char** argv){
  int i;
  int n = 1000;
  int a[n];
  int sum, ssum, iproc, nproc, ista, iend;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&iproc);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  para_range(1,n,nproc,iproc,ista,iend);
  for ( i = ista; i <= iend; i++ ){
    a[i-1] = i;
  }
  sum = 0;
  for ( i = ista; i <= iend; i++ ){
    sum = sum + a[i-1];
  }
  MPI_Reduce(&sum,&ssum,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD);
  sum = ssum;
  if ( iproc == 0 ){
    cout << "sum = " << sum << endl;
  }
  MPI_Finalize();
  return 0;
}
```

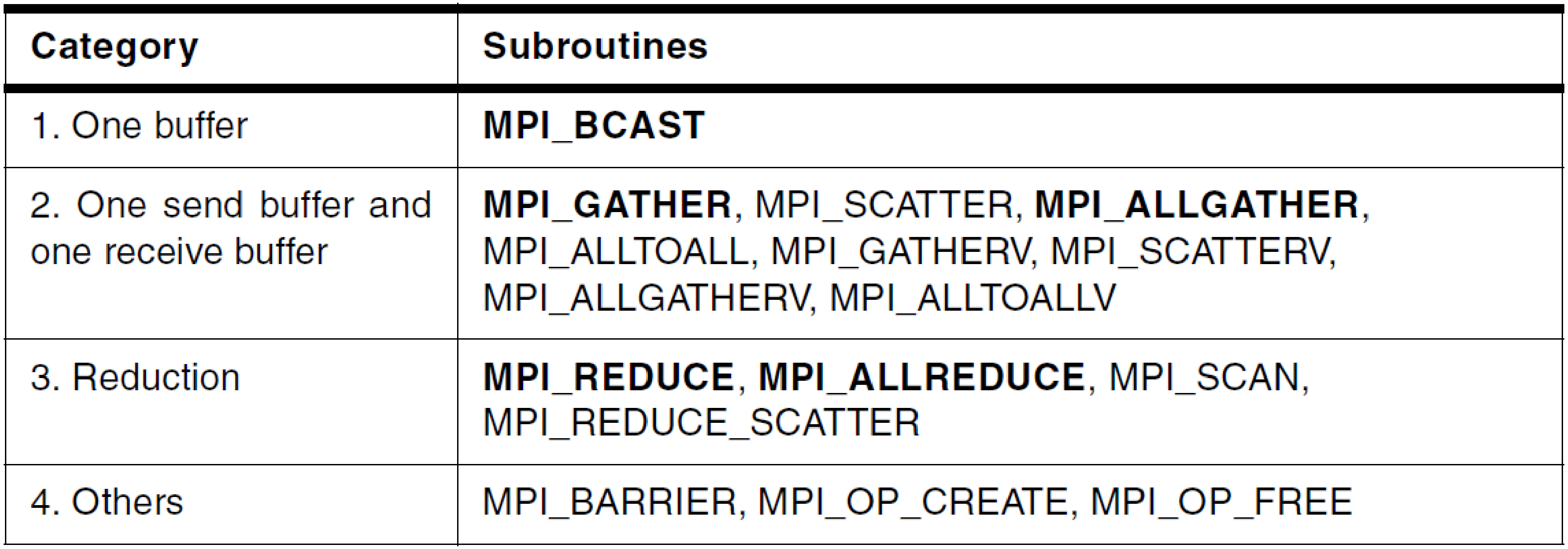
Collective communication:

Collective communication allows you to exchange data among a group of processes. The communicator argument in the collective communication subroutine calls specifies which processes are involved in the communication.

MPI_COMM_WORLD



MPI Collective Communication Subroutines:



Category	Subroutines
1. One buffer	**MPI_BCAST**
2. One send buffer and one receive buffer	**MPI_GATHER**, MPI_SCATTER, **MPI_ALLGATHER**, MPI_ALLTOALL, MPI_GATHERV, MPI_SCATTERV, MPI_ALLGATHERV, MPI_ALLTOALLV
3. Reduction	**MPI_REDUCE**, **MPI_ALLREDUCE**, MPI_SCAN, MPI_REDUCE_SCATTER
4. Others	MPI_BARRIER, MPI_OP_CREATE, MPI_OP_FREE

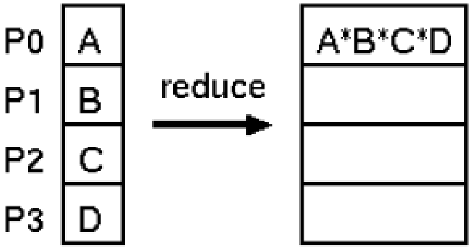
* The subroutines printed in boldface are used most frequently.

MPI_REDUCE:

Usage:

Fortran: CALL MPI_REDUCE(sendbuf, recvbuf, count, datatype, op, root, comm, ierror)

C / C++: MPI_Reduce(&sendbuf, &recvbuf, count, datatype, op, root, comm)

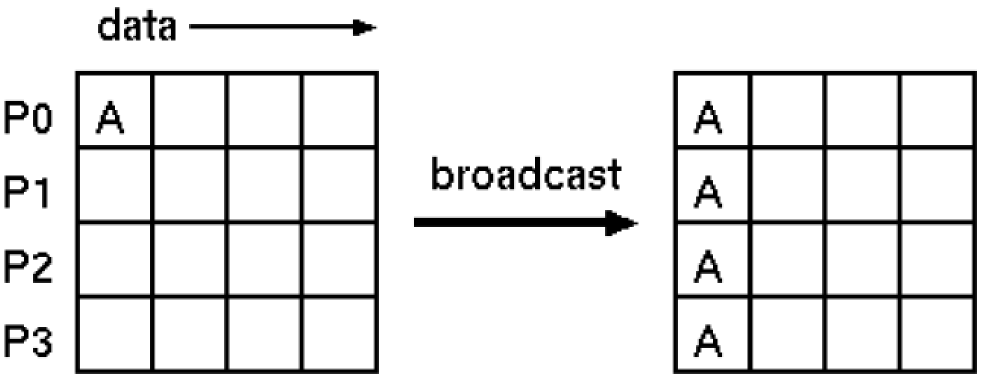


Operation	Data Type
MPI_SUM, MPI_PROD	MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_COMPLEX
MPI_MAX, MPI_MIN	MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION
MPI_MAXLOC, MPI_MINLOC	MPI_2INTEGER, MPI_2REAL, MPI_2DOUBLE_PRECISION
MPI_LAND, MPI_LOR, MPI_LXOR	MPI_LOGICAL
MPI_BAND, MPI_BOR, MPI_BXOR	MPI_INTEGER, MPI_BYTE

MPI_BCAST:

Usage:

Fortran: CALL MPI_BCAST(sendbuf, count, datatype, root, comm, ierror)
C / C++: MPI_Bcast(&sendbuf, count, datatype, root, comm)



Typical use: One process reads input data from disk and broadcasts data to all MPI processes.

MPI_BCAST example (Fortran)

```
program main
  implicit none
  include 'mpif.h'
  integer(4), parameter :: n = 100
  integer(4) :: a(n)
  integer(4) :: i
  integer(4) :: ierr
  integer(4) :: iproc
  integer(4) :: nproc
  call MPI_INIT(ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, iproc, ierr)
  if ( iproc == 0 ) then
     do i = 1, n
        a(i) = i
     end do
  end if
  call MPI_BCAST(a,n,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)
  call MPI_FINALIZE(ierr)
end program main
```

MPI_BCAST example (C++)

```
#include <iostream>
#include <mpi.h>
using namespace std;
int main(int argc, char** argv){
  int n = 100;
  int a[n];
  int i;
  int iproc;
  int nproc;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&iproc);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  if ( iproc == 0 ){
    for ( i = 1; i <= n; i++ ){
      a[i-1] = i;
    }
  }
  MPI_Bcast(&a,n,MPI_INTEGER,0,MPI_COMM_WORLD);
  MPI_Finalize();
  return 0;
}
```

Cyclic distribution:

In cyclic distribution, the iterations are assigned to processes in a round-robin fashion.

```
  DO i = n1, n2
       computation
  ENDDO
```

→

```
DO i = n1+iproc, n2, nproc
     computation
ENDDO
```

Example: Distributing **14** iterations over **4** cores in round-robin fashion

Iteration	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Rank	0	1	2	3	0	1	2	3	0	1	2	3	0	1

Sum up elements of an array (cyclic distribution, Fortran)

```
program main
  implicit none
  include 'mpif.h'
  integer(4), parameter :: n = 1000
  integer(4) :: a(n)
  integer(4) :: i, sum, ssum, ierr, iproc, nproc
  call MPI_INIT(ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, iproc, ierr)
  do i = 1 + iproc, n, nproc
     a(i) = i
  end do
  sum = 0.0
  do i = 1 + iproc, n, nproc
     sum = sum + a(i)
  end do
  call MPI_REDUCE(sum, ssum, 1, MPI_INTEGER,MPI_SUM, 0,MPI_COMM_WORLD, ierr)
  sum = ssum
  if ( iproc == 0 ) write(6,*)'sum =', sum
  call MPI_FINALIZE(ierr)
end program main
```

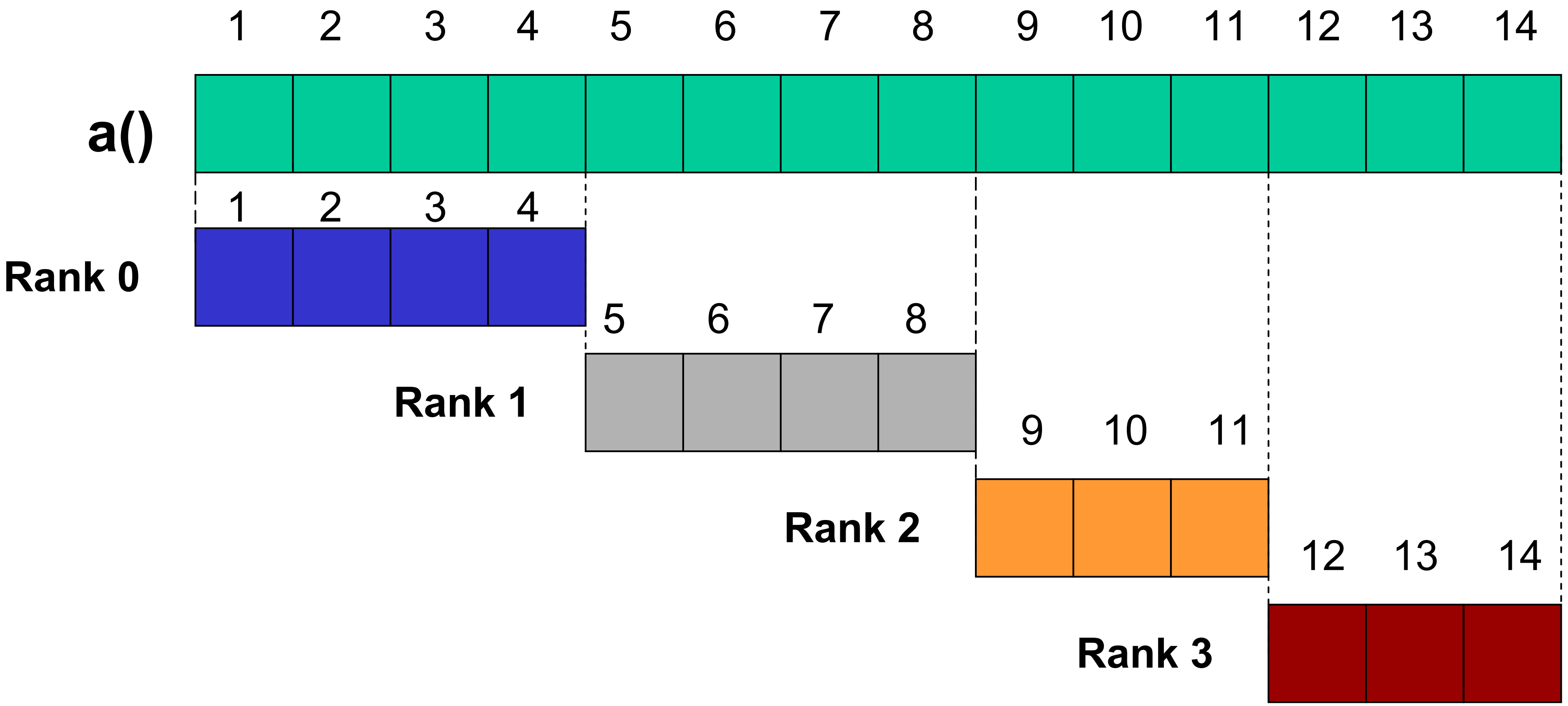
Sum up elements of an array (cyclic distribution, C++)

```
#include <iostream>
#include <mpi.h>
using namespace std;
int main(int argc, char** argv){
  int n = 1000;
  int a[n];
  int i, sum, ssum, iproc, nproc, ista, iend;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&iproc);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  for ( i = 1 + iproc; i <= n; i = i + nproc ){
    a[i-1] = i;
  }
  sum = 0;
  for ( i = 1 + iproc; i <= n; i = i + nproc ){
    sum = sum + a[i-1];
  }
  MPI_Reduce(&sum,&ssum,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD);
  sum = ssum;
  if ( iproc == 0 ){
    cout << "sum = " << sum << endl;
  }
  MPI_Finalize();
  return 0;
}
```

Shrinking arrays:

Extremely important for efficient memory management!!!

Block distribution of 14 iterations over 4 cores. Each MPI process needs only part of the array a()



Shrinking arrays, Fortran example:

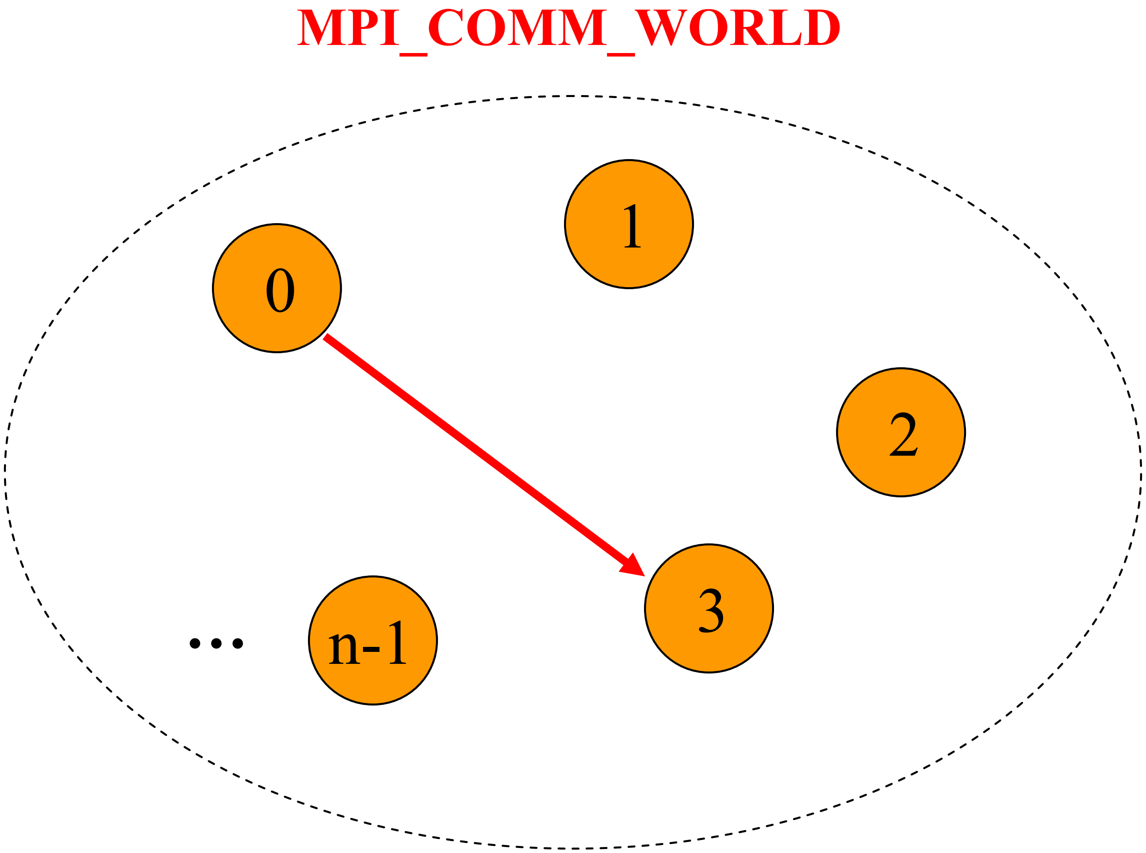
```
program main
  implicit none
  include 'mpif.h'
  integer(4) :: i, ista, iend, sum, ssum, ierr, iproc, nproc
  integer(4), parameter :: n = 1000
  integer(4), allocatable :: a(:)
  call MPI_INIT(ierr)
  call MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
  call MPI_COMM_RANK(MPI_COMM_WORLD, iproc, ierr)
  call para_range(1, n, nproc, iproc, ista, iend)
  if ( .not. allocated(a) ) allocate( a(ista:iend) )
  do i = ista, iend
     a(i) = i
  end do
  sum = 0.0
  do i = ista, iend
     sum = sum + a(i)
  end do
  call MPI_REDUCE(sum, ssum, 1, MPI_INTEGER,MPI_SUM, 0,MPI_COMM_WORLD, ierr)
  sum = ssum
  if ( iproc == 0 ) write(6,*)'sum =', sum
  if ( allocated(a) ) deallocate(a)
  call MPI_FINALIZE(ierr)
end program main
```

C++

```cpp
#include <iostream>
#include <mpi.h>
#include <new>
using namespace std;
int main(int argc, char** argv){
  int i, sum, ssum, iproc, nproc, ista, iend, loc_dim;
  int n = 1000;
  int *a;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&iproc);
  MPI_Comm_size(MPI_COMM_WORLD,&nproc);
  para_range(1,n,nproc,iproc,ista,iend);
  loc_dim = iend - ista + 1;
  a = new int[loc_dim];
  for ( i = 0; i < loc_dim; i++ ){
    a[i] = i + ista;
  }
  sum = 0;
  for ( i = 0; i < loc_dim; i++ ){
    sum = sum + a[i];
  }
  MPI_Reduce(&sum,&ssum,1,MPI_INTEGER,MPI_SUM,0,MPI_COMM_WORLD);
  sum = ssum;
  if ( iproc == 0 ){
    cout << "sum = " << sum << endl;
  }
  delete [] a;
  MPI_Finalize();
  return 0;
}
```

Point-to-point communication:

Point-to-point communication allows you to exchange data between any two MPI processes.



Blocking communication:

In blocking communication, the program will not return from the subroutine call until the copy from/to the system buffer has completed.

Fortran

```
if ( iproc == 0 ) then
   call MPI_SEND(sendbuf, icount, MPI_REAL8, 1, itag, MPI_COMM_WORLD, ierr)
else if ( iproc == 1 ) then
   call MPI_RECV(recvbuf, icount, MPI_REAL8, 0, itag, MPI_COMM_WORLD, istatus, ierr)
end if
```

C/C++

```
if ( iproc == 0 ) {
   MPI_Send(sendbuf, icount, MPI_REAL8, 1, itag, MPI_COMM_WORLD);
}
else if ( iproc == 1 ) {
   MPI_Recv(recvbuf, icount, MPI_REAL8, 0, itag, MPI_COMM_WORLD, istatus);
}
```

MPI_Send and MPI_Recv

Fortran

```
CALL MPI_SEND(buffer, count, datatype, destination, tag, communicator, ierr)
CALL MPI_RECV(buffer, count, datatype, source, tag, communicator, status ierr)
```

C/C++

```
MPI_Send(&buffer, count, datatype, destination, tag, communicator)
MPI_Recv(&buffer, count, datatype, source, tag, communicator, &status)
```

Buffer: Data to be sent / received (e.g., array)
Count: Number of data elements
Datatype: Type of data, for example MPI_INT, MPI_REAL8, etc
Destination: Rank of destination MPI process
Tag: Message label
Communicator: Set of MPI processes used for communication
Status: The status object
Ierr: Returned error code (Fortran only)

Non-blocking communication:

In non-blocking communication, the program will return immediately from the subroutine call and will not wait for the copy from / to the system buffer to be completed.

Fortran

```
if ( iproc == 0 ) then
   call MPI_ISEND(sendbuf, icount, MPI_REAL8, 1, itag, MPI_COMM_WORLD, ireq, ierr)
else if ( iproc == 1 ) then
   call MPI_IRECV(recvbuf, icount, MPI_REAL8, 0, itag, MPI_COMM_WORLD, ireq, ierr)
end if
call MPI_WAIT(ireq, istatus, ierr)
```

C/C++

```
if ( iproc == 0 ) {
   MPI_Isend(sendbuf, icount, MPI_REAL8, 1, itag, MPI_COMM_WORLD, ireq);
}
else if ( iproc == 1 ) {
   MPI_Irecv(recvbuf, icount, MPI_REAL8, 0, itag, MPI_COMM_WORLD, ireq);
}
MPI_Wait(ireq, istatus);
```

Compiling and running MPI codes on Odyssey:

Edit source code:

```
$vi mpi_program.f90
```

Compile:

Load one of the available MPI software modules, e.g.,

```
$module load hpc/openmpi-1.6.2_intel-13.0.079
```

```
Fortran 77: $mpif77 –o mpi_program mpi_program.f77
Fortran 90: $mpif90 –o mpi_program.x mpi_program.f90
C:          $mpicc –o mpi_program.x mpi_program.c
C++:        $mpicxx –o mpi_program.x mpi_program.cpp
```

Execute:

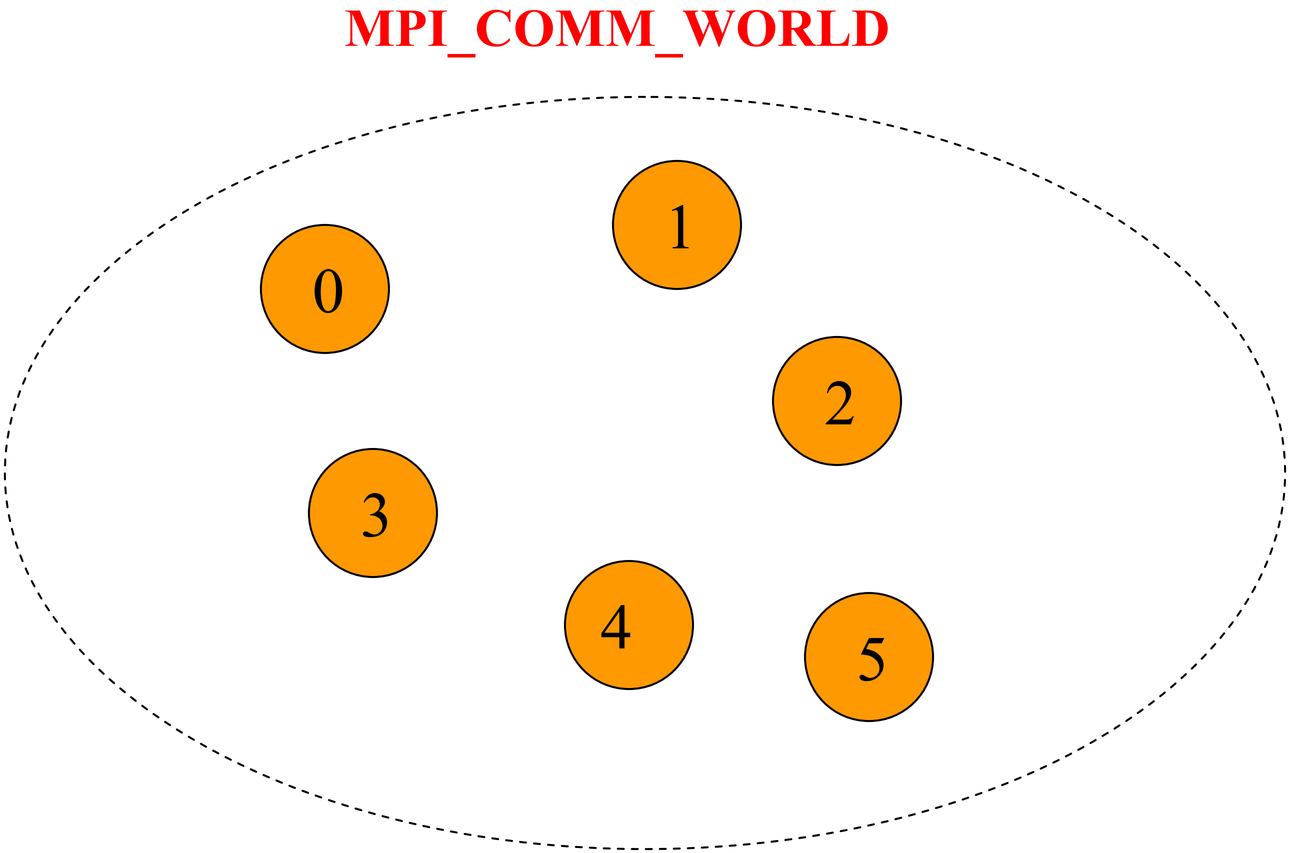
```
$bsub < mpi_program.run
```

LSF batch-job submission script:

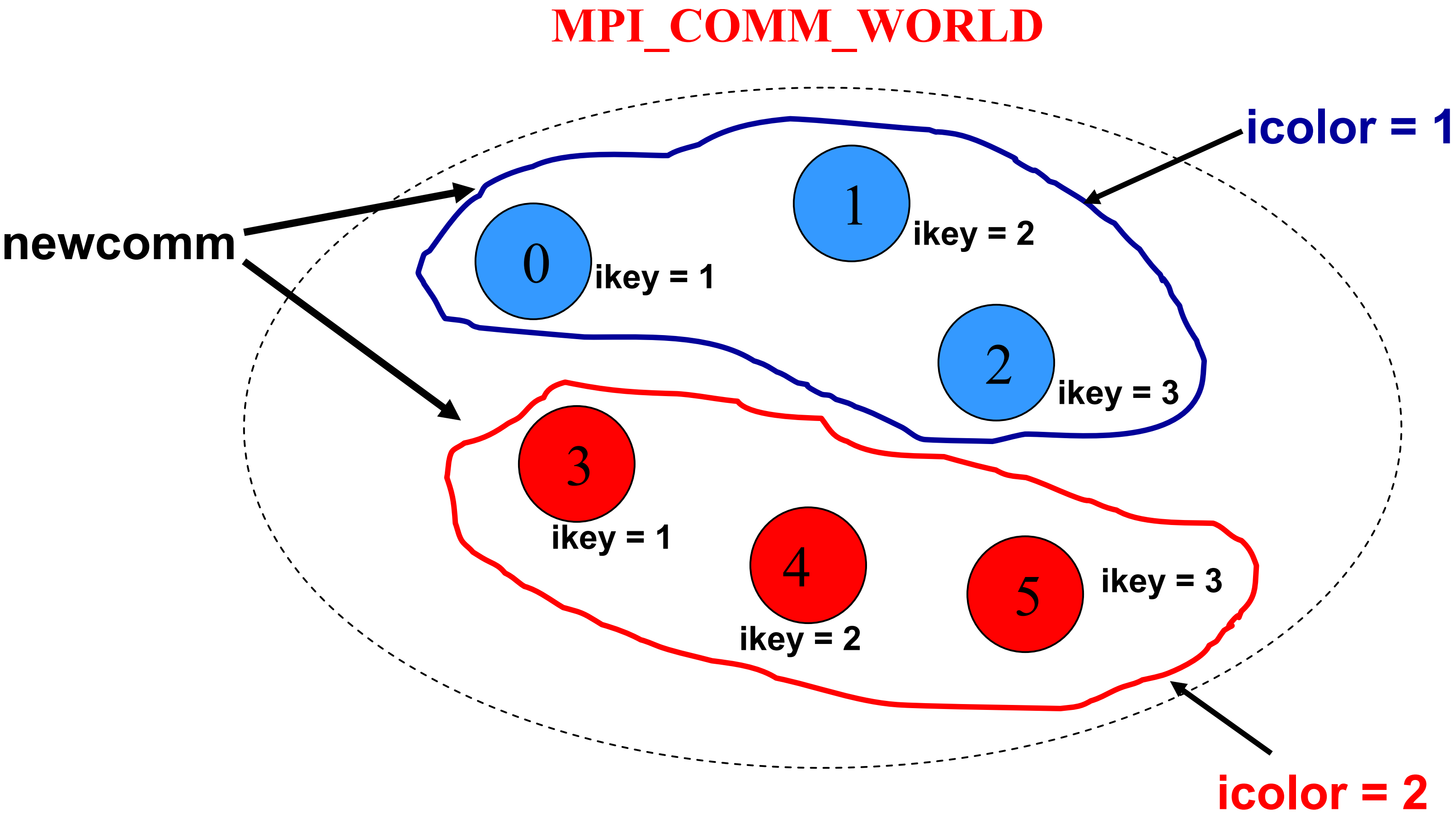
(mpi_program.run)

```
#!/bin/bash
#BSUB -n 8
#BSUB -J test
#BSUB -o mpi_program.out
#BSUB -e mpi_program.err
#BSUB -a openmpi
#BSUB -R "span[ptile=8]“
#BSUB -R "rusage[mem=36000]"
#BSUB –q normal_parallel
mpirun.lsf ./mpi_program.x
```

Communication groups:

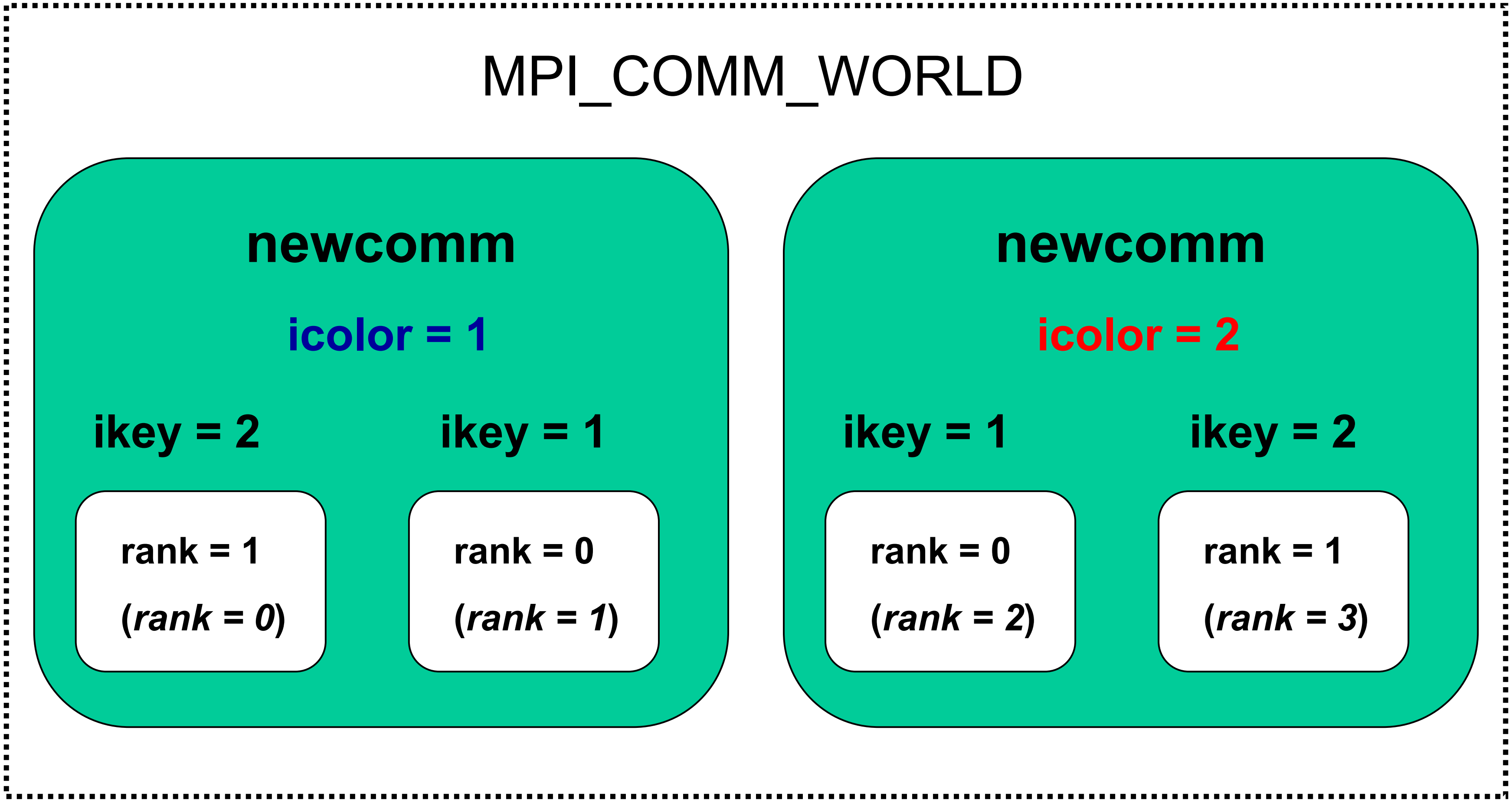


Communication groups:



Communication groups, cont'd:

Fortran: CALL MPI_COMM_SPLIT(comm, color, key, newcomm, ierr)
C/C++: MPI_Comm_split(comm, color, key, &newcomm)



Communication groups, Fortran example:

```
program comm_groups
 implicit none
 include 'mpif.h'
 integer(4) :: i, ierr, iproc, nproc, newcomm, new_iproc, new_nproc, icolor,
ikey
 call MPI_INIT(ierr)
 call MPI_COMM_SIZE(MPI_COMM_WORLD, nproc, ierr)
 call MPI_COMM_RANK(MPI_COMM_WORLD, iproc, ierr)
 if ( iproc == 0 ) then
    icolor = 1
    ikey = 2
 else if ( iproc == 1 ) then
    icolor = 1
    ikey = 1
 else if ( iproc == 2 ) then
    icolor = 2
    ikey = 1
 else if ( iproc == 3 ) then
    icolor = 2
    ikey = 2
 end if
 call MPI_COMM_SPLIT(MPI_COMM_WORLD, icolor, ikey, newcomm, ierr)
 call MPI_COMM_SIZE(newcomm, new_nproc, ierr)
 call MPI_COMM_RANK(newcomm, new_iproc, ierr)
call MPI_FINALIZE(ierr)
end program comm_groups
```

C++

```
#include <iostream>
#include <mpi.h>
#include <new>
using namespace std;
int main(int argc, char** argv){
 int i, iproc, nproc, new_iproc, new_nproc, ikey, icolor;
 MPI_Comm newcomm;
 MPI_Init(&argc,&argv);
 MPI_Comm_rank(MPI_COMM_WORLD,&iproc);
 MPI_Comm_size(MPI_COMM_WORLD,&nproc);
 if ( iproc == 0 ){
  icolor = 1;
  ikey = 2;
 }
 else if ( iproc == 1 ){
  icolor = 1;
  ikey = 1;
 }
 else if ( iproc == 2 ){
  icolor = 2;
  ikey = 1;
 }
 else if ( iproc == 3 ){
  icolor = 2;
  ikey = 2;
 }
 MPI_Comm_split(MPI_COMM_WORLD, icolor, ikey, &newcomm);
 MPI_Comm_rank(newcomm, &new_iproc);
 MPI_Comm_size(newcomm, &new_nproc);
 MPI_Finalize();
 return 0;
}
```

Parallel Input - Output (I/O)

Traditionally, most scientific applications have been compute bound. That is, the time to completion has been dominated by the time needed to perform computation. Increasingly, however, more and more scientific applications are becoming I/O bound. That is, the time to completion is being dominated by the time required to read data from disk and/or write data to disk. Increasingly, disk I/O time is becoming the limiting factor on problem sizes that can be computed. As processor and network performance continues to increase, this problem will only worsen.

Traditional methods for writing to disk:

(1) One master process handles all I/O

This solution works fine for application with a relatively small amount of I/O required. **However, as the amount of data increases, this solution does not scale and an I/O bottleneck may develop.**

(2) Each process writes/reads its own file

This can have very good performance if the process specific files are written to disks local to each processor. However, if this output is needed for future use, it must be gathered in some way. Additionally, if the output is used for check pointing, any restart requires the same processors to be used as the ones that wrote out the data.

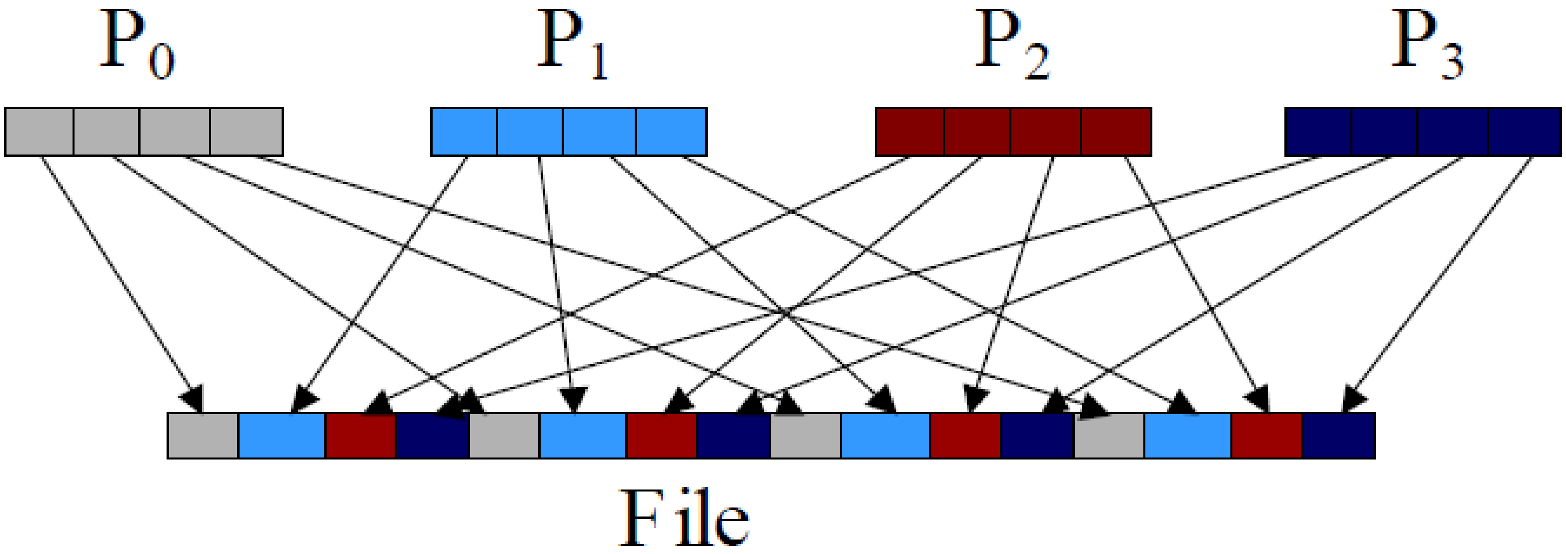
(3) Each process accesses disk at the same time

Though it looks like all the disk access is being done at once, if the underlying file system does not support simultaneous disk access (true for traditional Unix style I/O), all the disk accesses will be sequential. This results in poor performance and a bottleneck may develop as the problem size increases.

None of these is satisfactory for large problems

MPI-I/O:

- ❑ Large and complex, but very important topic
- ❑ Needs its own presentation
- ❑ Solves many issues in programming parallel I/O
- ❑ Check Web for references and tutorials



Python MPI:

Various implementations:

- ❑ Pypar – Parallel Python, http://code.google.com/p/pypar
- ❑ mpi4Py – MPI for Python, http://mpi4py.scipy.org

Python MPI codes are usually slower than Fortran and C/C++ MPI codes. When speed matters, Fortran and C/C++ are the languages of choice.

There are no good and bad computer languages. There are only good and bad computer languages for specific purpose.

Pypar example code:

```
#!/usr/bin/env python
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Parallel Python test program: PyPar
#++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
import pypar

nproc = pypar.size()                   # Size of communicator
iproc = pypar.rank()                   # Ranks in communicator
inode = pypar.get_processor_name()     # Node where this MPI process runs

if iproc == 0: print "This code is a test for PyPar."

for i in range(0,nproc):
   pypar.barrier()
   if iproc == i:
      print 'Rank %d out of %d' % (iproc,nproc)

pypar.finalize()
```

mpi4Py example code:

```
#! /usr/bin/env python
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
# Parallel Python test program: mpi4py
#+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
from mpi4py import MPI

nproc = MPI.COMM_WORLD.Get_size()   # Size of communicator
iproc = MPI.COMM_WORLD.Get_rank()   # Ranks in communicator
inode = MPI.Get_processor_name()    # Node where this MPI process runs

if iproc == 0: print "This code is a test for mpi4py."

for i in range(0,nproc):
   MPI.COMM_WORLD.Barrier()
   if iproc == i:
      print 'Rank %d out of %d' % (iproc,nproc)

MPI.Finalize()
```

Performance Considerations:

- ❑ Important to understand where “bottlenecks” occur.

- ❑ More abstraction usually means worse performance.

- ❑ Speedup is limited by the time needed for the serial portion of the program.

- ❑ Communication overhead: you cannot achieve perfect speedup even if all execution can be parallelized. Communication overhead will usually dominate for large count of MPI processes.

- ❑ Load balancing: a parallel program will only run as fast as the slowest rank. Workload should be distributed as evenly as possible across ranks. Often good serial algorithms perform poorly in parallel. Sometimes it is better to change algorithms entirely.

- ❑ May want to duplicate computational work rather than add communication overhead.

MPI Debugging:

In general tends to be difficult

Totalview is good for finding problems on a small scale. Has a lot of useful functionality

Resources:

http://rc.fas.harvard.edu/kb/high-performance-computing/totalview-debugging-parallel-applications

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

Slides and Codes from this workshop will be uploaded to:

http://rc.fas.harvard.edu/kb/training/

Contact information:

Harvard Research Computing Website:

http://rc.fas.harvard.edu

Email:

rchelp@fas.harvard.edu

plamenkrastev@fas.harvard.edu