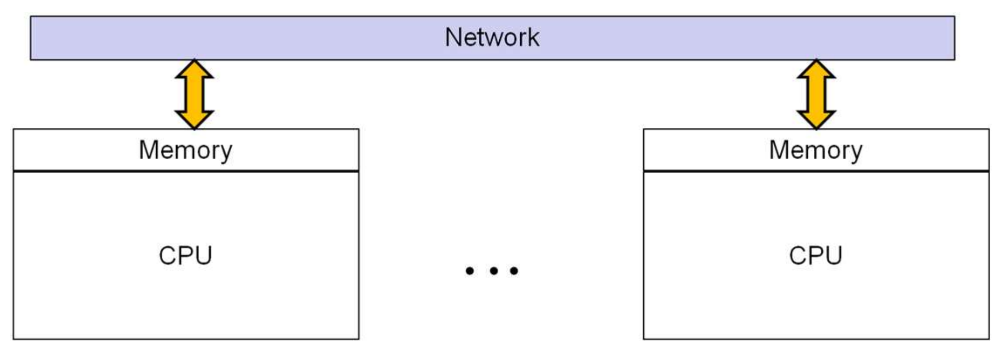
The Message Passing Interface (MPI): Parallelism on Distributed CPUs

http://mpi-forum.org
https://www.open-mpi.org/

Oregon State University

Mike Bailey

mjb@cs.oregonstate.edu



Why Two URLs?

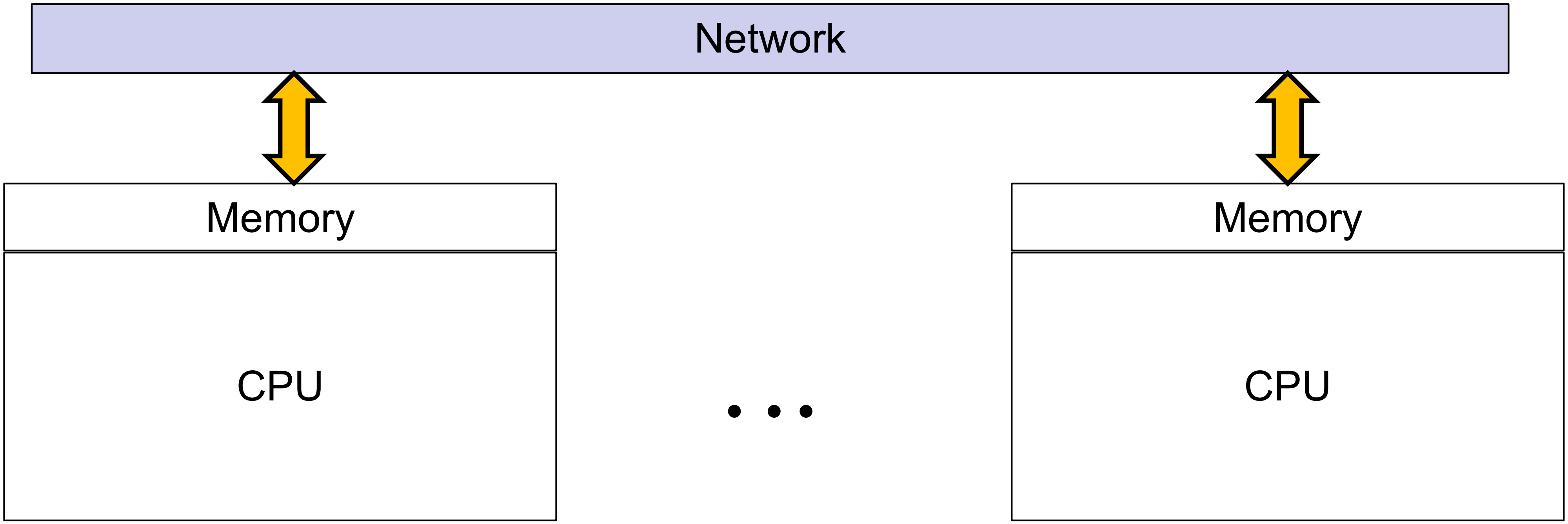
http://mpi-forum.org
This is the definitive reference for the MPI standard. Go here if you want to read the official specification, which, BTW, continues to evolve.

https://www.open-mpi.org/
This consortium formed later. This is the open source version of MPI. If you want to start using MPI, I recommend you look here.
This is the MPI that the COE systems use

https://www.open-mpi.org/doc/v4.0/
This URL is also really good – it is a link to all of the MPI man pages

The Open MPI Consortium

MPI: The Basic Idea



Programs on different CPUs coordinate computations by passing messages between each other

Note: Each CPU in the MPI "cluster" must be prepared ahead of time by having the MPI server code installed on it. It must then have that server code running and listening on its socket connection.

Each MPI CPU must also have an integer ID assigned to it (called its ***rank***).

This paradigm is how modern supercomputers work!

The Texas Advanced Computing Center’s *Frontera* supercomputer

How to SSH to the COE MPI Cluster

ssh over to an MPI submission machine -- **submit-a** and **submit-b** will also work

flip3 151% ssh **submit-c**.hpc.engr.oregonstate.edu

submit-c 142% module load slurm

Type this right away to set your path correctly

BTW, you can find out more about the COE cluster here:
https://it.engineering.oregonstate.edu/hpc

"The College of Engineering HPC cluster is a heterogeneous mix of about 130 servers providing nearly 4000 CPU cores, over 200 GPUs, and over 43 TB total RAM. The systems are connected via gigabit ethernet and Infiniband. Most of the latest servers utilize Mellanox EDR or HDR InfiniBand network connection. The cluster also has access to 150TB global scratch from the College of Engineering's Dell/EMC Isilon enterprise storage. The CoE HPC Cluster is rated at over 1700 peak TFLOPS (double-precision)."

Compiling and Running

```
mpicc   -o   program   program.c . . .
```

C

or

```
mpic++   -o   program   program.cpp . . .
```

C++

```
mpiexec -mca  btl  self,tcp   -np   4   program
```

All distributed processors execute the *same* program at the same time

# of processors to use

Warning – use *mpic++* and *mpiexec* !

Don't use g++.
Don't run by just typing the name of the executable!

Running with a *bash* Batch Script

submit.bash:

```
#!/bin/bash
#SBATCH  -J  AutoCorr
#SBATCH  -A  cs475-575
#SBATCH  -p  classmpitest
#SBATCH  -N 16      # number of nodes
#SBATCH  -n 16      # number of tasks
#SBATCH  -o  mpiproject.out
#SBATCH  -e  mpiproject.err
#SBATCH  --mail-type=END,FAIL
#SBATCH  --mail-user=joeparallel@oregonstate.edu
module  load  openmpi
mpic++ mpiproject.cpp -o mpiproject -lm
mpiexec  -mca  btl  self,tcp  -np  4  ./mpiproject
```

Your Job Name

This is the partition name that we use for our class when debugging and testing your program. Use **classmpifinal** for taking your final performance numbers.

These 3 lines are bash code

submit-c 143% sbatch submit.bash
Submitted batch job 258759

What is the Difference Between the Partitions *classmpitest* and *classmpifinal*?

classmpitest lets your program get into the system sooner, but it might be running alongside other jobs, so its performance might suffer. But, you don't care because you are just compiling and debugging, not taking performance numbers for your report.

classmpifinal makes your program wait in line until it can get dedicated resources so that you get performance results that are much more representative of what the machines can do, and thus are worthy to be listed in your report.

Auto-Notifications via Email

#SBATCH --mail-user=**joeparallel@oregonstate.edu**

You don't have to ask the system to email information to you, but if you do, *please be sure you get your own email address right*!

Our IT people are getting *really* tired of fielding the bounced emails when people misspell their own email address.

Use slurm's *scancel* if your Job Needs to Be Killed

submit-c 143% sbatch submit.bash
Submitted batch job 258759

submit-c 144% scancel 258759

Setting Up and Finishing MPI

```
#include <mpi.h>

int
main( int argc, char *argv[ ] )
{
• • •
        MPI_Init( &argc, &argv );


• • •


        MPI_Finalize( );
        return 0;
}
```

You don’t need to process command line arguments if you don’t want to.
You can just call MPI_Init( ) as:

MPI_Init(NULL, NULL);

MPI Follows a Single-Program-Multiple-Data (SPMD) Model

A ***communicator*** is a collection of CPUs that are capable of sending messages to each other

Oh, look, a **communicator** of deer!

Oh, look, a **communicator** of turkeys!

This requires MPI server code getting installed on all those CPUs. That code then needs to be running and listening on a socket connection. Only an administrator can do this.

Getting information about our place in the **communicator**:

```
int  numCPUs;        // total # of cpus involved
int  me;             // which one I am

MPI_Comm_size( MPI_COMM_WORLD, &numCPUs );

MPI_Comm_rank( MPI_COMM_WORLD, &me );
```

Size, i.e., how many altogether?

Rank, i.e., which one am I?

It is then each CPU's job to figure out what piece of the overall problem it is responsible for and then go do it.

A First Test of MPI

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

#define THEBOSS 0

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

	int  numCPUs;		// total # of cpus involved
	int  me;		// which one I am

	MPI_Comm_size( MPI_COMM_WORLD, &numCPUs );
	MPI_Comm_rank( MPI_COMM_WORLD, &me );

	if( me == THEBOSS )
		fprintf( stderr, "Rank %d says that we have a Communicator of size %d\n", THEBOSS, numCPUs );
	else
		fprintf( stderr, "Welcome from Rank %d\n", me );

	MPI_Finalize( );
	return 0;
}
```

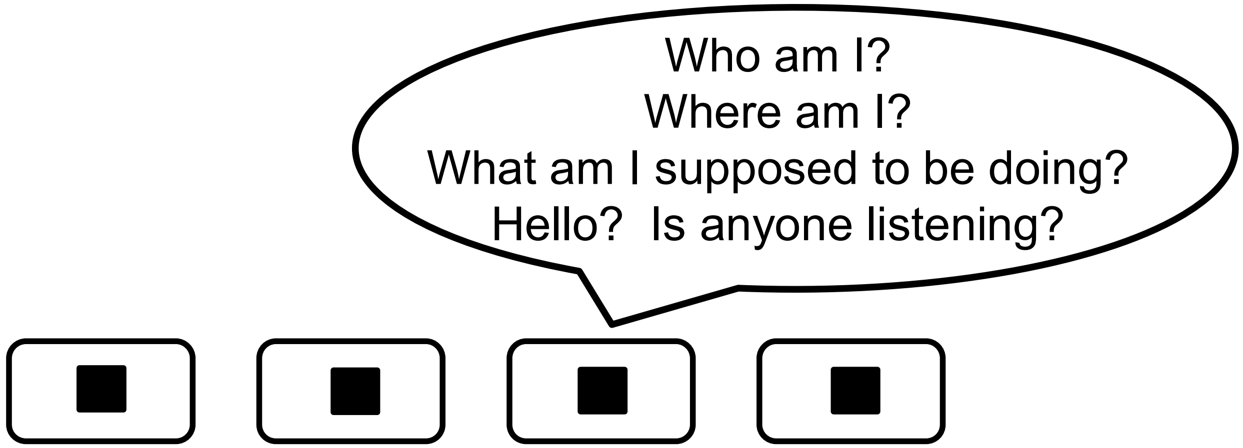
```
submit-c 165% mpiexec -np 16 ./first
Welcome from Rank 13
Welcome from Rank 15
Welcome from Rank 3
Welcome from Rank 7
Welcome from Rank 5
Welcome from Rank 8
Welcome from Rank 9
Welcome from Rank 11
Rank 0 says that we have a Communicator of size 16
Welcome from Rank 1
Welcome from Rank 12
Welcome from Rank 14
Welcome from Rank 6
Welcome from Rank 2
Welcome from Rank 10
Welcome from Rank 4
```

```
submit-c 166% mpiexec -np 16 ./first
Welcome from Rank 1
Welcome from Rank 5
Welcome from Rank 7
Welcome from Rank 9
Welcome from Rank 11
Welcome from Rank 13
Welcome from Rank 15
Rank 0 says that we have a Communicator of size 16
Welcome from Rank 2
Welcome from Rank 3
Welcome from Rank 4
Welcome from Rank 6
Welcome from Rank 8
Welcome from Rank 12
Welcome from Rank 14
Welcome from Rank 10
```

```
submit-c 167% mpiexec -np 16 ./first
Welcome from Rank 9
Welcome from Rank 11
Welcome from Rank 13
Welcome from Rank 7
Welcome from Rank 1
Welcome from Rank 3
Welcome from Rank 10
Welcome from Rank 15
Welcome from Rank 4
Welcome from Rank 5
Rank 0 says that we have a Communicator of size 16
Welcome from Rank 2
Welcome from Rank 6
Welcome from Rank 8
Welcome from Rank 14
Welcome from Rank 12
```

```
submit-c 168% mpiexec -np 16 ./first
Welcome from Rank 13
Welcome from Rank 15
Welcome from Rank 7
Welcome from Rank 3
Welcome from Rank 5
Welcome from Rank 9
Welcome from Rank 11
Welcome from Rank 1
Welcome from Rank 12
Welcome from Rank 14
Welcome from Rank 4
Welcome from Rank 2
Rank 0 says that we have a Communicator of size 16
Welcome from Rank 8
Welcome from Rank 10
Welcome from Rank 6
```

So, we have a group (a "communicator") of distributed processors. How do they communicate about what work they are supposed to do?



Example: You could coordinate the units of our DGX system using MPI

Oregon State University
Computer Graphics

A Good Place to Start: MPI Broadcasting

MPI_Bcast(array, count, type, src, MPI_COMM_WORLD);

Address of the data to send *from* if you are the *src* node;

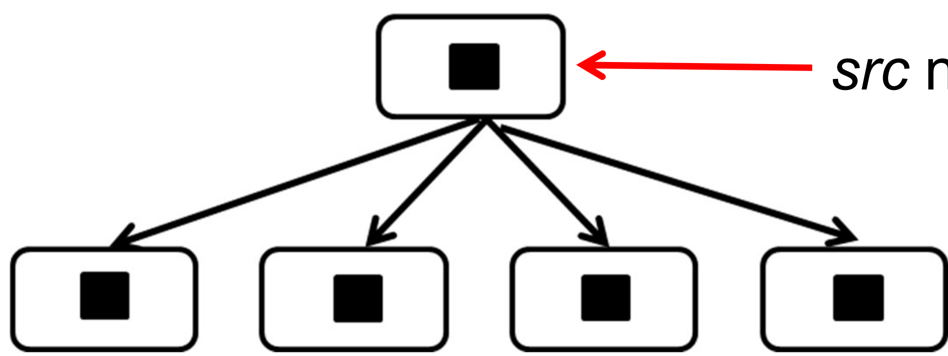
Address of the data to receive *into* if you are not

 # elements

MPI_CHAR
MPI_INT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE
• • •

rank of the CPU doing the sending

Broadcast



src node

≠ *src* nodes

Both the sender and receivers need to execute **MPI_Bcast** – there is no separate receive function

MPI Broadcast Example

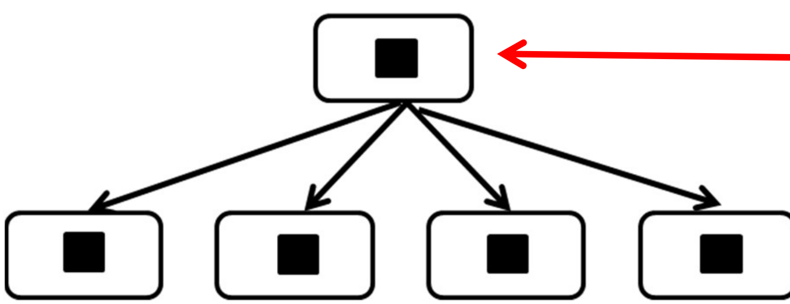
This is our heat transfer equation from before. Clearly, every CPU will need to know this value.

$$\Delta T_i = \left(\frac{k}{\rho C}\right)\left(\frac{T_{i-1} - 2T_i + T_{i+1}}{(\Delta x)^2}\right)\Delta t$$

```
int    numCPUs;
int    me;
float  k_over_rho_c;                    // the THEBOSS node will know this value, the others won't (yet)

#define THEBOSS         0

MPI_Comm_size(  MPI_COMM_WORLD,  &numCPUs );            // how many are in this communicator
MPI_Comm_rank( MPI_COMM_WORLD,  &me );                  // which one am I?
. . .
if( me == THEBOSS )
{
      << read  k_over_rho_c  from the data file >>
}

MPI_Bcast( &k_over_rho_c, 1, MPI_FLOAT, THEBOSS, MPI_COMM_WORLD );// send if I am THEBOSS, and receive if not
```

I am the THEBOSS: this identifies this call as a send

Broadcast



src node

≠ *src* nodes

How Does this Work?
Think Star Trek Wormholes!

Sending Data from One Source CPU to Just One Destination CPU

MPI_Send(array, numToSend, type, dst, tag, MPI_COMM_WORLD);

address of data to send from

 # elements
(note: this is the number of *elements*, not the number of *bytes*!)

MPI_CHAR
MPI_INT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE
• • •

rank of the CPU to send to

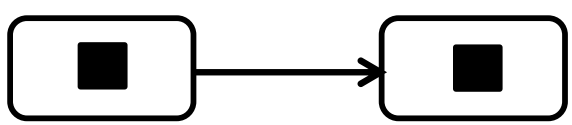
An integer or character to differentiate this transmission from any other transmission.
I like to use chars.

Rules:

• One message from a specific *src* to a specific *dst* cannot overtake a previous message from the same *src* to the same *dst*.

• MPI_Send( ) blocks until the transfer is far enough along that *array* can be destroyed or re-used.

• There are no guarantees on order from different *src*'s .

src node  *dst* node

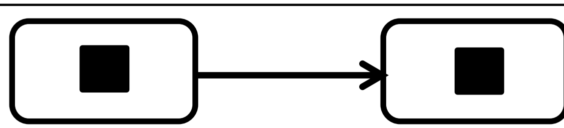
Receiving Data in a Destination CPU from a Source CPU

```
MPI_Recv( array, maxCanReceive, type, src, tag, MPI_COMM_WORLD, &status );
```

- **array**: address of data to receive into
- **maxCanReceive**: # elements we can receive, at most
- **type**: MPI_CHAR, MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, • • •
- **src**: Rank of the CPU we are expecting to get a transmission from
- **tag**: An integer or character to differentiate what transmission we are looking for with this call (be sure this matches what the sender is sending!). I like to use chars.
- **&status**: Type = MPI_Status

Rules:

- The receiver blocks waiting for data that matches what it declares to be looking for
- One message from a specific *src* to a specific *dst* cannot overtake a previous message from the same *src* to the same *dst*
- There are no guarantees on the order from different *src*'s
- The order from different *src*'s could be implied in the *tag*
- **status** is type MPI_Status – the "&status" can be replaced with MPI_STATUS_IGNORE

src node  *dst* node

Example

Remember, this *identical code* runs on all CPUs:

```
int  numCPUs;
int  me;
#define MYDATA_SIZE     128
char  mydata[ MYDATA_SIZE ];
#define THEBOSS         0

MPI_Comm_size( MPI_COMM_WORLD, &numCPUs );
MPI_Comm_rank( MPI_COMM_WORLD, &me );

if( me == THEBOSS )       // the primary
{
        for( int dst = 0; dst < numCPUs; dst++ )
        {
                if( dst != THEBOSS )
                {
                        char *InputData = "Hello, Beavers!";
                        MPI_Send( InputData, strlen(InputData)+1, MPI_CHAR, dst, 'B', MPI_COMM_WORLD );
                }
        }
}
else                      // a secondary
{
        MPI_Recv( myData, MYDATA_SIZE, MPI_CHAR, THEBOSS, 'B', MPI_COMM_WORLD, MPI_STATUS_IGNORE );
        printf( " '%s' from rank # %d\n", in, me );
}
```

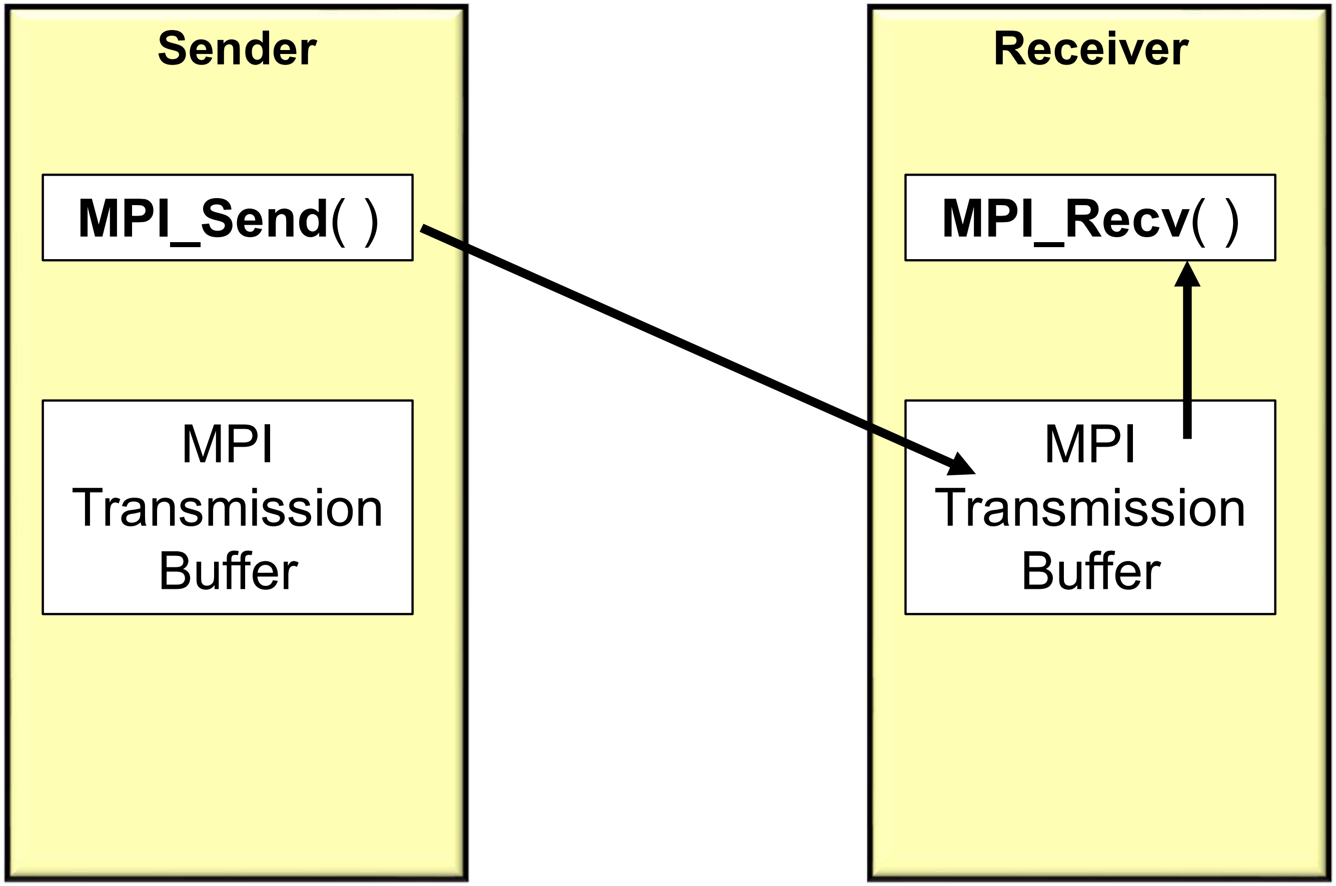
Be sure the receiving tag matches the sending tag

The tag to label this transmission with

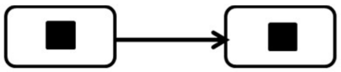
The tag to expect

You are highly discouraged from sending to yourself. Because both the send and receive are capable of blocking, the result could be deadlock.

How does MPI let the Sender perform an MPI_Send( ) even if the Receivers are not ready to MPI_Recv( )?



MPI_Send( ) blocks until the transfer is far enough along that the *array* can be destroyed or re-used.

src node  *dst* node

Another Example

You typically don't send the entire workload to each dst – you just send part of it, like this:

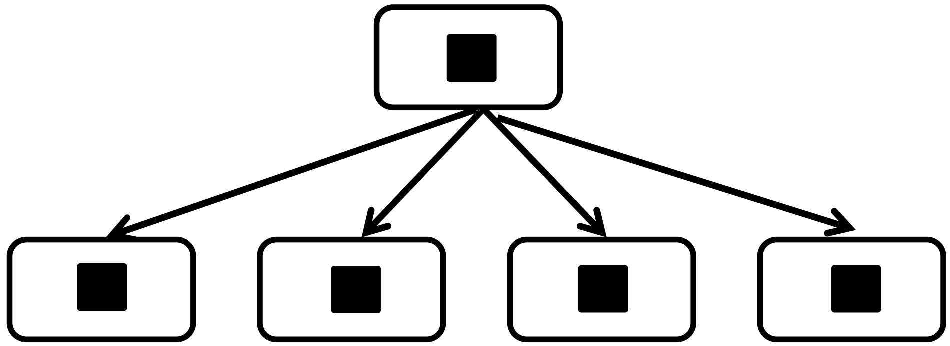
```
#define NUMELEMENTS    ?????
int   numCPUs;
int   me;
#define THEBOSS        0

MPI_Comm_size(  MPI_COMM_WORLD, &numCPUs );
MPI_Comm_rank( MPI_COMM_WORLD, &me );

int PPSize = NUMELEMENTS / numCPUs;          // per-processor data size -- assuming it comes out evenly
float *myData = new float [ PPSize ];

if( me == THEBOSS )      // the sender
{
        float *InputData = new float [ NUMELEMENTS ];
        << read the full input data into InputData from disk >>
        for( int dst = 0; dst < numCPUs; dst++ )
        {
                if( dst != THEBOSS )
                {
                        MPI_Send( &InputData[dst*PPSize], PPSize, MPI_FLOAT, dst, 0, MPI_COMM_WORLD );
                }
        }
}
else                     // a receiver
{
        MPI_Recv( myData, PPSize, MPI_FLOAT, THEBOSS, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
        // do something with this subset of the data
}
```

The address of node *dst*'s share of the data to send

Each *dst* node will store its data in this array

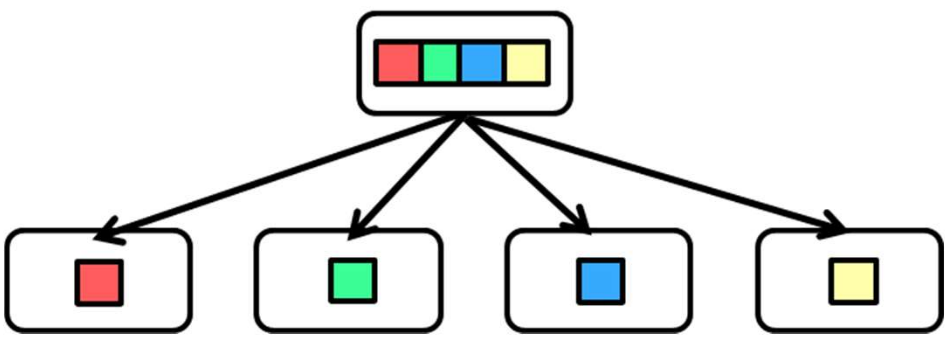
In Distributed Computing, You Often Hear About These Design Patterns

Broadcast



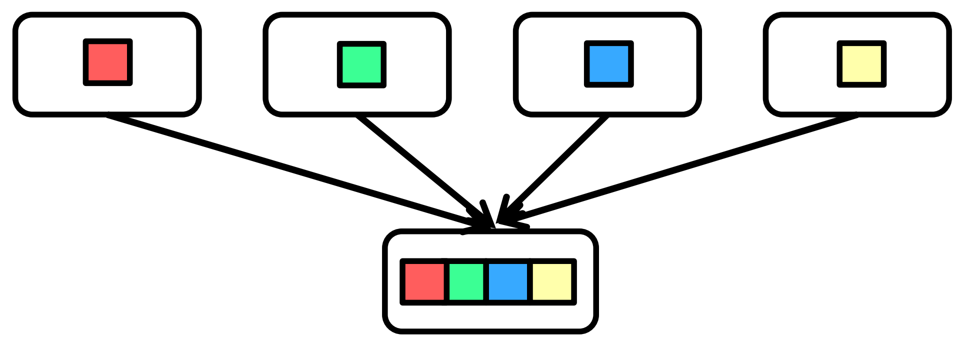
This sends the same data to each processor

Scatter



This sends pieces of the data to each processor

Gather

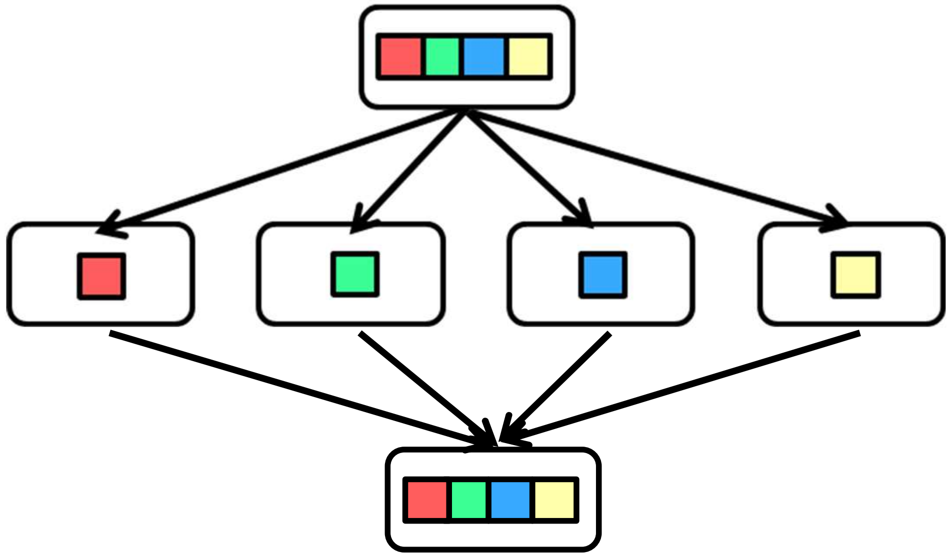


This brings back pieces of the results from each processor

Scatter and Gather Usually Go Together

This sends pieces of the data to
each processor

Scatter



Gather

This brings back pieces of the results from
each processor

Note surprisingly, this is referred to by the combined term ***Scatter/Gather***

MPI Scatter

Take a data array, break it into ~equal portions, and send it to each CPU

MPI_Scatter(snd_array, snd_count, snd_type, rcv_array, rcv_count, rcv_type, src, MPI_COMM_WORLD);

The total large array to split up

elements to send *per-processor*

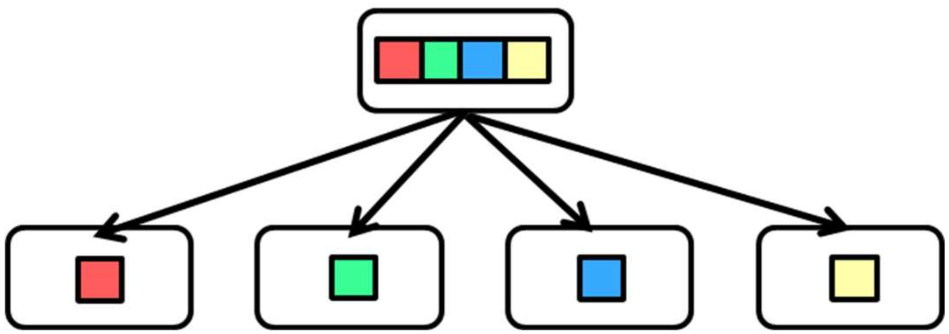
MPI_CHAR
MPI_INT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE
• • •

Local array to store this processor's piece in

elements to receive *per-processor*

MPI_CHAR
MPI_INT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE
• • •

This is who is doing the sending – everyone *else* is receiving



Scatter

Both the sender and receivers need to execute **MPI_Scatter.**
There is no separate receive function

MPI Gather

MPI_Gather(snd_array, snd_count, snd_type, rcv_array, rcv_count, rcv_type, dst, MPI_COMM_WORLD);

The total large array to put the pieces back into

elements to return *per-processor*

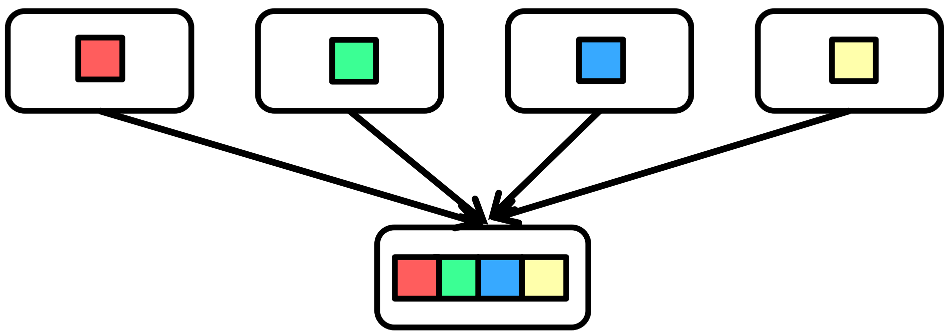
MPI_CHAR
MPI_INT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE
• • •

Local array that this processor is sending back

elements to send back *per-processor*

MPI_CHAR
MPI_INT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE
• • •

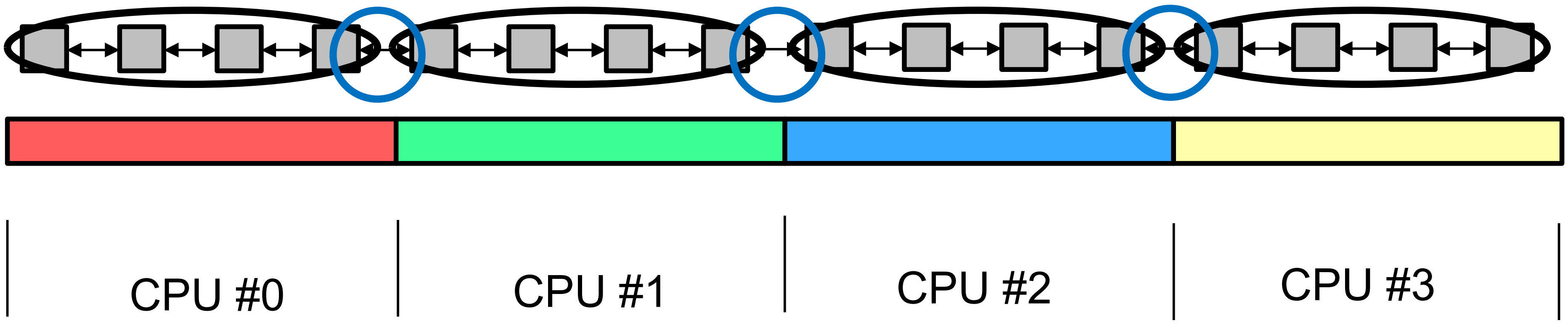
This is who is doing the receiving – everyone *else* is sending



Gather

Both the sender and receivers need to execute **MPI_Gather.**
There is no separate transmit function

Remember This? It's Baaaaaack as a complete Scatter/Gather Example



The **Compute : Communicate Ratio** still applies, except that it is even more important now because there is much more overhead in the Communicate portion.

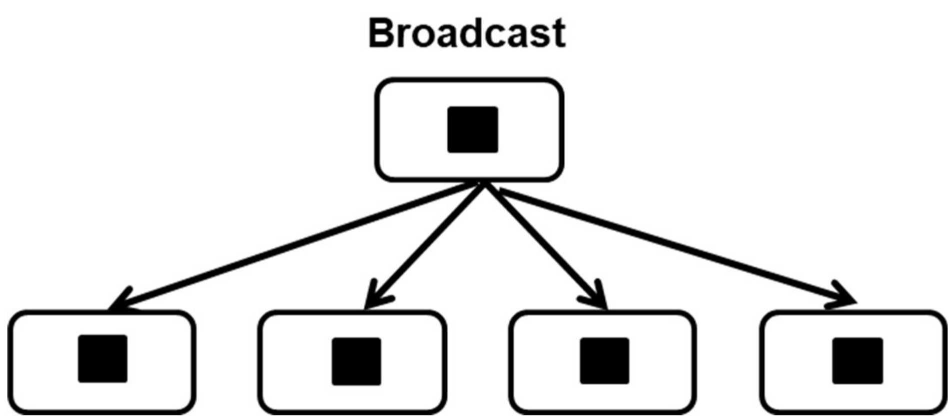
This pattern of breaking a big problem up into pieces, sending them to different CPUs, computing on the pieces, and getting the results back is *very* common. That's why MPI has its own scatter and gather functions.

heat.cpp, I

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

const float RHO = 8050.;
const float C   = 0.466;
const float K   = 20.;
float k_over_rho_c =  K / (RHO*C);              // units of m^2/sec   NOTE: this cannot be a const!
// K / (RHO*C) = 5.33x10^-6 m^2/sec

const float DX    =        1.0;
const float DT    =        1.0;

#define THEBOSS 0

#define NUMELEMENTS             (8*1024*1024)
#define  NUM_TIME_STEPS         4
#define DEBUG                   false

float *         NextTemps;      // per-processor array to hold computer next-values
int             NumCpus;        // total # of cpus involved
int             PPSize;         // per-processor local array size
float *         PPTemps;        // per-processor local array temperature data
float *         TempData;       // the overall NUMELEMENTS-big temperature data

void            DoOneTimeStep( int );
```

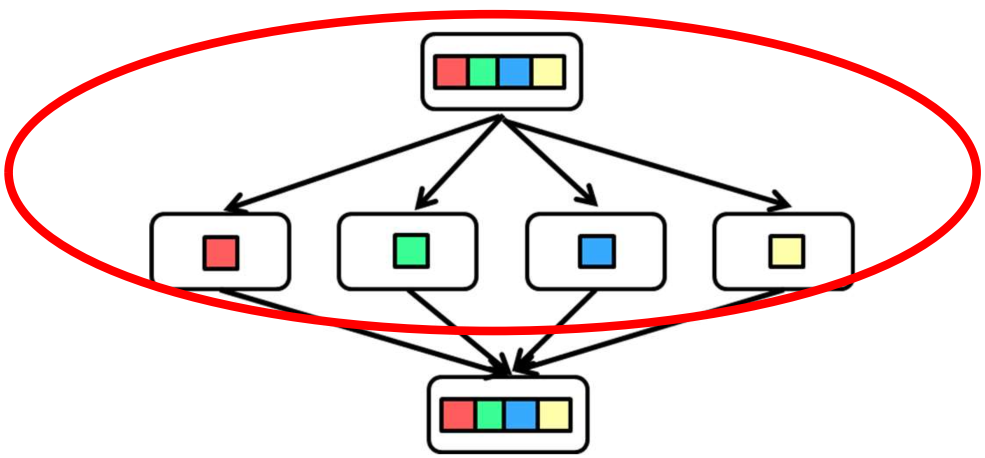
heat.cpp, II

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

        int  me;                 // which one I am

        MPI_Comm_size( MPI_COMM_WORLD, &NumCpus );
        MPI_Comm_rank( MPI_COMM_WORLD, &me );

        // decide how much data to send to each processor:
        PPSize = NUMELEMENTS / NumCpus;                  // assuming it comes out evenly
        PPTemps    = new float [PPSize];   // all processors now have this uninitialized Local array
        NextTemps  = new float [PPSize];   // all processors now have this uninitialized local array too

        // broadcast the constant:
        MPI_Bcast( (void *)&k_over_rho_c, 1, MPI_FLOAT, THEBOSS, MPI_COMM_WORLD );
```



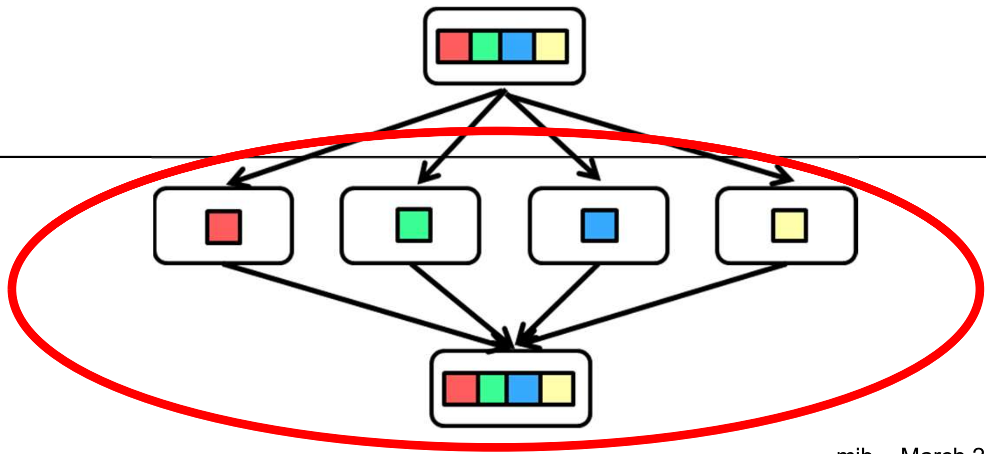
heat.cpp, III


```
if( me == THEBOSS )   // this is the data-creator
{
        TempData = new float [NUMELEMENTS];
        for( int i = 0; i < NUMELEMENTS; i++ )
                TempData[ i ] =   0.;
        TempData[NUMELEMENTS/2] = 100.;
}

MPI_Scatter( TempData, PPSize, MPI_FLOAT,  PPTemps, PPSize, MPI_FLOAT,
        THEBOSS, MPI_COMM_WORLD );
```



heat.cpp, IV

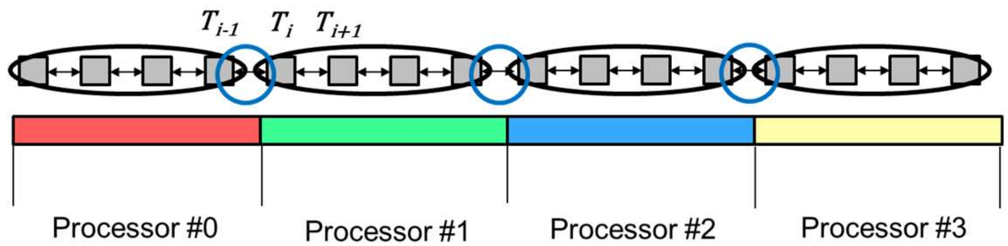
```
          // all the PPTemps arrays have now been filled
          // do the time steps:

          double time0 = MPI_Wtime( );

          for( int steps = 0; steps < NUM_TIME_STEPS; steps++ )
          {
                    // do the computation for one time step:
                    DoOneTimeStep( me );

                    // ask for all the data:
#ifdef WANT_EACH_TIME_STEPS_DATA_BACK
                    MPI_Gather( PPTemps, PPSize, MPI_FLOAT, TempData, PPSize, MPI_FLOAT,
                              THEBOSS, MPI_COMM_WORLD );
#endif
          }

#ifndef WANT_EACH_TIME_STEPS_DATA_BACK
          MPI_Gather( PPTemps, PPSize, MPI_FLOAT, TempData, PPSize, MPI_FLOAT,
                    THEBOSS, MPI_COMM_WORLD );
#endif

          double time1 = MPI_Wtime( );
```



heat.cpp, V

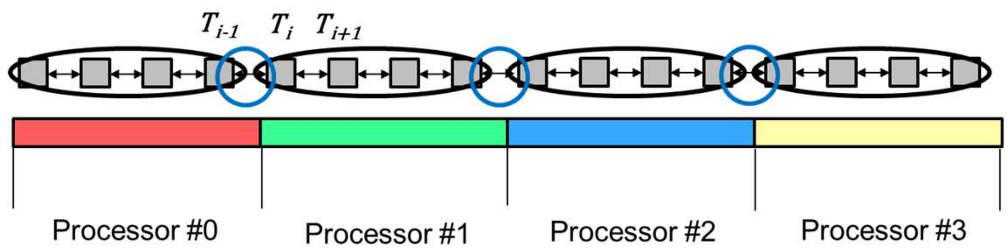
```
        if( me == THEBOSS )
        {
                double seconds = time1 - time0;
                double performance =
                        (double)NUM_TIME_STEPS * (double)NUMELEMENTS / seconds / 1000000.;
                                // mega-elements computed per second
                fprintf( stderr, "%3d, %10d, %8.2lf\n", NumCpus, NUMELEMENTS, performance );
        }

        MPI_Finalize( );
        return 0;
}
```

DoOneTimeStep, I



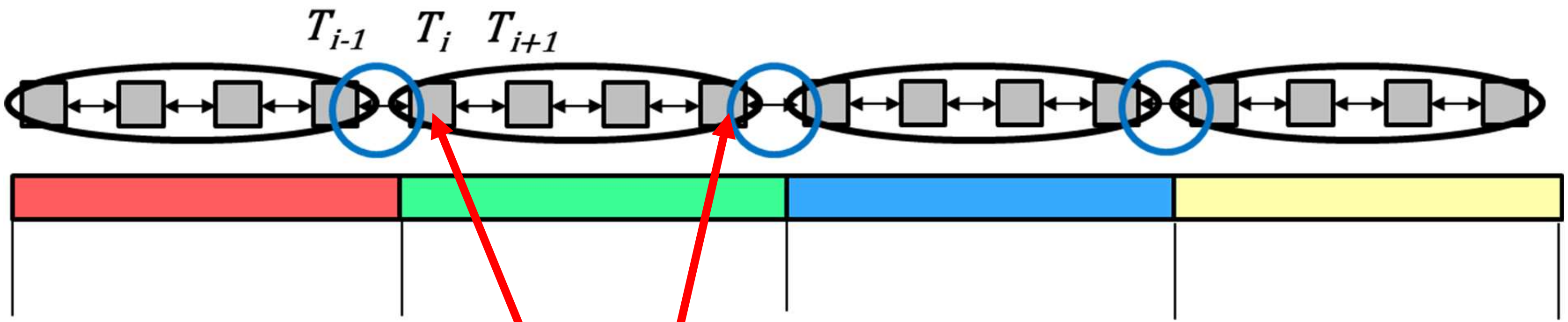

```
// read from PerProcessorData[ ],  write into NextTemps[ ]
void
DoOneTimeStep( int me )
{
        MPI_Status status;

        // send out the left and right end values:
        // (the tag is from the point of view of the sender)
        if( me != 0 )                           // i.e., if i'm not the first group on the left
        {
                // send my PPTemps[0] to me-1 using tag 'L'
                MPI_Send( &PPTemps[0], 1, MPI_FLOAT, me-1, 'L', MPI_COMM_WORLD );
                if( DEBUG )   fprintf( stderr, "%3d sent 'L' to %3d\n", me, me-1 );
        }

        if( me != NumCpus-1 )                   // i.e., not the last group on the right
        {
                // send my PPTemps[PPSize-1] to me+1 using tag 'R'
                MPI_Send( &PPTemps[PPSize-1], 1, MPI_FLOAT, me+1, 'R', MPI_COMM_WORLD );
                if( DEBUG )  fprintf( stderr, "%3d sent 'R' to %3d\n", me, me+1 );
        }
```

DoOneTimeStep, II



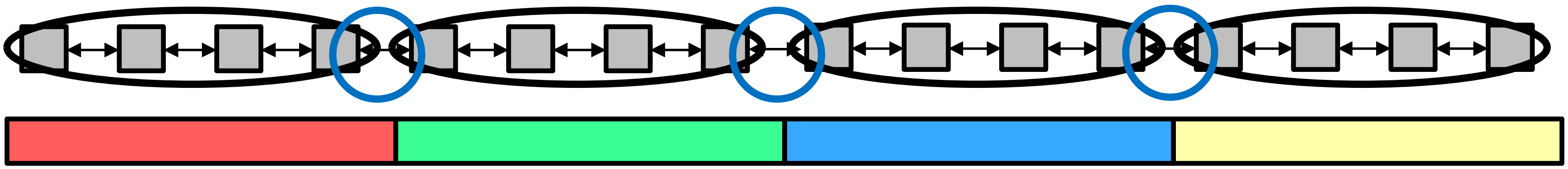

```
float left  = 0.;
float right = 0.;

if( me != 0 )                              // i.e., if i'm not the first group on the left
{
        // receive my "left" from me-1 using tag 'R'
        MPI_Recv( &left, 1, MPI_FLOAT, me-1, 'R', MPI_COMM_WORLD, &status );
        if( DEBUG )   fprintf( stderr, "%3d received 'R' from %3d\n", me, me-1 );
}

if( me != NumCpus-1 )                      // i.e., not the last group on the right
{
        // receive my "right" from me+1 using tag 'L'
        MPI_Recv( &right, 1, MPI_FLOAT, me+1, 'L', MPI_COMM_WORLD, &status );
        if( DEBUG )    fprintf( stderr, "%3d received 'L' from %3d\n", me, me+1 );
}
```

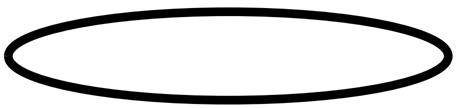
Sharing Values Across the Boundaries



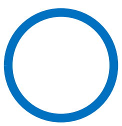
1 sent 'L' to 0
1 sent 'R' to 2
2 sent 'L' to 1
2 sent 'R' to 3
2 received 'R' from 1
0 sent 'R' to 1
0 received 'L' from 1
1 received 'R' from 0
1 received 'L' from 2
3 sent 'L' to 2
3 received 'R' from 2
2 received 'L' from 3

1D Compute-to-Communicate Ratio





Intraprocessor computing



Interprocessor communication

Compute : Communicate ratio = N : 2

where N is the number of compute cells per processor

In the above drawing, Compute : Communicate is 4 : 2

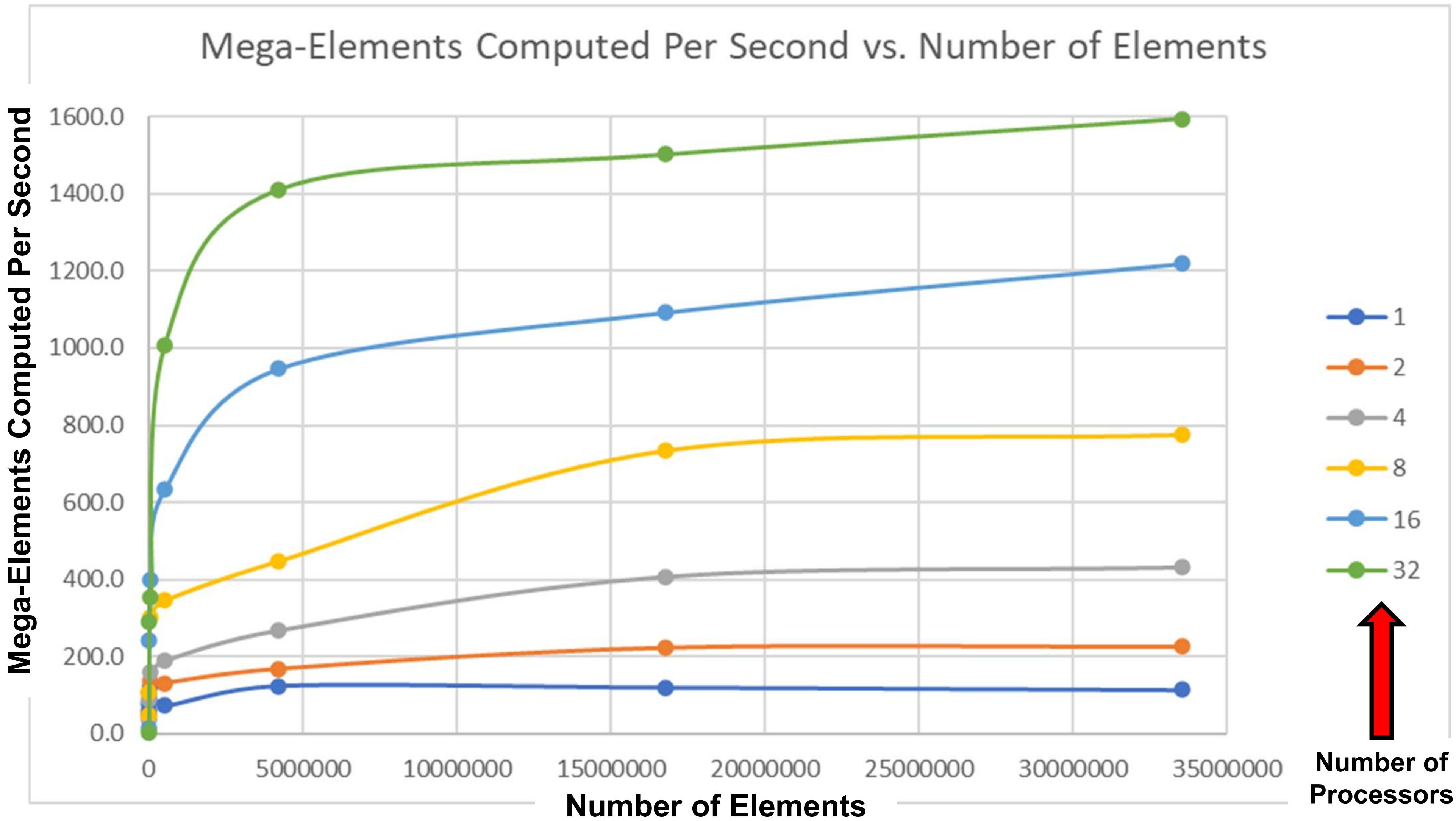
DoOneTimeStep, III

```
// first element on the left (0):
{
        float dtemp = ( k_over_rho_c *
                ( left - 2.*PPTemps[0] +  PPTemps[1] ) / ( DX*DX ) ) * DT;
        NextTemps[0] = PPTemps[0] + dtemp;
}


// all the nodes in the middle:
for( int i = 1; i < PPSize-1; i++ )
{
        float dtemp = ( k_over_rho_c *
                ( PPTemps[i-1] - 2.*PPTemps[ i ] + PPTemps[i+1] ) / ( DX*DX ) ) * DT;
        NextTemps[i] = PPTemps[i] + dtemp;
}


// last element on the right (PPSize-1):
{
        float dtemp = ( k_over_rho_c *
                ( PPTemps[PPSize-2] - 2.*PPTemps[PPSize-1] + right ) / ( DX*DX ) ) * DT;
        NextTemps[PPSize-1] = PPTemps[PPSize-1] + dtemp;
 }
```

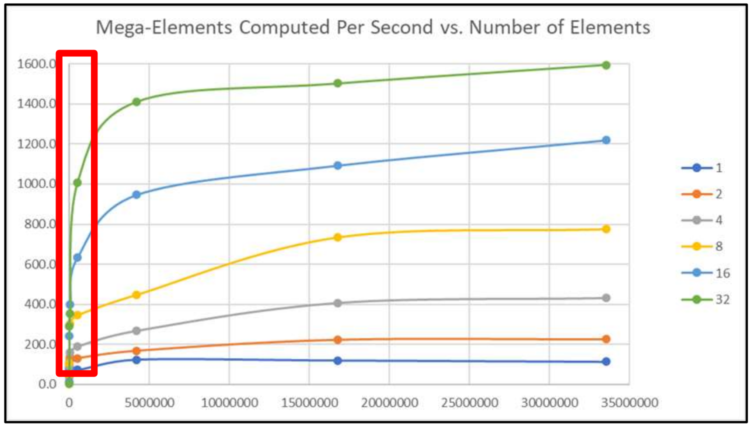
DoOneTimeStep, IV

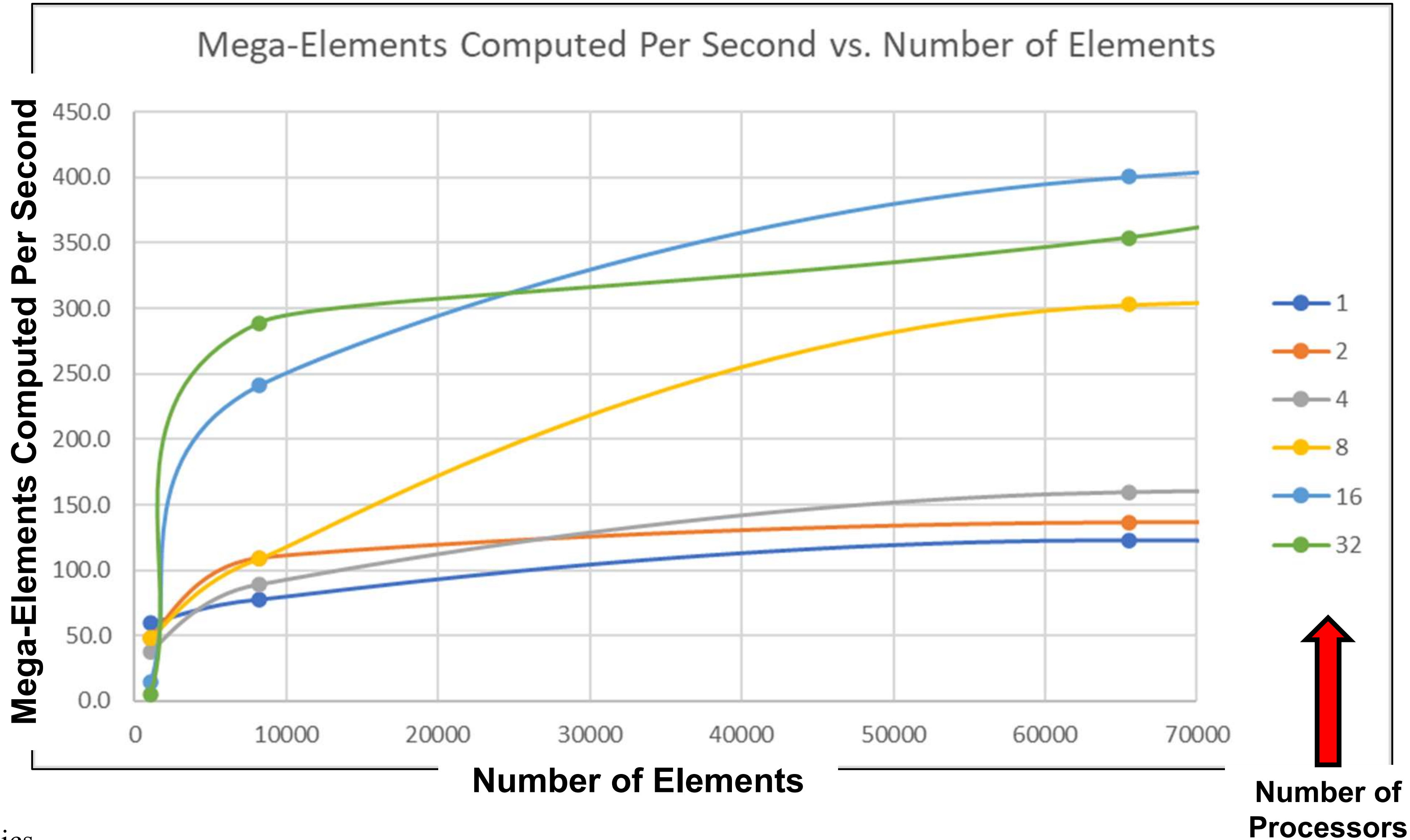

```
          // update the local dataset:

          for( int i = 0; i < PPSize; i++ )
          {
                    PPTemps[ i ] = NextTemps[ i ];
          }
}
```

MPI Performance

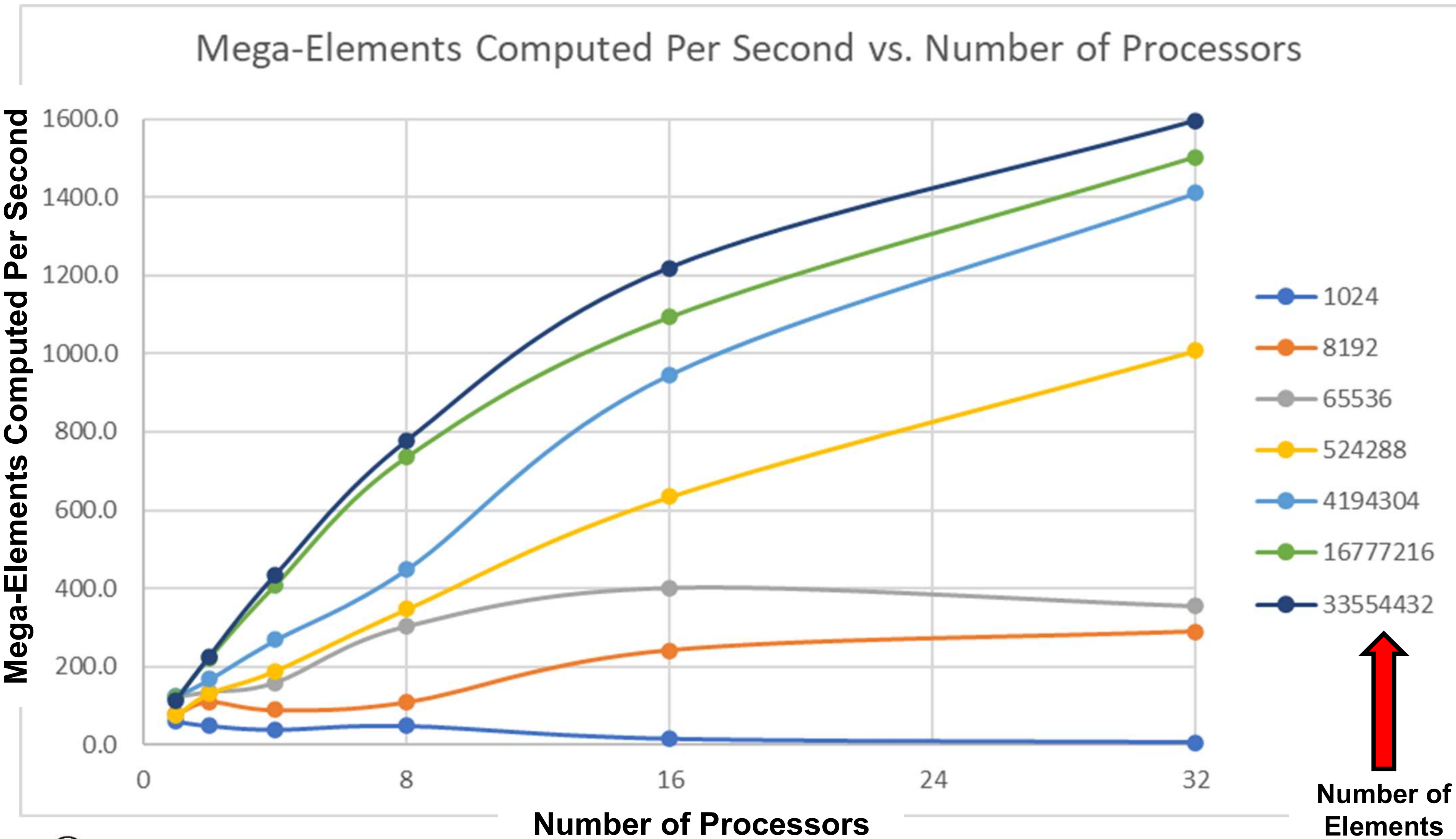


Low Dataset-Size MPI Performance



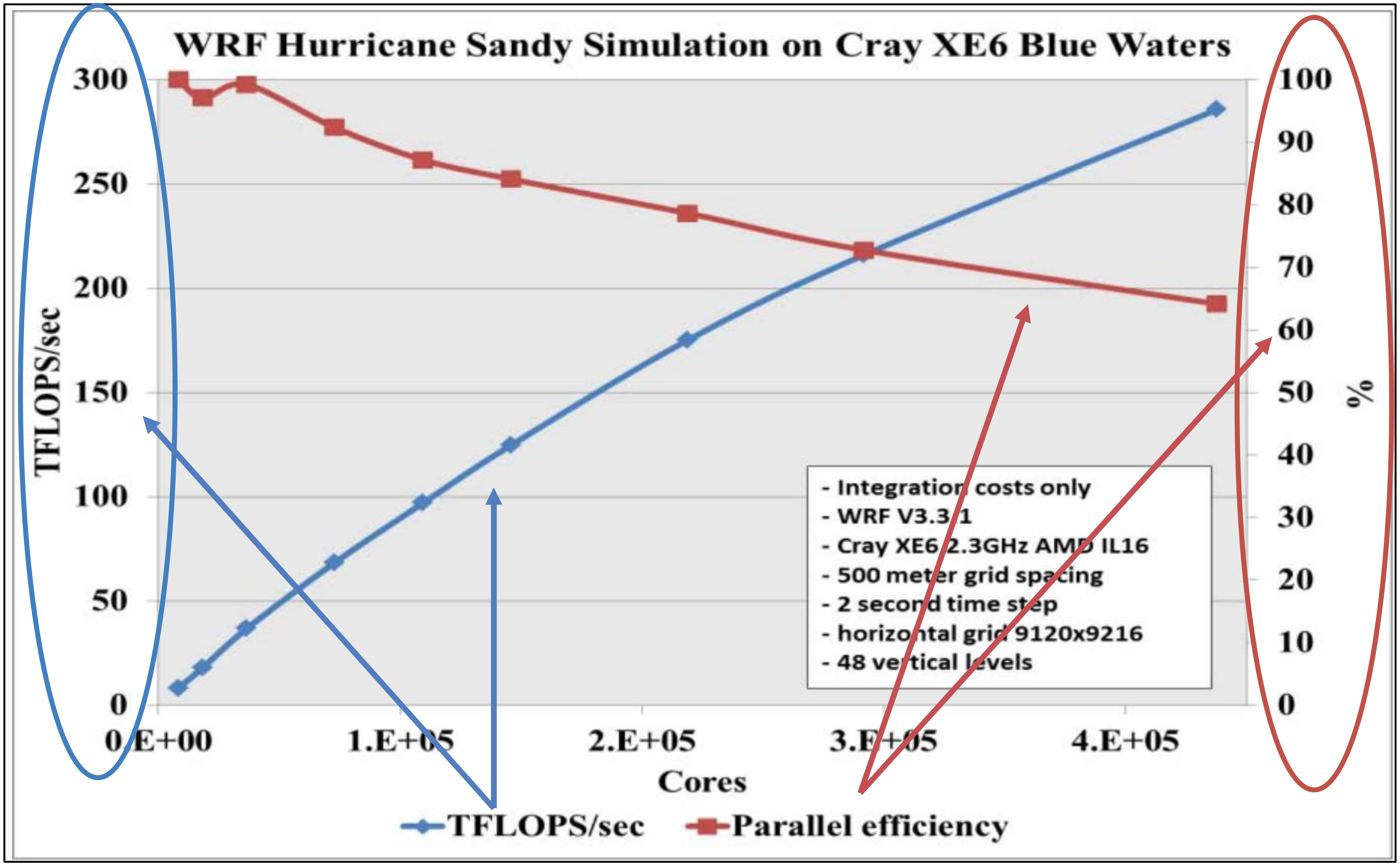


MPI Performance



Oregon State University
Computer Graphics

Using MPI and OpenMP on 13,680 nodes (437,760 cores) of the Cray XE6 at NCSA at the University of Illinois



From: Peter Johnsen, Mark Straka, Melvyn Shapiro, Alan Norton, Thomas Galarneau, *Petascale WRF Simulation of Hurricane Sandy.*

MPI Reduction

MPI_Reduce(partialResult, globalResult, count, type, operator, dst, MPI_COMM_WORLD);

Where the partial result is stored on each CPU

Place to store the full result on the dst CPU

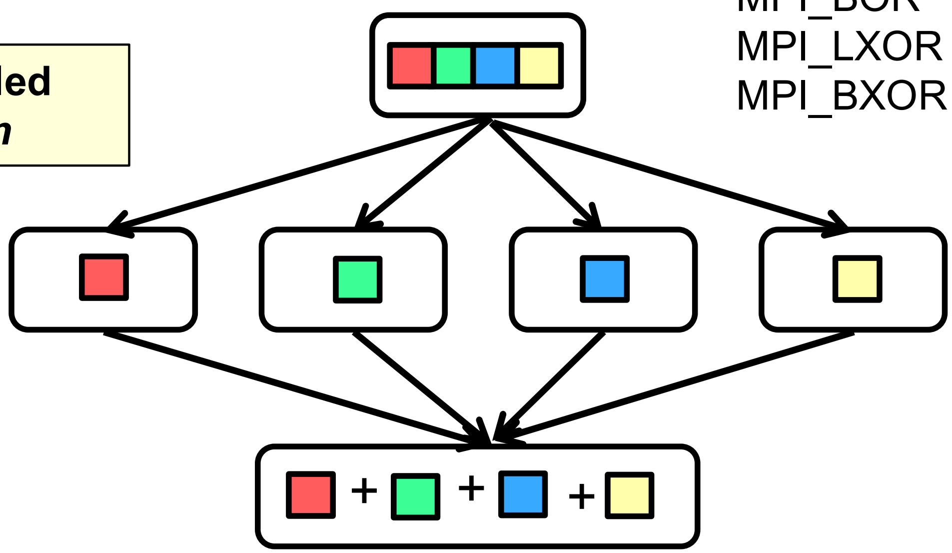
Number of elements in the partial result

MPI_CHAR
MPI_INT
MPI_LONG
MPI_FLOAT
MPI_DOUBLE
• • •

MPI_MIN
MPI_MAX
MPI_SUM
MPI_PROD
MPI_MINLOC
MPI_MAXLOC
MPI_LAND
MPI_BAND
MPI_LOR
MPI_BOR
MPI_LXOR
MPI_BXOR

Who is given the final answer

This really should be called *Scatter/Gather/Reduction*

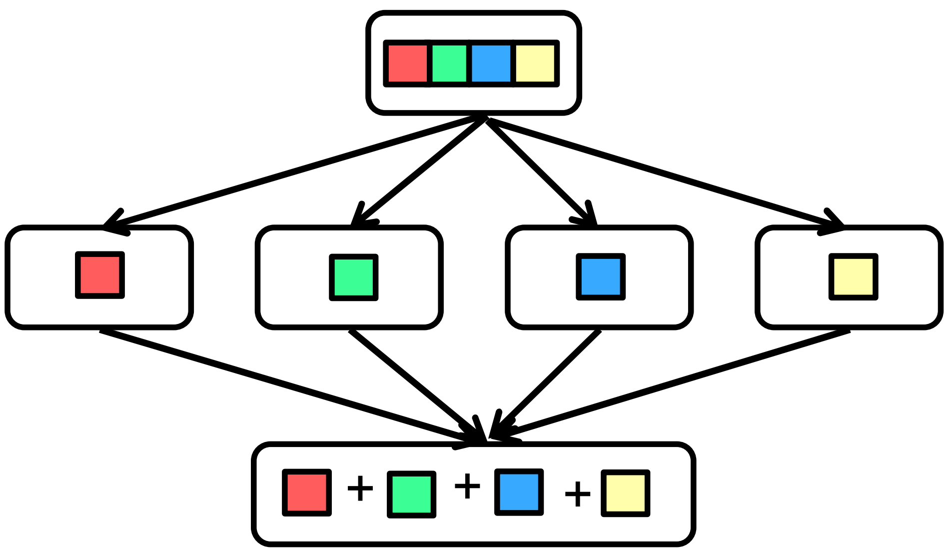


Reduction

Both the sender and receivers need to execute **MPI_Reduce**.
There is no separate receive function

MPI Reduction Example

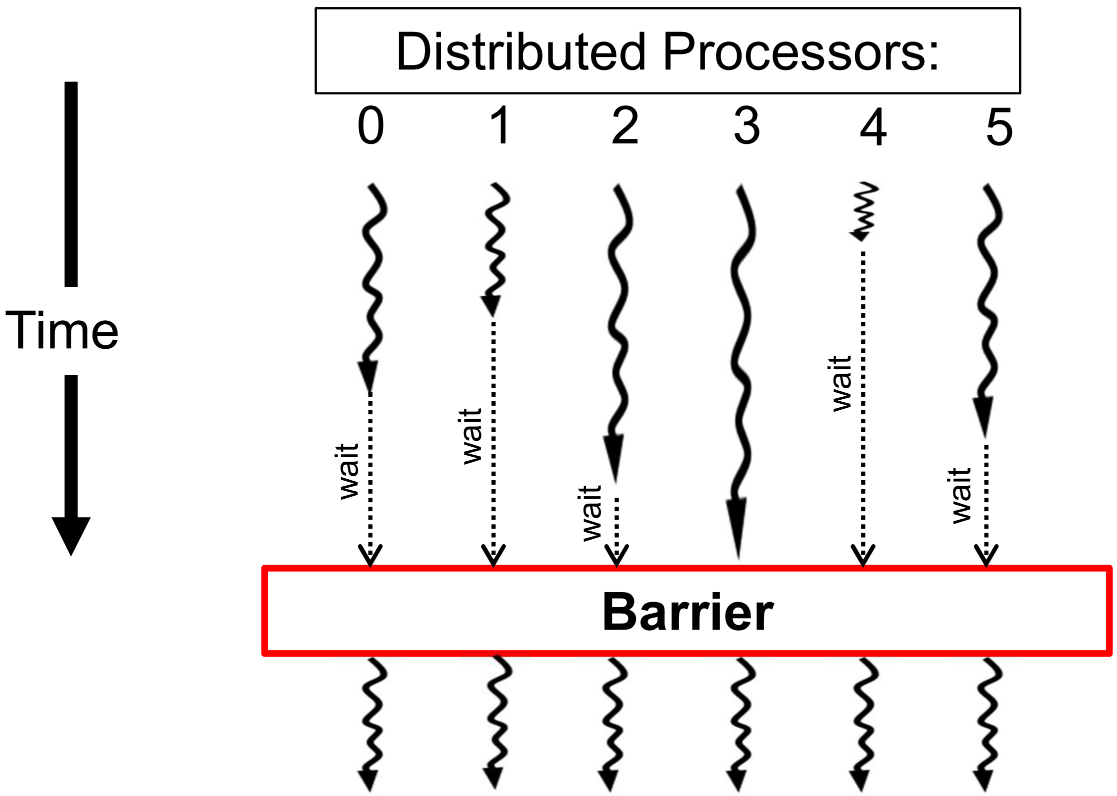

```
// gratuitous use of a reduce -- average all the temperatures:

        float partialSum = 0.;
        for( int i = 0; i < PPSize; i++ )
                partialSum += PPTemps[ i ];

        float globalSum = 0.;
        MPI_Reduce( &partialSum, &globalSum, 1, MPI_FLOAT, MPI_SUM, THEBOSS, MPI_COMM_WORLD );

        if( me == THEBOSS )
                fprintf( stderr, "Average temperature = %f\n", globalSum/(float)NUMELEMENTS );
```



Reduction

MPI Barriers

MPI_Barrier(MPI_COMM_WORLD);



All CPUs must execute the call to MPI_Barrier( ) before any of the CPUs can move past it. That is, each CPU's MPI_Barrier( ) blocks until all CPUs execute a call to MPI_Barrier( ).

MPI Derived Types

Idea: In addition to types MPI_INT, MPI_FLOAT, etc., allow the creation of new MPI types so that you can transmit an "array of structures".

Reason: There is significant overhead with each transmission. Better to send one entire array of structures instead of sending several arrays separately.

```
MPI_Type_create_struct( count, blocklengths, displacements, types, datatype );
```

```
struct point
{
        int     pointSize;
        float   x, y, z;
};
```

```
MPI_Datatype  MPI_POINT;
int blocklengths[ ]    = { 1, 1, 1, 1 };
int displacements[ ] = { 0, 4, 8, 12 };
MPI_type types[ ]    = { MPI_INT,  MPI_FLOAT,  MPI_FLOAT,  MPI_FLOAT );

MPI_Type_create_struct( 4, blocklengths, displacements, types, &MPI_POINT );
```

You can now use **MPI_POINT** everywhere you could have used **MPI_INT, MPI_FLOAT**,etc.

MPI Timing

```
double MPI_Wtick( );
```

Returns the resolution of the clock, in seconds.

```
double MPI_Wtime( );
```

Returns the time, in seconds, since "some time in the past".

Warning: the clocks on the different CPUs are not guaranteed to be synchronized!

MPI Status-Checking

Some MPI calls have a **&status** in their argument list.

The **status** argument is declared to be of type **MPI_Status**, which is actually a struct:

```
typedef struct _MPI_Status
{
        int  MPI_SOURCE;
        int  MPI_TAG;
        int  MPI_ERROR;
} MPI_Status;
```

- MPI_SOURCE is the rank of the node who sent this
- MPI_TAG is the tag used during the send
- MPI_ERROR is the error number that occurred

Example:

```
MPI_Status  status;
MPI_Recv( myData, MYDATA_SIZE, MPI_CHAR, THEBOSS, MPI_ANY_TAG, MPI_COMM_WORLD,
&status );

fprintf( stderr, "Rank = %d, Tag = %d, Error Code = %d\n",
        status.MPI_SOURCE, status.MPI_TAG, status.MPI_ERROR );
```

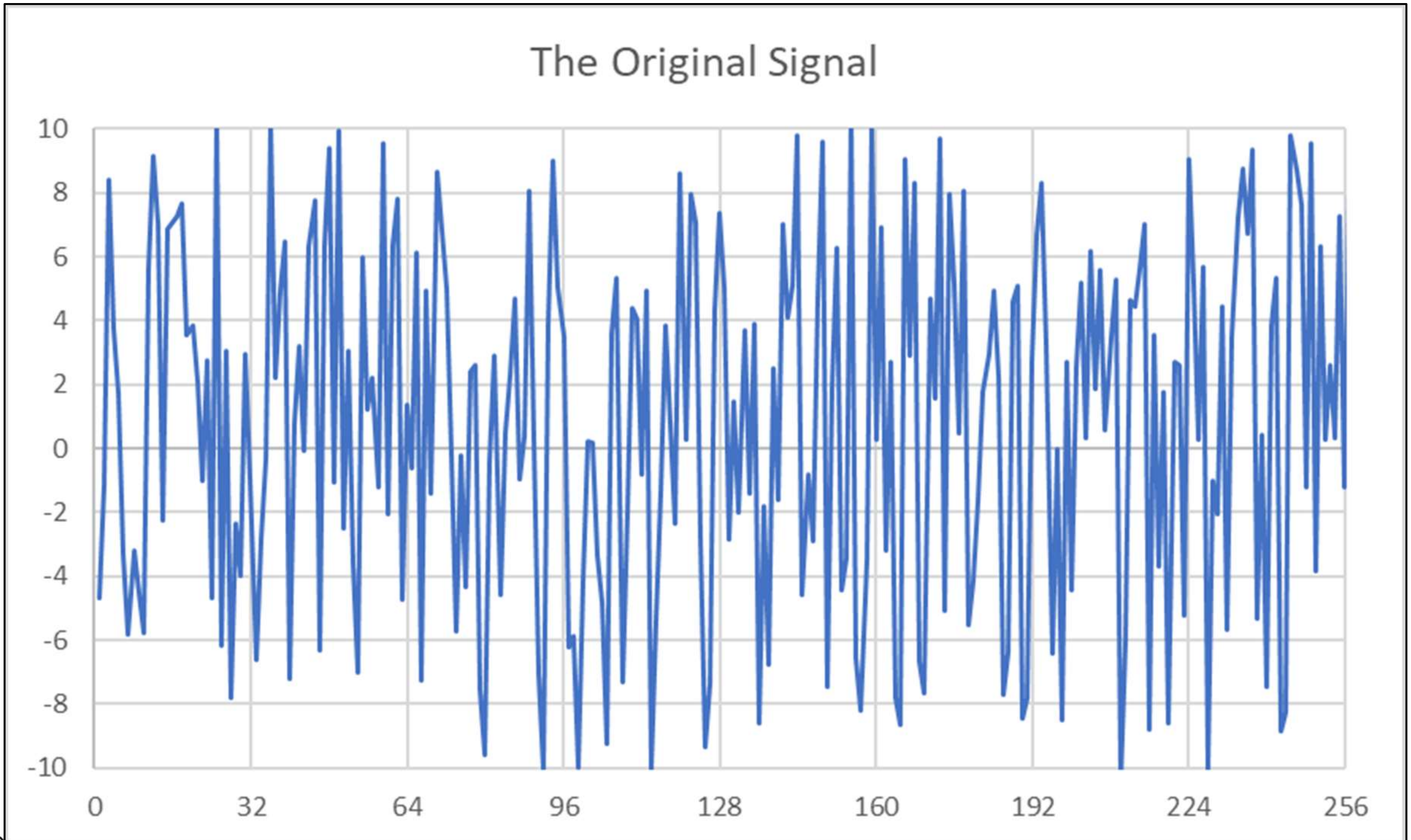
MPI Error Codes

MPI_SUCCESS	No error
MPI_ERR_BUFFER	Invalid buffer pointer
MPI_ERR_COUNT	Invalid count argument
MPI_ERR_TYPE	Invalid datatype argument
MPI_ERR_TAG	Invalid tag argument
MPI_ERR_COMM	Invalid communicator
MPI_ERR_RANK	Invalid rank
MPI_ERR_REQUEST	Invalid request (handle)
MPI_ERR_ROOT	Invalid root
MPI_ERR_GROUP	Invalid group
MPI_ERR_OP	Invalid operation
MPI_ERR_TOPOLOGY	Invalid topology
MPI_ERR_DIMS	Invalid dimension argument
MPI_ERR_ARG	Invalid argument of some other kind
MPI_ERR_UNKNOWN	Unknown error
MPI_ERR_TRUNCATE	Message truncated on receive
MPI_ERR_OTHER	Known error not in this list
MPI_ERR_INTERN	Internal MPI (implementation) error
MPI_ERR_IN_STATUS	Error code is in status
MPI_ERR_PENDING	Pending request

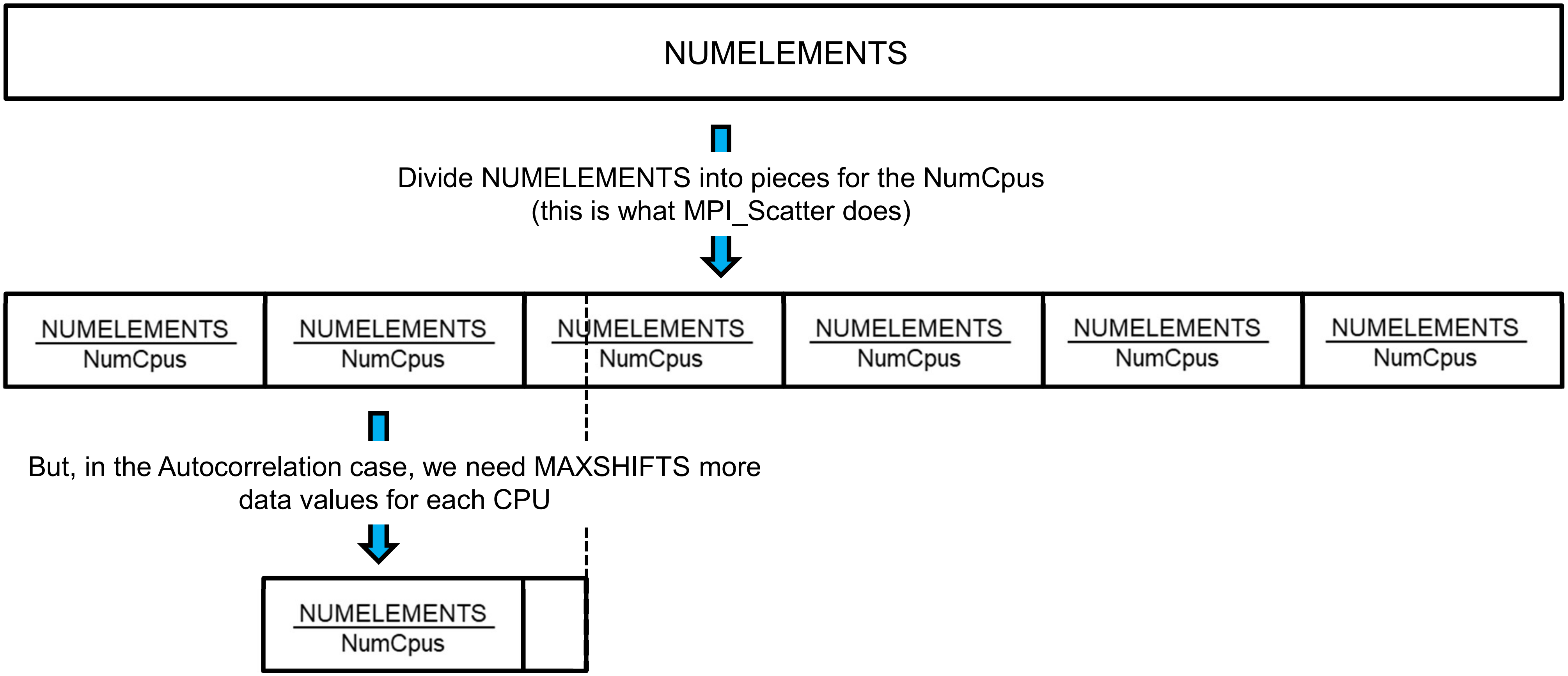
MPI_ERR_KEYVAL	Invalid keyval has been passed
MPI_ERR_NO_MEM	MPI_ALLOC_MEM failed because memory is exhausted
MPI_ERR_BASE	Invalid base passed to MPI_FREE_MEM
MPI_ERR_INFO_KEY	Key longer than MPI_MAX_INFO_KEY
MPI_ERR_INFO_VALUE	Value longer than MPI_MAX_INFO_VAL
MPI_ERR_INFO_NOKEY	Invalid key passed to MPI_INFO_DELETE
MPI_ERR_SPAWN	Error in spawning processes
MPI_ERR_PORT	Invalid port name passed to MPI_COMM_CONNECT
MPI_ERR_SERVICE	Invalid service name passed to MPI_UNPUBLISH_NAME
MPI_ERR_NAME	Invalid service name passed to MPI_LOOKUP_NAME
MPI_ERR_WIN	Invalid win argument
MPI_ERR_SIZE	Invalid size argument
MPI_ERR_DISP	Invalid disp argument
MPI_ERR_INFO	Invalid info argument
MPI_ERR_LOCKTYPE	Invalid locktype argument
MPI_ERR_ASSERT	Invalid assert argument
MPI_ERR_RMA_CONFLICT	Conflicting accesses to window
MPI_ERR_RMA_SYNC	Wrong synchronization of RMA calls

MPI_ERR_FILE	Invalid file handle
MPI_ERR_NOT_SAME	Collective argument not identical on all processes, or collective routines called in a different order by different processes
MPI_ERR_AMODE	Error related to the amode passed to MPI_FILE_OPEN
MPI_ERR_UNSUPPORTED_DATAREP	Unsupported datarep passed to MPI_FILE_SET_VIEW
MPI_ERR_UNSUPPORTED_OPERATION	Unsupported operation, such as seeking on a file which supports sequential access only
MPI_ERR_NO_SUCH_FILE	File does not exist
MPI_ERR_FILE_EXISTS	File exists
MPI_ERR_BAD_FILE	Invalid file name (e.g., path name too long)
MPI_ERR_ACCESS	Permission denied
MPI_ERR_NO_SPACE	Not enough space
MPI_ERR_QUOTA	Quota exceeded
MPI_ERR_READ_ONLY	Read-only file or file system
MPI_ERR_FILE_IN_USE	File operation could not be completed, as the file is currently open by some process
MPI_ERR_DUP_DATAREP	Conversion functions could not be registered because a data representation identifier that was already defined was passed to MPI_REGISTER_DATAREP
MPI_ERR_CONVERSION	An error occurred in a user supplied data conversion function.
MPI_ERR_IO	Other I/O error
MPI_ERR_LASTCODE	Last error code

Example: Autocorrelation



Autocorrelation – More than Just a Scatter



Autocorrelation – How the Shifting Works