ME964
High Performance Computing for Engineering Applications

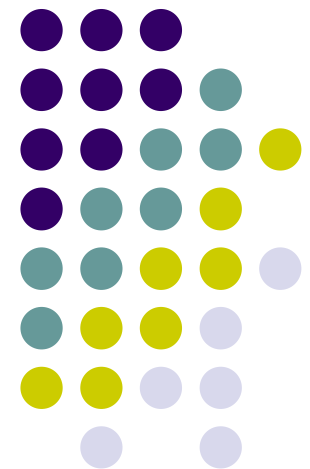
Parallel Computing with MPI

Wrap up, MPI Send/Receive

Collective Communications with MPI

MPI Examples

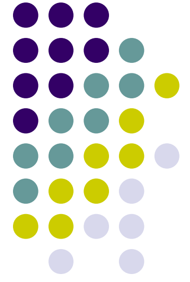
April 12, 2012



© Dan Negrut, 2012
ME964 UW-Madison

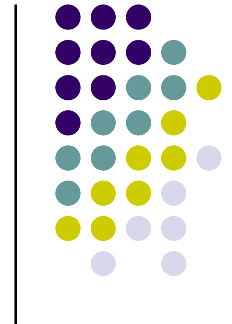
You know that uncertainty you feel today? It never goes away.
The question is, do you know how to make uncertainty your friend?
– David Brooks, journalist, The New York Times

Before We Get Started…



- Last lecture
  - Running and Debugging MPI code from Eclipse
  - MPI Send/Receive, the blocking versions
- Today
  - MPI Send/Receive, the non-blocking flavor
  - Collective communications: MPI_Broadcast, MPI_Reduce, MPI_Scatter
  - Look at a couple of examples to put things in context
- Other issues
  - Assignment 10 due on Sunday at 11:59 pm
  - Assignment 11 available as of this weekend. Due: Sunday, April 22 at 11:59 pm
  - Midterm Project due on Tuesday, April 17 at 11:59 pm
  - Midterm Exam on Tuesday
  - Q/A session for Midterm Exam: Monday, at 6 pm in room 1153ME (across the hallway)

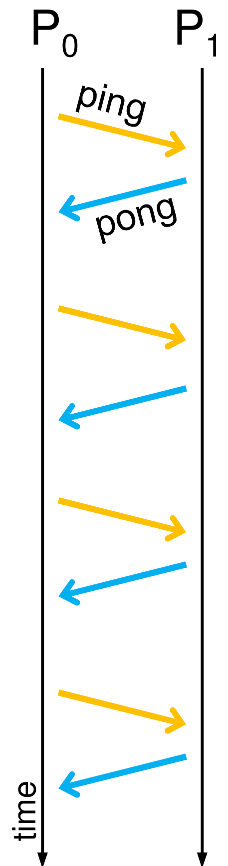
A Word on Assignment 11



- Write a program according to the time-line diagram:
  - process 0 sends a message to process 1 (ping)
  - after receiving this message, process 1 sends a message back to process 0 (pong)
- Repeat this ping-pong with a loop of length 50
- Add timing calls before and after the loop
- For timing purposes, you might want to use

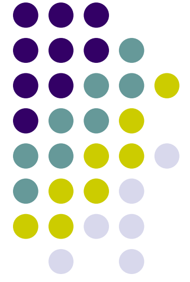
```
double MPI_Wtime();
```

- `MPI_Wtime` returns a wall-clock time in seconds
- At process 0, print out the transfer time in seconds
  - Might want to use a log scale



More on Timing

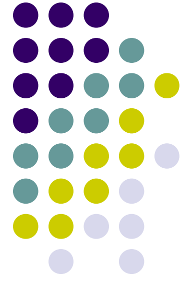
[Useful, assignment 10 and 11]



```
int main()
{
   double starttime, endtime;
   starttime = MPI_Wtime();
    ....  stuff to be timed  ...
   endtime   = MPI_Wtime();
   printf("That took %f seconds\n", endtime - starttime);
   return 0;
}
```

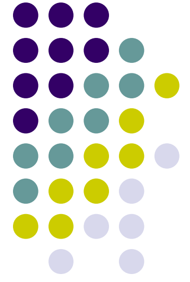
- Resolution is typically 1E-3 seconds
- Time of different processes might actually be synchronized, controlled by the variable `MPI_WTIME_IS_GLOBAL`

More on Timing
[Useful, assignment 10 and 11; Cntd.]



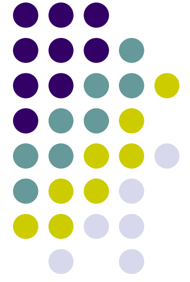
- Latency = transfer time for zero length messages
- Bandwidth = message size (in bytes) / transfer time

- Message transfer time and bandwidth change based on the nature of the MPI send operation
  - Standard send (`MPI_Send`)
  - Synchronous send (`MPI_Ssend`)
  - Buffered send (`MPI_Bsend`)
  - Etc.



Non-Blocking Communication

Non-blocking Send/Receive

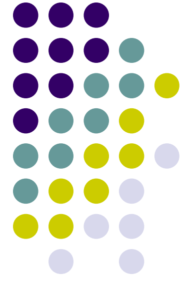


- Syntax

```
int MPI_Isend(void *buf, int count, MPI_Datatype datatype, int dest, int tag,
              MPI_Comm comm, MPI_Request *request);
```

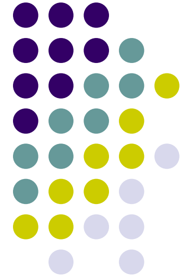
```
int MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source, int tag,
              MPI_Comm comm, MPI_Request *request);
```

Non-Blocking Communications: Motivation

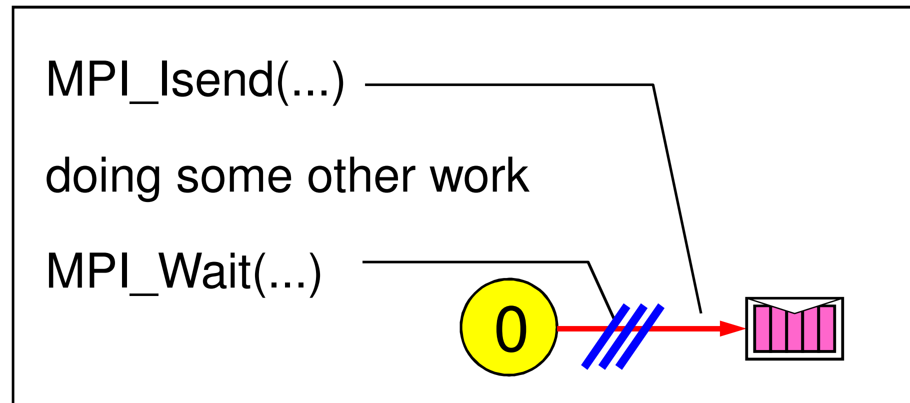


- Overlap communication with execution (just like w/ CUDA):
  - Initiate non-blocking communication
    - Returns **I**mmediately
    - Routine name starting with MPI_**I**…
  - Do some work
    - “latency hiding”
  - Wait for non-blocking communication to complete

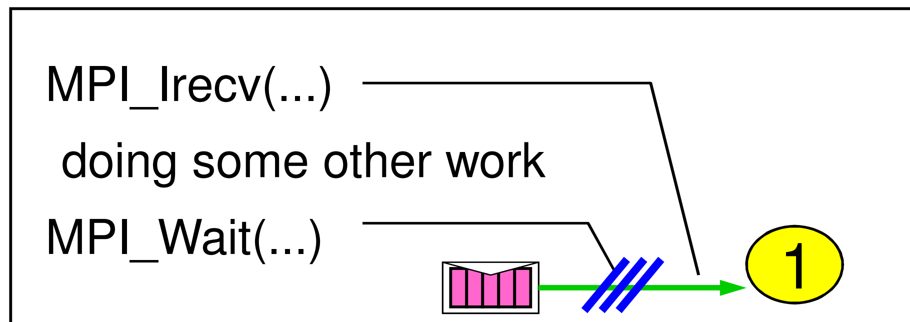
Non-Blocking Examples



- Non-blocking send

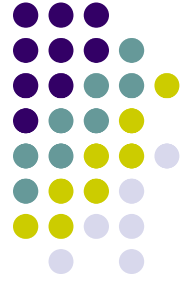


- Non-blocking receive



Non-Blocking Send/Receive

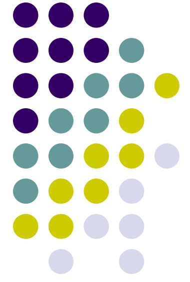
Some Tools of the Trade



- Call returns immediately. Therefore, user must worry whether …
  - Data to be sent is out of the send buffer before trampling on the buffer
  - Data to be received has finished arriving before using the content of the buffer

- Tools that come in handy:
  - For sends and receives in flight
    - `MPI_Wait` – blocking - you go synchronous
    - `MPI_Test` – non-blocking - returns quickly with status information
  - Check for existence of data to receive
    - Blocking: `MPI_Probe`
    - Non-blocking: `MPI_Iprobe`

Waiting for isend/ireceive to Complete



- Waiting on a single send

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

- Waiting on multiple sends (get status of all)
  - Till all complete, as a barrier

```
int MPI_Waitall(int count, MPI_Request *requests, MPI_Status *statuses);
```

  - Till at least one completes

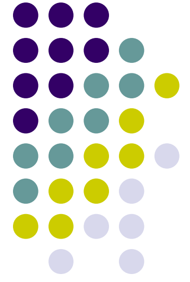
```
int MPI_Waitany(int count, MPI_Request *requests,int *index, MPI_Status *status);
```

  - Helps manage progressive completions

```
int MPI_Waitsome(int incount, MPI_Request *requests, int *outcount,
                 int *indices, MPI_Status *statuses);
```


[Alexander]→

MPI_Test...



- Flag true means completed

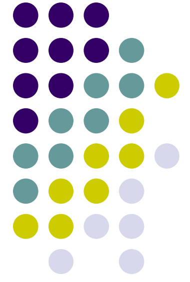
```
int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status);

int MPI_Testall(int count, MPI_Request *requests, int *flag, MPI_Status *statuses);

int MPI_Testany(int count, MPI_Request *requests, int *index, int *flag,
                MPI_Status *status);
```

- Like a non blocking MPI_Waitsome

```
int MPI_Testsome(int incount, MPI_Request *requests, int *outcount, int *indices,
                 MPI_Status *statuses);
```


[Alexander]→

Probe to Receive



- Probes yield incoming size

- Blocking Probe, wait till match

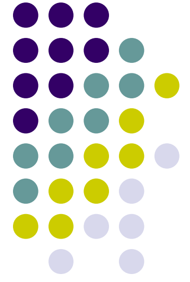
```
int MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status);
```

- Non Blocking Probe, flag true if ready

```
int MPI_Iprobe(int source,int tag,MPI_Comm comm,int *flag,MPI_Status *status);
```

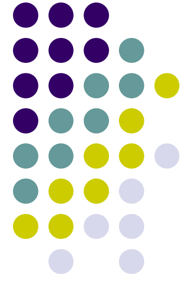

[Alexander]→

The Need for MPI_Probe and MPI_Iprobe

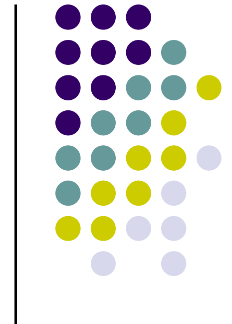


- The MPI_PROBE and MPI_IPROBE operations allow incoming messages to be checked for, without actually receiving them
- The user can then decide how to receive them, based on the information returned by the probe (basically, the information returned by status)
- In particular, the user may allocate memory for the receive buffer, according to the length of the probed message

MPI Point-to-Point Communication ~Take Away Slide~

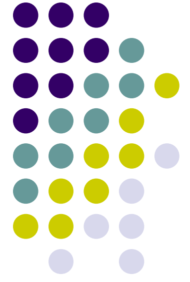


- Two <u>types</u> of communication:
  - Blocking:
    - Safe to change content of buffer holding on to data in the MPI send call
  - Non-blocking:
    - Be careful with the data in the buffer, since you might step on/use it too soon
- MPI provides four <u>modes</u> for these two types
  - standard, synchronous, buffered, ready



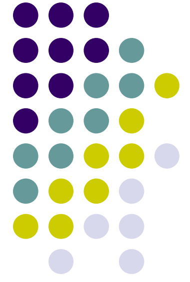
Collective Communication

Collective Communication



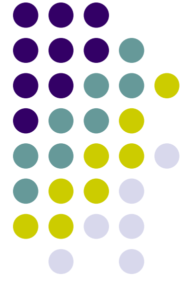
- Communications involving a group of processes
- Must be called by all processes in a communicator
- Types of Collective Communication (three of them):
  - Global Synchronization (barrier synchronization)
  - Global Communication (broadcast, scatter, gather, etc.)
  - Global Operations (sum, global maximum, etc.)

Characteristics of Collective Communication



- Optimized routines involving a group of processes
- Collective action over a communicator
  - That is, all processes must call the collective routine
- Inter-process synchronization may or may not occur
- All collective operations are blocking
- No tags
- Receive buffers must have exactly the same size as send buffers

Barrier Synchronization

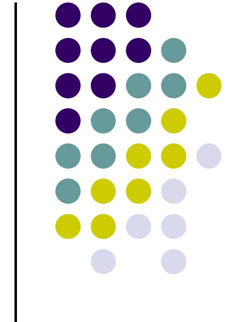


- Syntax:

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

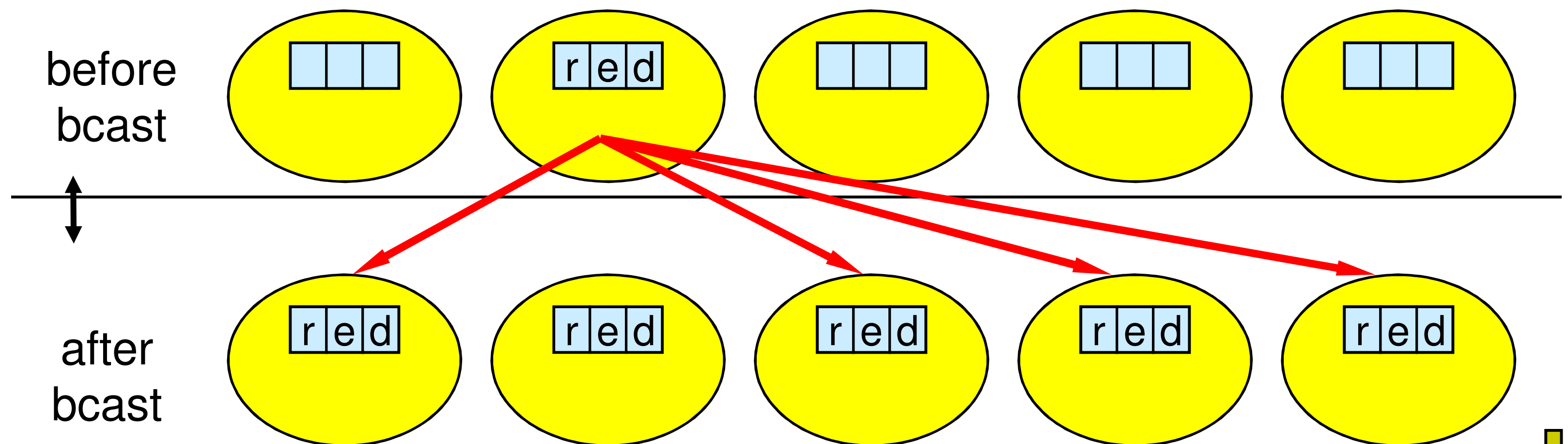
- `MPI_Barrier` not needed that often:
  - All synchronization is done automatically by the data communication
    - A process cannot continue before it has the data that it needs
  - If used for debugging
    - Remember to remove for production release

Broadcast



- Function prototype:

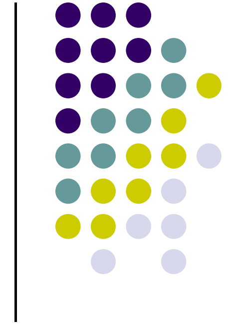
```
int MPI_Bcast( void *buf, int count, MPI_Datatype datatype, int root, MPI_Comm comm);
```



- rank of the sending process (i.e., root process)
- must be given identically by all processes

[ICHEC]→

MPI_Bcast



data (buffer)

processes

A_0					

broadcast

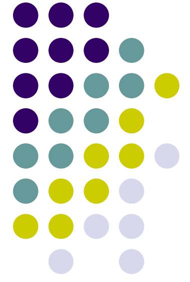
data (buffer)

processes

A_0					
A_0					
A_0					
A_0					
A_0					
A_0					

A_0 : any chunk of contiguous data described with MPI_Datatype and count

MPI_Bcast

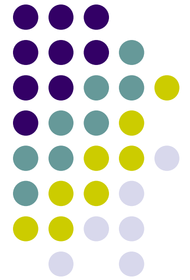


```
int MPI_Bcast (void *buffer, int count, MPI_Datatype type, int root, MPI_Comm comm);
```

INOUT : `buffer` (starting address, as usual)
IN : `count` (number of entries in buffer)
IN : `type` (can be user-defined)
IN : `root` (rank of broadcast root)
IN : `com` (communicator)

- Broadcasts message from `root` to all processes (including `root`)
- `com` and `root` must be identical on all processes
- On return, contents of `buffer` is copied to all processes in `com`

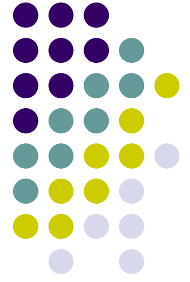
Example: MPI_Bcast



- Read a parameter file on a single processor and send data to all processes

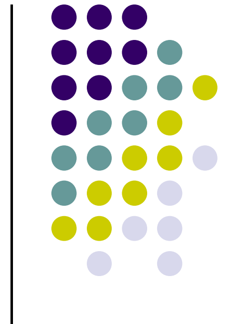
```
#include "mpi.h"
#include <assert.h>
#include <stdlib.h>

int main(int argc, char **argv){
    int mype, nprocs;
    float data = -1.0;
    FILE *file;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &mype);

    if( mype==0 ) {
        char input[100];
        file = fopen("data1.txt", "r");
        assert (file != NULL);
        fscanf(file, "%s\n", input);
        data = atof(input);
    }
    printf("data before: %f\n", data);
    MPI_Bcast(&data, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
    printf("data after: %f\n", data);

    MPI_Finalize();
}
```

Example: MPI_Bcast
[Output]



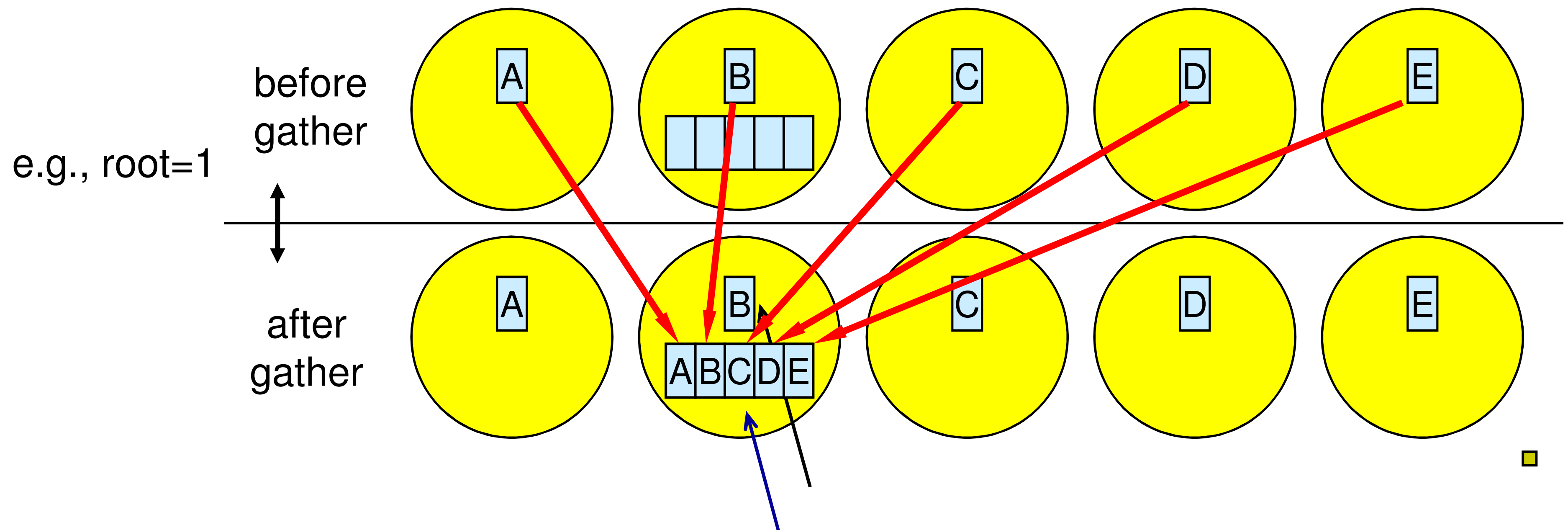
```
[negrut@euler CodeBits]$ qsub -I -l nodes=8:ppn=4,walltime=5:00
qsub: waiting for job 16114.euler to start
qsub: job 16114.euler ready

[negrut@euler17 CodeBits]$ mpicxx testMPI.cpp
[negrut@euler17 CodeBits]$ mpiexec -np 4 a.out
data before: -1.000000
data before: -1.000000
data before: -1.000000
data before: 23.330000
data after: 23.330000
data after: 23.330000
data after: 23.330000
data after: 23.330000
```

Gather



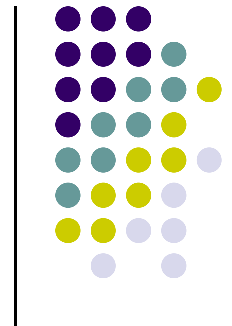
- Function Prototype

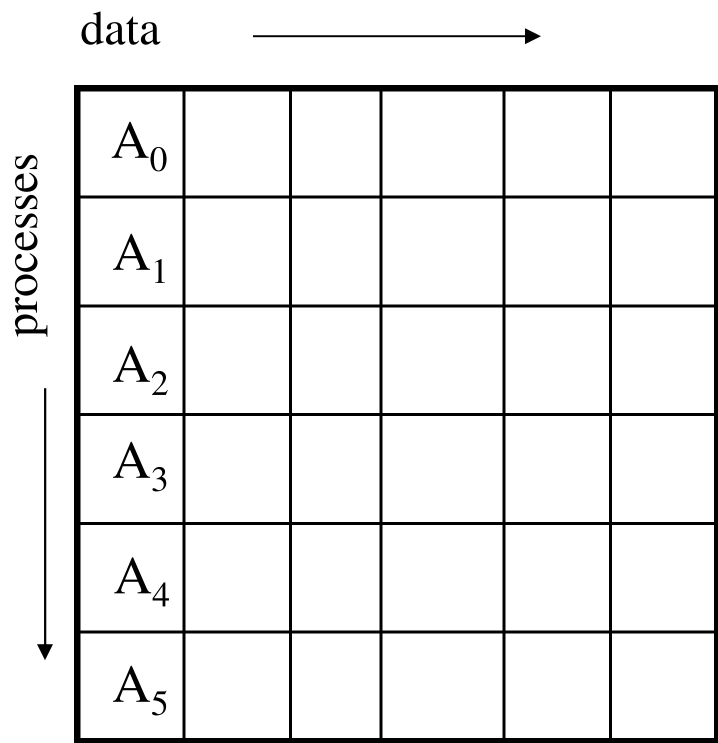
```
int MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
               int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```



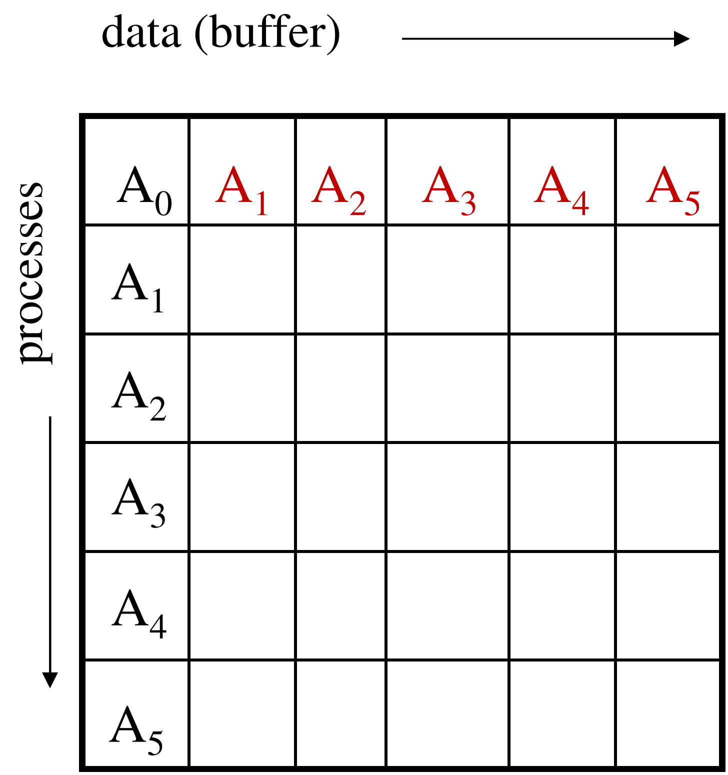
[ICHEC]→

MPI_Gather



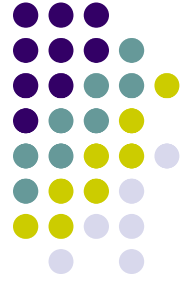


Gather



[A. Siegel]→

MPI_Gather

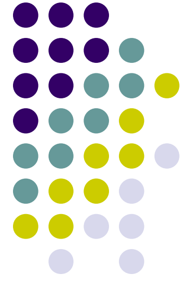


```
int MPI_Gather (void *sendbuf, int sendcount, MPI_Datatype sendtype,
void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

- IN `sendbuf` (starting address of send buffer)
- IN `sendcount` (number of elements in send buffer)
- IN `sendtype` (type)
- OUT `recvbuf` (address of receive buffer)
- IN `recvcount` (n-elements **for any single receive**)
- IN `recvtype` (data type of recv buffer elements)
- IN `root` (rank of receiving process)
- IN `comm` (communicator)

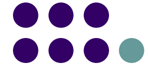
[A. Siegel]→

MPI_Gather



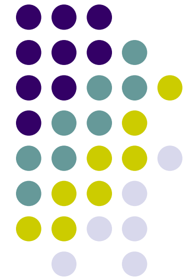
- Each process sends content of send buffer to the root process
- Root receives and stores in rank order
- Remarks:
  - Receive buffer argument ignored for all non-root processes (also recvtype, etc.)
  - `recvcount` on root indicates number of items received from each process, not total. This is a very common error
- Exercise: Sketch an implementation of MPI_Gather using only send and receive operations.

[A. Siegel]→



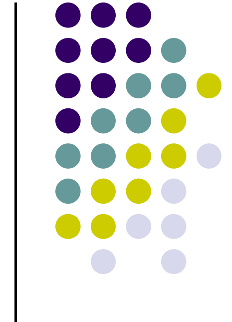
```c
#include "mpi.h"
#include <stdlib.h>
int main(int argc, char **argv){
    int myRank, nprocs, nlcl=2, n, i;
    float *data, *data_loc;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    /* local array size on each proc = nlcl */
    data_loc = (float *) malloc(nlcl*sizeof(float));

    for (i = 0; i < nlcl; ++i) data_loc[i] = myRank;

    if (myRank == 0)  data = (float *) malloc(nprocs*sizeof(float)*nlcl);

    MPI_Gather(data_loc, nlcl, MPI_FLOAT, data, nlcl, MPI_FLOAT, 0, MPI_COMM_WORLD);

    if (myRank == 0){
        for (i = 0; i < nlcl*nprocs; ++i){
            printf("%f\n", data[i]);
        }
    }

    MPI_Finalize();
    return 0;
}
```

[A. Siegel]→



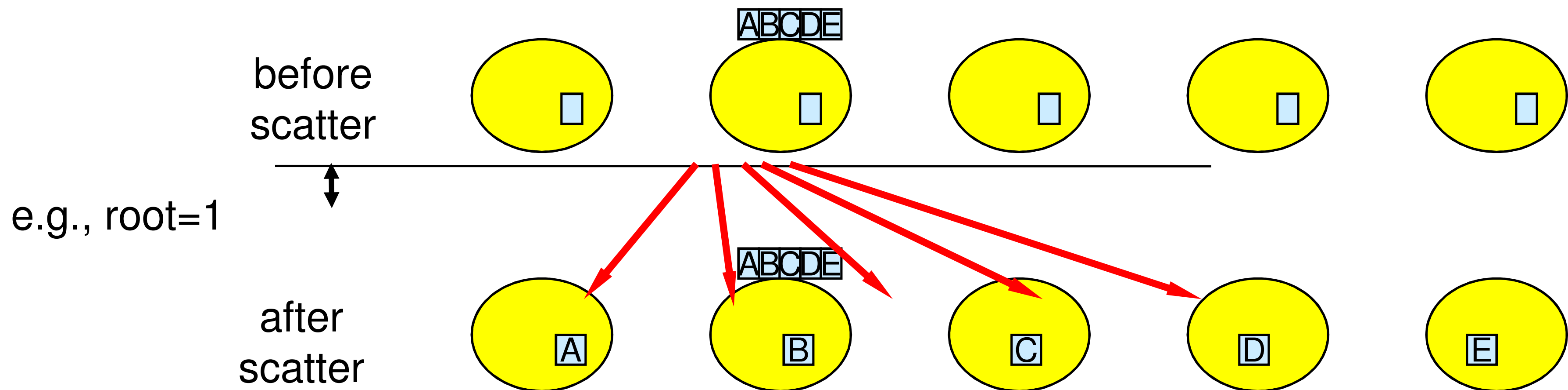
```
[negrut@euler20 CodeBits]$ mpicxx testMPI.cpp
[negrut@euler20 CodeBits]$ mpiexec -np 6 a.out
0.000000
0.000000
1.000000
1.000000
2.000000
2.000000
3.000000
3.000000
4.000000
4.000000
5.000000
5.000000
[negrut@euler20 CodeBits]$
```

Scatter



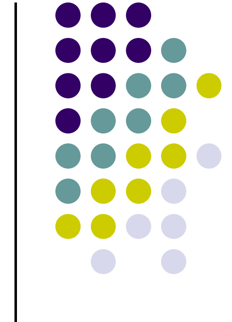
- Function prototype

```
int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
                 int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```



[ICHEC]→

MPI_Scatter



data (buffer) →

processes ↓

A_0	A_1	A_2	A_3	A_4	A_5

Scatter →

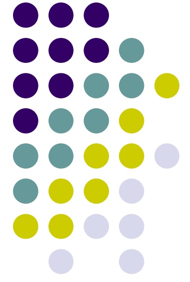
data (buffer) →

processes ↓

A_0	A_1	A_2	A_3	A_4	A_5
A_1					
A_2					
A_3					
A_4					
A_5					

[A. Siegel]→

MPI_Scatter

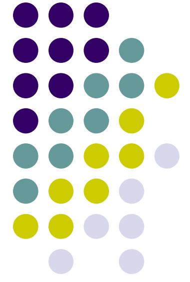


```
int MPI_Scatter (void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf,
                 int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm);
```

- IN `sendbuf` (starting address of send buffer)
- IN `sendcount` (number of elements **sent to each process**)
- IN `sendtype` (type)
- OUT `recvbuf` (address of receive bufer)
- IN `recvcount` (n-elements **in receive buffer**)
- IN `recvtype` (data type of receive elements)
- IN `root` (rank of sending process)
- IN `comm` (communicator)

[A. Siegel]→

MPI_Scatter



- Inverse of MPI_Gather

- Data elements on root listed in rank order – each processor gets corresponding data chunk after call to scatter

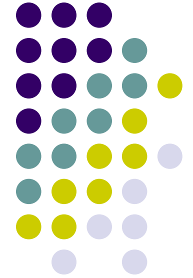
- Remarks:
  - All arguments are significant on **root**, while on other processes only **recvbuf**, **recvcount**, **recvtype**, **root**, and **comm** are significant

[A. Siegel]→

```c
#include "mpi.h"
#include <stdlib.h>

int main(int argc, char **argv){
    int myRank, nprocs, n_lcl=2;
    float *data, *data_l;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

    /* local array size on each proc = n_lcl */
    data_l = (float *) malloc(n_lcl*sizeof(float));

    if( myRank==0 ) {
        data = (float *) malloc(nprocs*sizeof(float)*n_lcl);
        for( int i = 0; i < nprocs*n_lcl; ++i) data[i] = i;
    }

    MPI_Scatter(data, n_lcl, MPI_FLOAT, data_l, n_lcl, MPI_FLOAT, 0, MPI_COMM_WORLD);

    for( int n=0; n < nprocs; ++n ){
        if( myRank==n ){
            for (int j = 0; j < n_lcl; ++j)  printf("%f\n", data_l[j]);
        }
        MPI_Barrier(MPI_COMM_WORLD);
    }

    MPI_Finalize();
    return 0;
}
```

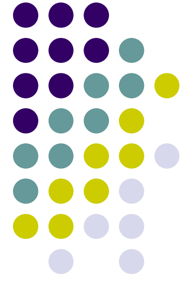
This is interesting. Think what's happening here…

[A. Siegel]→



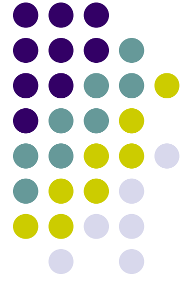
```
[negrut@euler20 CodeBits]$ mpicxx testMPI.cpp
[negrut@euler20 CodeBits]$ mpiexec -np 6 a.out
0.000000
1.000000
2.000000
3.000000
4.000000
5.000000
6.000000
7.000000
8.000000
9.000000
10.000000
11.000000
[negrut@euler20 CodeBits]$
```

Putting Things in Perspective…



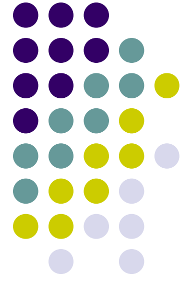
- Gather: you automatically create a serial array from a distributed one
- Scatter: you automatically create a distributed array from a serial one

Global Reduction Operations



- To perform a global reduce operation across all members of a group.
- d_0 **o** d_1 **o** d_2 **o** d_3 **o** … **o** d_{s-2} **o** d_{s-1}
  - d_i = data in process rank i
    - single variable, or
    - vector
  - **o** = associative operation
  - Example:
    - global sum or product
    - global maximum or minimum
    - global user-defined operation
- Floating point rounding may depend on usage of associative law:
  - $[(d_0 \mathbf{o} d_1) \mathbf{o} (d_2 \mathbf{o} d_3)] \mathbf{o} [\ldots \mathbf{o} (d_{s-2} \mathbf{o} d_{s-1})]$
  - $((((((d_0 \mathbf{o} d_1) \mathbf{o} d_2) \mathbf{o} d_3) \mathbf{o} \ldots) \mathbf{o} d_{s-2}) \mathbf{o} d_{s-1})$

Example of Global Reduction

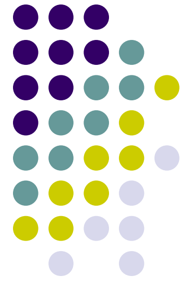


- Global integer sum
- Sum of all inbuf values should be returned in resultbuf.
- Assume root=0;

```
MPI_Reduce(&inbuf, &resultbuf, 1, MPI_INT, MPI_SUM,  root, MPI_COMM_WORLD);
```

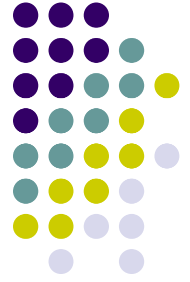
- The result is only placed in resultbuf at the root process.

Predefined Reduction Operation Handles



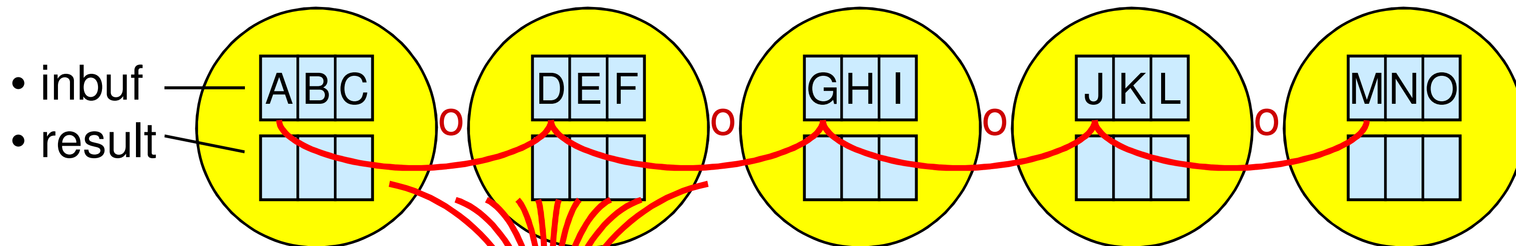
Predefined operation handle	Function
MPI_MAX	Maximum
MPI_MIN	Minimum
MPI_SUM	Sum
MPI_PROD	Product
MPI_LAND	Logical AND
MPI_BAND	Bitwise AND
MPI_LOR	Logical OR
MPI_BOR	Bitwise OR
MPI_LXOR	Logical exclusive OR
MPI_BXOR	Bitwise exclusive OR
MPI_MAXLOC	Maximum and location of the maximum
MPI_MINLOC	Minimum and location of the minimum

MPI_Reduce

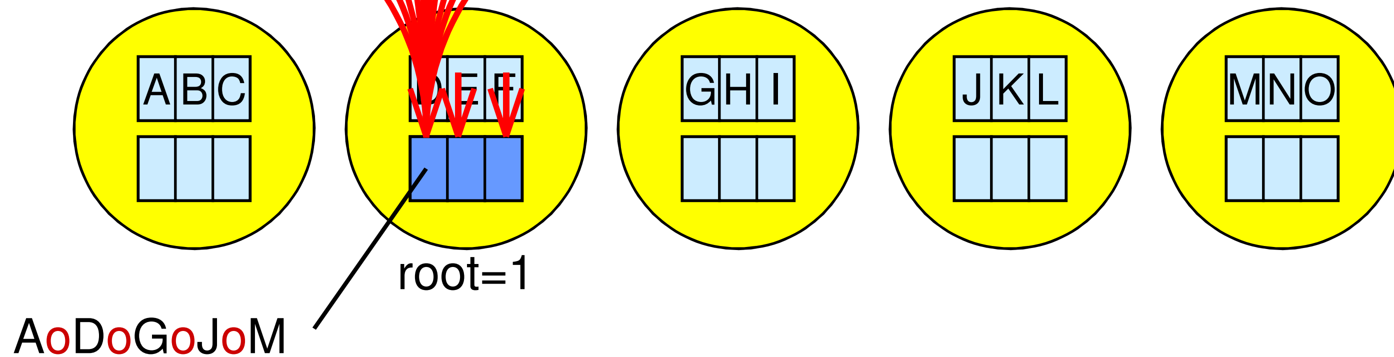


before MPI_REDUCE

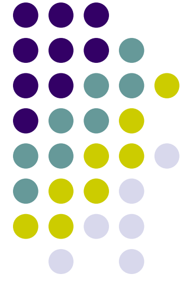
- inbuf
- result

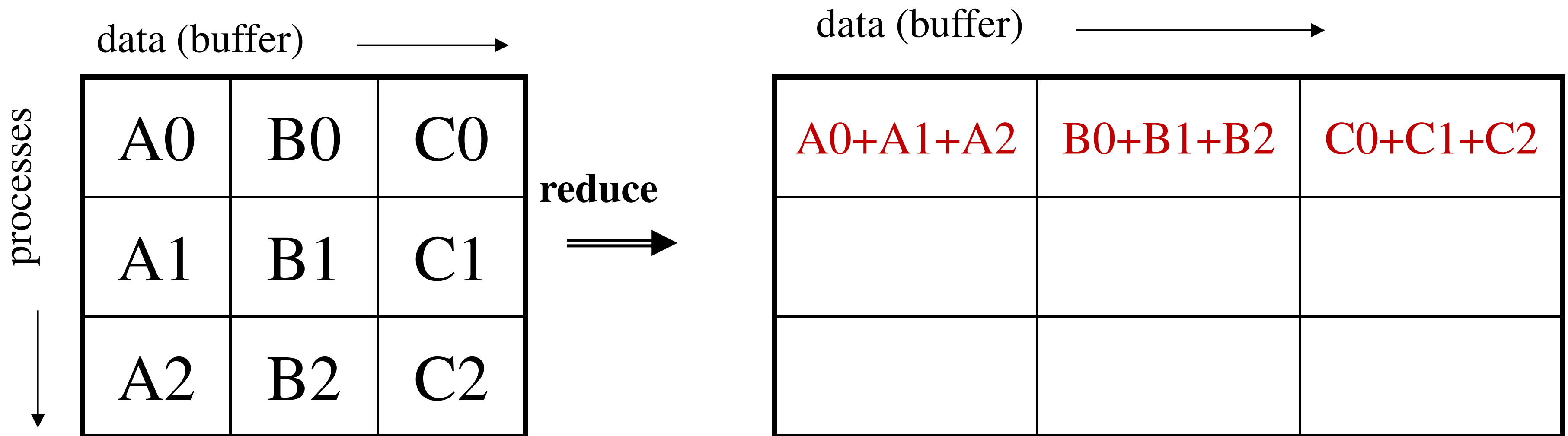


after



Reduce Operation

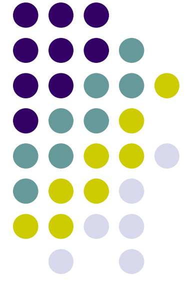




Assumption: Rank 0 is the root

[A. Siegel]→

MPI_Reduce

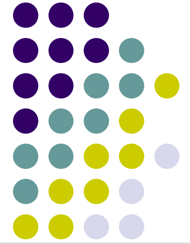


```
int MPI_Reduce (void *sendbuf, void *recvbuf, int count,
                MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm);
```

- IN `sendbuf` (address of send buffer)
- OUT `recvbuf` (address of receive buffer)
- IN `count` (number of elements in send buffer)
- IN `datatype` (data type of elements in send buffer)
- IN `op` (reduce operation)
- IN `root` (rank of root process)
- IN `comm` (communicator)

[A. Siegel]→

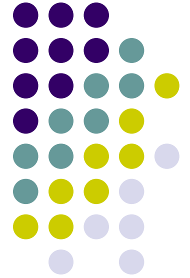
Example 1: MPI_Reduce



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

int main(int argc, char **argv){
    int mype, nprocs, gsum, gmax, gmin, data_l;

    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &mype);

    data_l = mype;

    MPI_Reduce(&data_l, &gsum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    MPI_Reduce(&data_l, &gmax, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);
    MPI_Reduce(&data_l, &gmin, 1, MPI_INT, MPI_MIN, 0, MPI_COMM_WORLD);

    if (mype == 0) printf("gsum: %d, gmax: %d  gmin:%d\n", gsum,gmax,gmin);
    MPI_Finalize();
}
```

[A. Siegel]→

Example 1: MPI_Reduce
[Output]



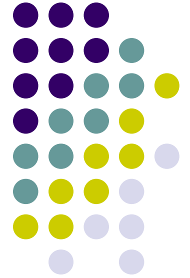
```
[negrut@euler04 CodeBits]$ mpicxx testMPI.cpp
[negrut@euler04 CodeBits]$ mpiexec -np 9 a.out
gsum: 36, gmax: 8  gmin:0
```

Example 2: MPI_Reduce

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

int main( int argc, char *argv[] ){
    int *sendbuf, *recvbuf, i, rank, size, root;

    MPI_Init( &argc, &argv );
    MPI_Comm_rank( MPI_COMM_WORLD, &rank );
    MPI_Comm_size( MPI_COMM_WORLD, &size );

    int count = 7;
    root = 3;
    sendbuf = (int *)malloc( count * sizeof(int) );
    recvbuf = (int *)malloc( count * sizeof(int) );

    for (i=0; i<count; i++) sendbuf[i] = i;
    MPI_Reduce( sendbuf, recvbuf, count, MPI_INT, MPI_SUM, root, MPI_COMM_WORLD );
    if (rank == root) {
        int errs = 0;
        for (i=0; i<count; i++) {
            if (recvbuf[i] != i * size)  errs++;
         }
        printf("Rank %d reporting.  Number of errors: %d.\n", root, errs);
    }
    free( sendbuf );
    free( recvbuf );
    MPI_Finalize();
    return errs;
}
```

[A. Siegel]→

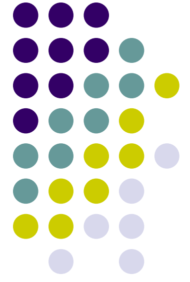
Example 2: MPI_Reduce
[Output]



```
[negrut@euler04 CodeBits]$ mpicxx testMPI.cpp
[negrut@euler04 CodeBits]$ mpiexec -np 9 a.out
Rank 3 reporting.  Number of errors: 0
```

```
[negrut@euler04 CodeBits]$ mpiexec -np 100000 a.out
[euler04:05974] [[41246,0],0] ORTE_ERROR_LOG: The system limit on number of
children a process can have was reached in file base/odls_base_default_fns.c at
line 678
--------------------------------------------------------------------------
mpiexec was unable to launch the specified application as it encountered an
error:

Error: system limit exceeded on number of processes that can be started
Node: euler04

when attempting to start process rank 0.

This can be resolved by either asking the system administrator for that node to
increase the system limit, or by rearranging your processes to place fewer of
them on that node.
--------------------------------------------------------------------------
```

Variants of Reduction Operations



- MPI_ALLREDUCE
  - No root
  - Returns the result in all processes
- MPI_REDUCE_SCATTER
  - Result vector of the reduction operation is scattered to the processes into the real result buffers
- MPI_SCAN
  - Prefix reduction
  - Result at process with rank i :=
    reduction of inbuf-values from rank 0 to rank i

[A. Siegel]→