The Parallel Programming world beyond OpenMP

Tim Mattson

Intel Corp.

timothy.g.mattson@ intel.com

Legal Disclaimer & Optimization Notice

- INFORMATION IN THIS DOCUMENT IS PROVIDED “AS IS”. NO LICENSE, EXPRESS OR IMPLIED, BY ESTOPPEL OR OTHERWISE, TO ANY INTELLECTUAL PROPERTY RIGHTS IS GRANTED BY THIS DOCUMENT. INTEL ASSUMES NO LIABILITY WHATSOEVER AND INTEL DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO THIS INFORMATION INCLUDING LIABILITY OR WARRANTIES RELATING TO FITNESS FOR A PARTICULAR PURPOSE, MERCHANTABILITY, OR INFRINGEMENT OF ANY PATENT, COPYRIGHT OR OTHER INTELLECTUAL PROPERTY RIGHT.
- Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products.
- Copyright © , Intel Corporation. All rights reserved. Intel, the Intel logo, Xeon, Xeon Phi, Core, VTune, and Cilk are trademarks of Intel Corporation in the U.S. and other countries.

Optimization Notice

Intel’s compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice revision #20110804

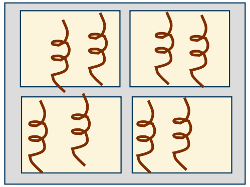
Disclaimer

- The views expressed in this talk are those of the speaker and not his employer.
- If I say something “smart” or worthwhile:
  - Credit goes to the many smart people I work with.
- If I say something stupid…
  - It’s my own fault

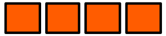
I work in Intel’s research labs. I don’t build products. Instead, I get to poke into dark corners and think silly thoughts… just to make sure we don’t miss any great ideas.

Hence, my views are by design far “off the roadmap”.

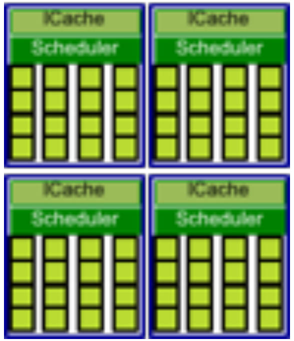
Hardware is diverse … and its only getting worse!!!



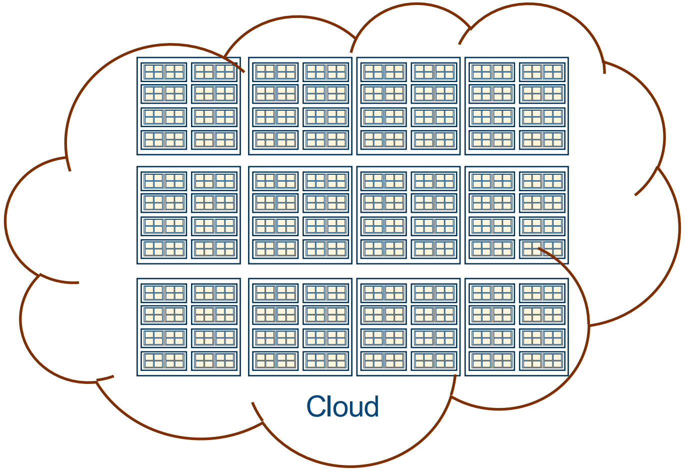
CPU



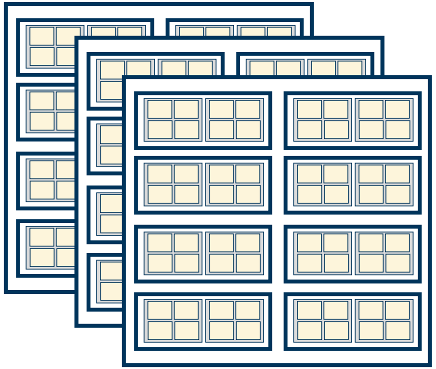
SIMD/Vector



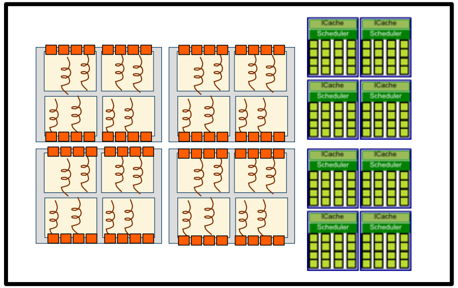
GPU



Cloud



Cluster



Heterogeneous node

The Big Three

- In HPC, 3 programming environments dominate … covering the major classes of hardware.
  - **MPI**: distributed memory systems … though it works nicely on shared memory computers.
  - **OpenMP**: Shared memory systems … more recently, GPGPU too.
  - **CUDA**, **OpenCL**, **Sycl**, **OpenACC**, **OpenMP** … : GPU programming (use CUDA if you don't mind locking yourself to a single vendor … it is a really nice programming model)

You are all OpenMP experts and know a great deal about multithreading

- Even if you don't plan to spend much time programming with these systems … a well rounded HPC programmer should know what they are and how they work.

The Big Three

If you don’t know MPI, you aren’t really an HPC programmer!

- In HPC, 3 programming environments dominate … covering the major classes of hardware.
  - **MPI**: distributed memory systems … though it works nicely on shared memory computers.
  - **OpenMP**: Shared memory systems … more recently, GPGPU too.
  - **CUDA**, **OpenCL**, **Sycl**, **OpenACC**, **OpenMP** … : GPU programming (use CUDA if you don’t mind locking yourself to a single vendor … it is a really nice programming model)
- Even if you don’t plan to spend much time programming with these systems … a well rounded HPC programmer should know what they are and how they work.

A “Hands-on” Introduction to MPI

Tim Mattson Intel Corp. timothy.g.mattson@ intel.com

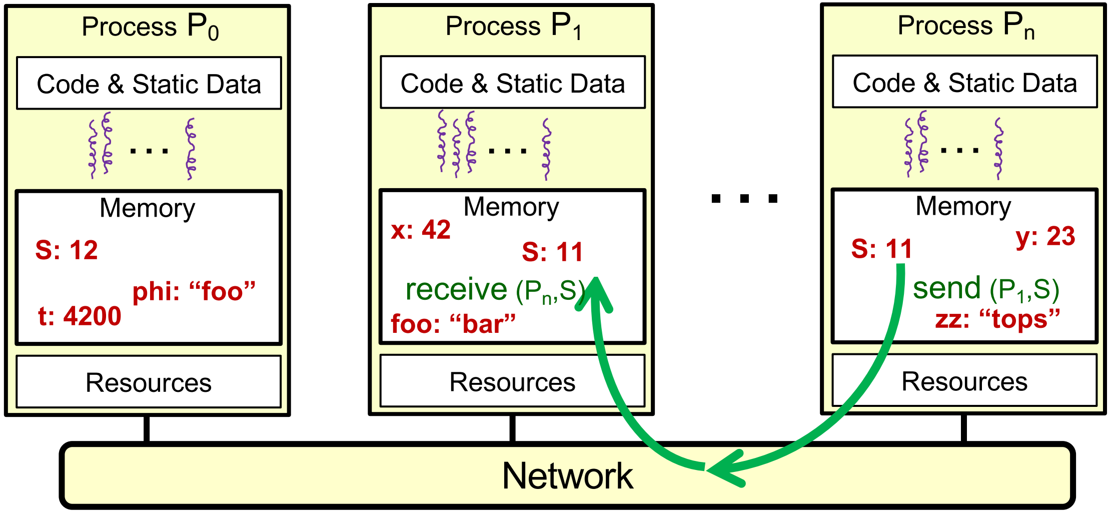
* The name “MPI” is the property of the MPI forum (http://www.mpi-forum.org).

Outline

- MPI and distributed memory systems
- The Bulk Synchronous Pattern and MPI collective operations
- Introduction to message passing
- The diversity of message passing in MPI
- Geometric Decomposition and MPI
- Concluding Comments

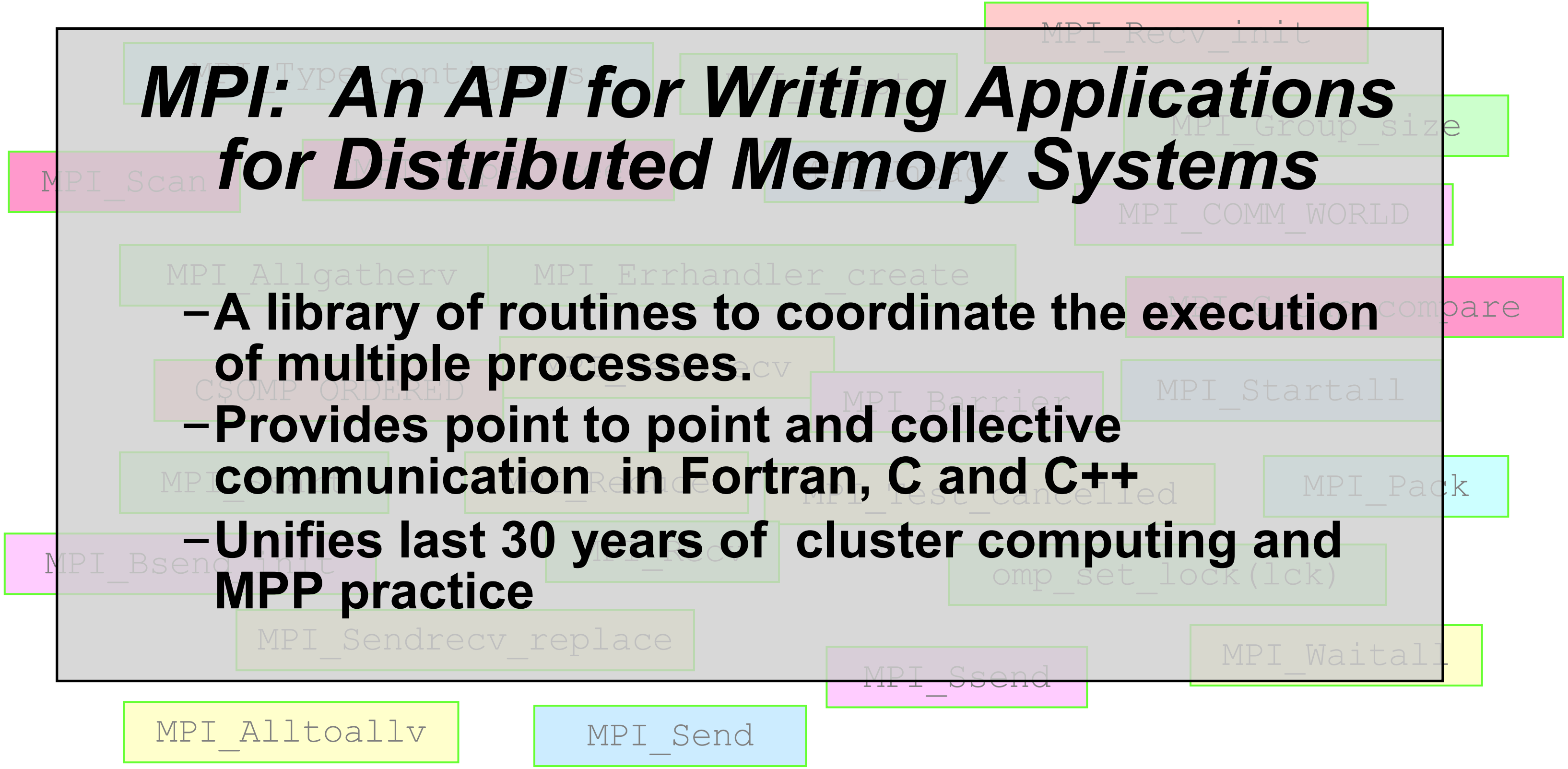
Execution Model: Distributed memory, CSP*

- Program consists of a collection of **named** processes.
  - Number of processes almost always fixed at program startup time
  - Local address space per node -- NO physically shared memory.
- Processes communicate by explicit send/receive pairs
  - Coordination is implicit in every communication event.
  - MPI (Message Passing Interface) is the most commonly used API



*CSP: communicating sequential processes

Parallel API's: MPI, the Message Passing Interface



MPI: An API for Writing Applications for Distributed Memory Systems

- A library of routines to coordinate the execution of multiple processes.
- Provides point to point and collective communication in Fortran, C and C++
- Unifies last 30 years of cluster computing and MPP practice

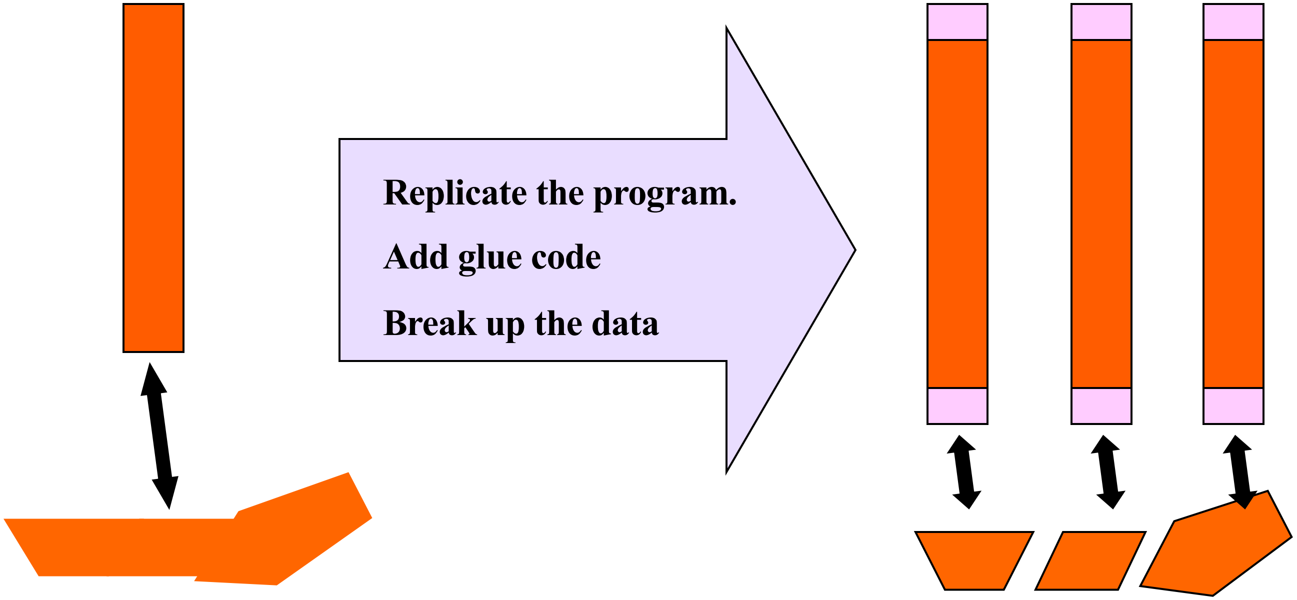
How do people use MPI? The SPMD Design Pattern

•A single program working on a decomposed data set.

•Use Node ID and numb of nodes to split up work between processes

• Coordination by passing messages.

A sequential program working on a data set



Replicate the program.

Add glue code

Break up the data

Running MPI programs

The programs **mpirun** or **mpiexec** are largely equivalent and are used to launch a job on the processes across a cluster. On our cluster, we'll use **mpirun**

- MPI implementations include a way to start "P processes" on the system.

- For MPIch (the most common MPI implementation), this is done with the mpirun command:

```
> mpirun –n P ./a.out
```

Run the program locally as P processes

- There are many options for mpirun.

```
> mpirun –hostfile hostfile –n P ./a.out
```

Run the program as P processes on the nodes from hostfile.

A hostfile has node names one to a line followed by a colon and the number of available processors

```
> mpirun –h
```

Ask mpirun for information about mpirun options.

MPIch from Argonne national lab: https://www.mpich.org/

Building and running MPI programs at PSFC

- Log in to a gpu node, one hour request for one node to compile:

```
srun --nodes=1 --ntasks=1 --time=01:00:00  --pty /bin/bash
```

- Then compile

```
mpicc/mpif90/mpic++  -o program program.cc/f90/C
```

- To run, exit current shell, then

```
srun --nodes=2 --ntasks-per-node=3 --time=00:01:00 ./program
```

- Will run 6 processes over 2 nodes.

Exercise: Hello world part 1

- Goal
  - To confirm that you can run a program in parallel.
- Program
  - Write a program that prints “hello world” to the screen.

- Log in to a the PSFC cluster. Compile and build the program on the log-in node

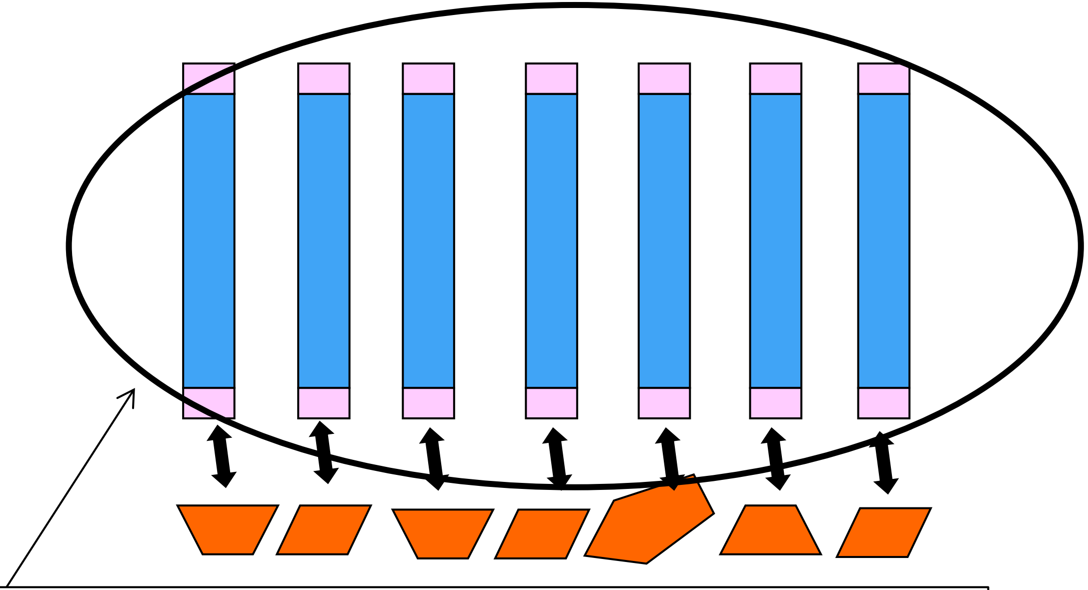
- Submit to run on the GPU cluster

```
srun --nodes=2 --ntasks-per-node=3 --time=00:01:00 ./program
```

- Will run 6 processes over 2 nodes

An MPI program at runtime

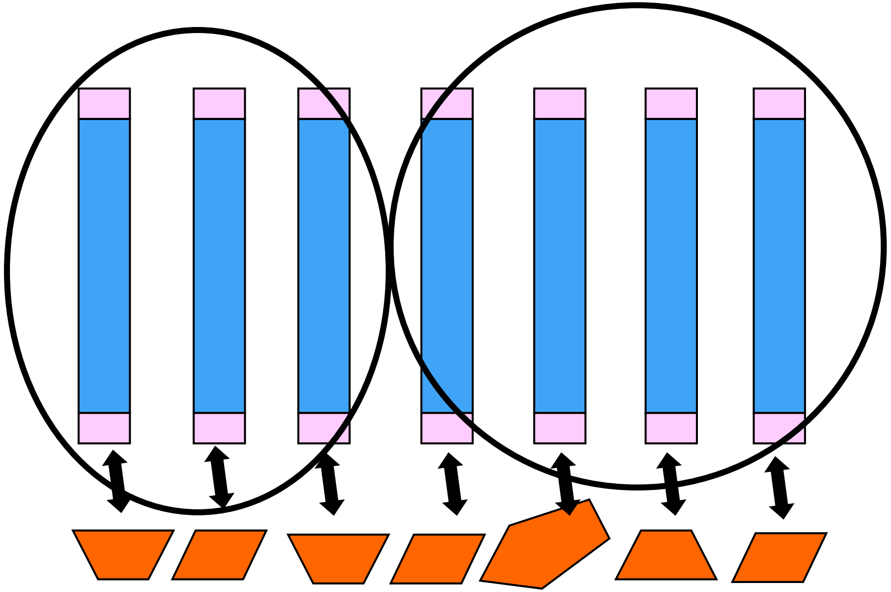
- Typically, when you run an MPI program, multiple processes all running the same program are launched … working on their own block of data.



The collection of processes involved in a computation is called "a **process group**"

An MPI program at runtime

- Typically, when you run an MPI program, multiple processes all running the same program are launched … working on their own block of data.



You can dynamically split a **process group** into multiple subgroups to manage how processes are mapped onto different tasks

MPI Hello World

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
   int rank, size;
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   printf( "Hello from process %d of %d\n",
                              rank, size );
   MPI_Finalize();
   return 0;
}
```

Initializing and finalizing MPI

```
int MPI_Init (int* argc, char* argv[])
```

- Initializes the MPI library … called before any other MPI functions.
- agrc and argv are the command line args passed from main()

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
   int rank, size;
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   printf( "Hello from process %d of %d\n",
                            rank, size );
   MPI_Finalize();
   return 0;
}
```

```
int MPI_Finalize (void)
```

- Frees memory allocated by the MPI library … close every MPI program with a call to MPI_Finalize

How many processes are involved?

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

- returns the number of processes in the process group

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
   int rank, size;
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   printf( "Hello from process %d of %d\n",
                               rank, size );
   MPI_Finalize();
   return 0;
}
```

How many processes are involved?

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

- returns the number of processes in the process group

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
   int rank, size;
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   printf( "Hello from process %d of %d\n",
                               rank, size );
   MPI_Finalize();
   return 0;
}
```

What is MPI_COMM_WORLD?

It's a communicator (of type MPI_Comm)

MPI_COMM_WORLD defines a name space for the communication events inside MPI. This includes the process group and any other meta-data about the set of cooperating processes.

How many processes are involved?

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

- returns the number of processes in the process group

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
   int rank, size;
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   printf( "Hello from process %d of %d\n",
                                rank, size );
   MPI_Finalize();
   return 0;
}
```

Other than init() and finalize(), every MPI function has a communicator.

You can build your own communicators to support libraries or segregate operations into different process groups.

But most of us just use the one global communicator, MPI_COMM_WORLD

Which process “am I” (the rank)

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

- **`MPI_Comm_rank`** An integer ranging from 0 to “(num of procs)-1”

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
   int rank, size;
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   printf( "Hello from process %d of %d\n",
                                rank, size );
   MPI_Finalize();
   return 0;
}
```

Running the program

- On a 4 node cluster, I’d run this program (hello) as:

  > mpiexec –n 4 hello

- What would this program would output?

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
   int rank, size;
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   printf( "Hello from process %d of %d\n",
                              rank, size );
   MPI_Finalize();
   return 0;
}
```

Exercise: Hello world part 2

- Goal
  - To confirm that you can run an MPI program on our cluster
- Program
  - Write a program that prints “hello world” to the screen.
  - Modify it to run as an MPI program … with each printing “hello world” and its rank

- Log in to a gpu node, one hour request for one node to compile:

```
srun --nodes=1 --ntasks=1 --time=01:00:00  --pty /bin/bash
```

- Then compile

```
mpicc/mpif90/mpic++  -o program program.cc/f90/C
```

- To run, exit current shell, then

```
srun --nodes=2 --ntasks-per-node=3 --time=00:01:00 ./program
```

- Will run 6 processes over 2 nodes.

```
#include <mpi.h>
int size, rank, argc;   char **argv;
MPI_Init (&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Finalize();
Char name[MPI_MAX_PROCESSOR_NAME];
int MPI_Get_processor_name( char *name, int *resultLen )
```

Get the name of the node you’re running on

Running the program

- On a 4 node cluster, I’d run this program (hello) as:

```
> mpirun –n 4 hello
Hello from process 1 of 4
Hello from process 2 of 4
Hello from process 0 of 4
Hello from process 3 of 4
```

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
   int rank, size;
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
   printf( "Hello from process %d of %d\n",
                              rank, size );
   MPI_Finalize();
   return 0;
}
```

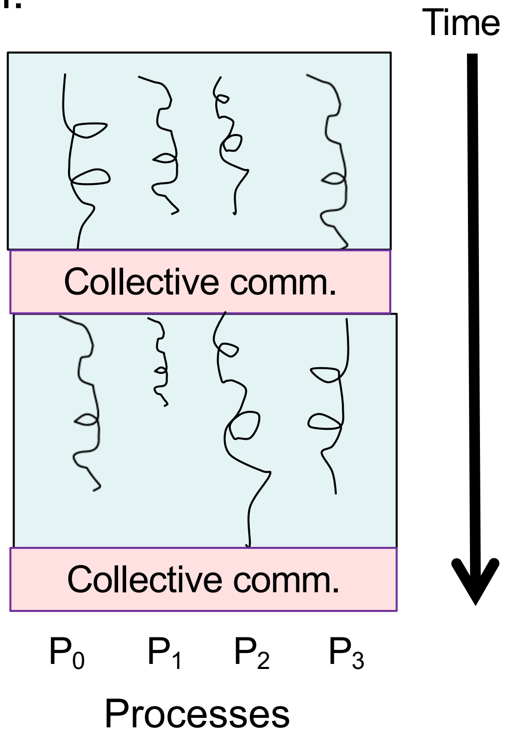
Outline

- MPI and distributed memory systems
- ➡ The Bulk Synchronous Pattern and MPI collective operations
- Introduction to message passing
- The diversity of message passing in MPI
- Geometric Decomposition and MPI
- Concluding Comments

A typical pattern with MPI Programs

- Many MPI applications directly call few (if any) message passing routines. They use the following very common pattern:
  - Use the Single Program Multiple Data pattern
  - Each process maintains a local view of the global data
  - A problem broken down into phases each of which is composed of two subphases:
    - Compute on local view of data
    - Communicate to update global view on all processes (collective communication).
  - Continue phases until complete

This is a subset or the SPMD pattern sometimes referred to as the Bulk Synchronous pattern.



Collective Communication: Reduction

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

Returns MPI_SUCCESS if there were no errors

- **`MPI_Reduce`** performs specified reduction operation (**op**) on the **count** values in **sendbuf** from all processes in communicator. Places result in **recvbuf** on the process with rank **root** only.

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

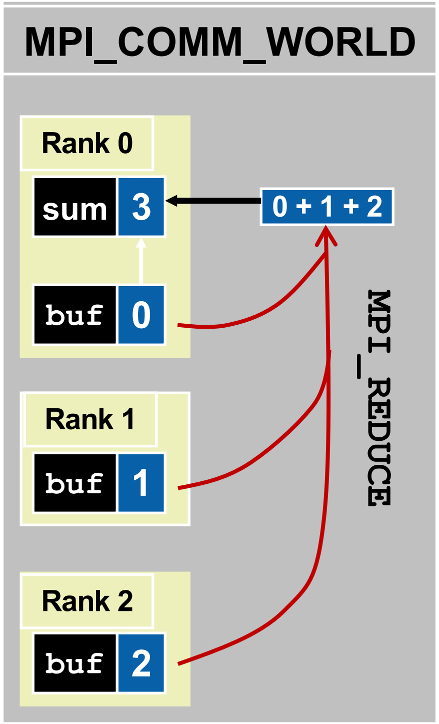
*This is a subset of available MPI types

Operation	**Function**
`MPI_SUM`	Summation
`MPI_PROD`	Product
`MPI_MIN`	Minimum value
`MPI_MINLOC`	Minimum value and location
`MPI_MAX`	Maximum value
`MPI_MAXLOC`	Maximum value and location
`MPI_LAND`	Logical AND

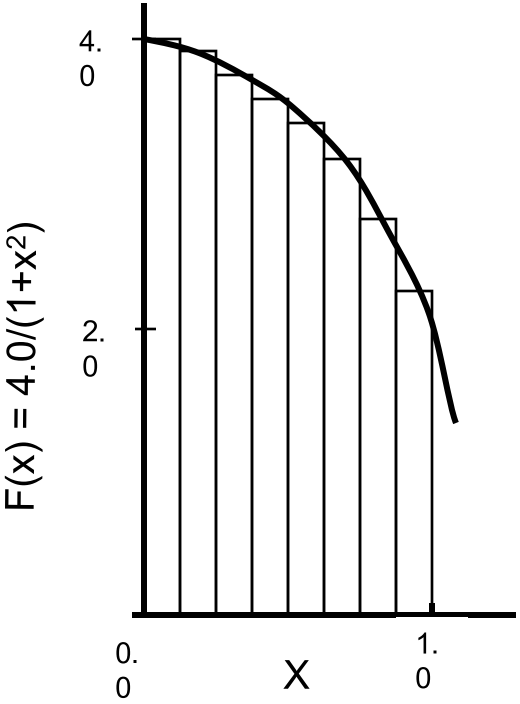
Operation	**Function**
`MPI_BAND`	Bitwise AND
`MPI_LOR`	Logical OR
`MPI_BOR`	Bitwise OR
`MPI_LXOR`	Logical exclusive OR
`MPI_BXOR`	Bitwise exclusive OR
User-defined	It is possible to define new reduction operations

MPI_REDUCE Example

```
#include <mpi.h>

int main(int argc, char* argv[]) {
  int buf, sum, nprocs, myrank;

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

  sum = 0;
  msg = myrank;

  MPI_Reduce(&buf, &sum, 1, MPI_INT,
             MPI_SUM, 0, MPI_COMM_WORLD);

  MPI_Finalize();
}
```



Example Problem: Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\, dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

PI Program: an example

```
static long num_steps = 100000;
double step;
void main ()
{       int i;      double x, pi, sum = 0.0;

        step = 1.0/(double) num_steps;
           x = 0.5 * step;
        for (i=0;i<= num_steps; i++){
              x+=step;
              sum += 4.0/(1.0+x*x);
        }
        pi = step * sum;
}
```

Exercise: Pi Program

- Goal
  - To write a simple Bulk Synchronous, SPMD program
- Program
  - Start with the provided “pi program” and using an MPI reduction, write a parallel version of the program. Explore its scalability on your system.

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

MPI_Op	Function
MPI_SUM	Summation

```
#include <mpi.h>
int size, rank, argc;   char **argv;
MPI_Init (&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Finalize();
```

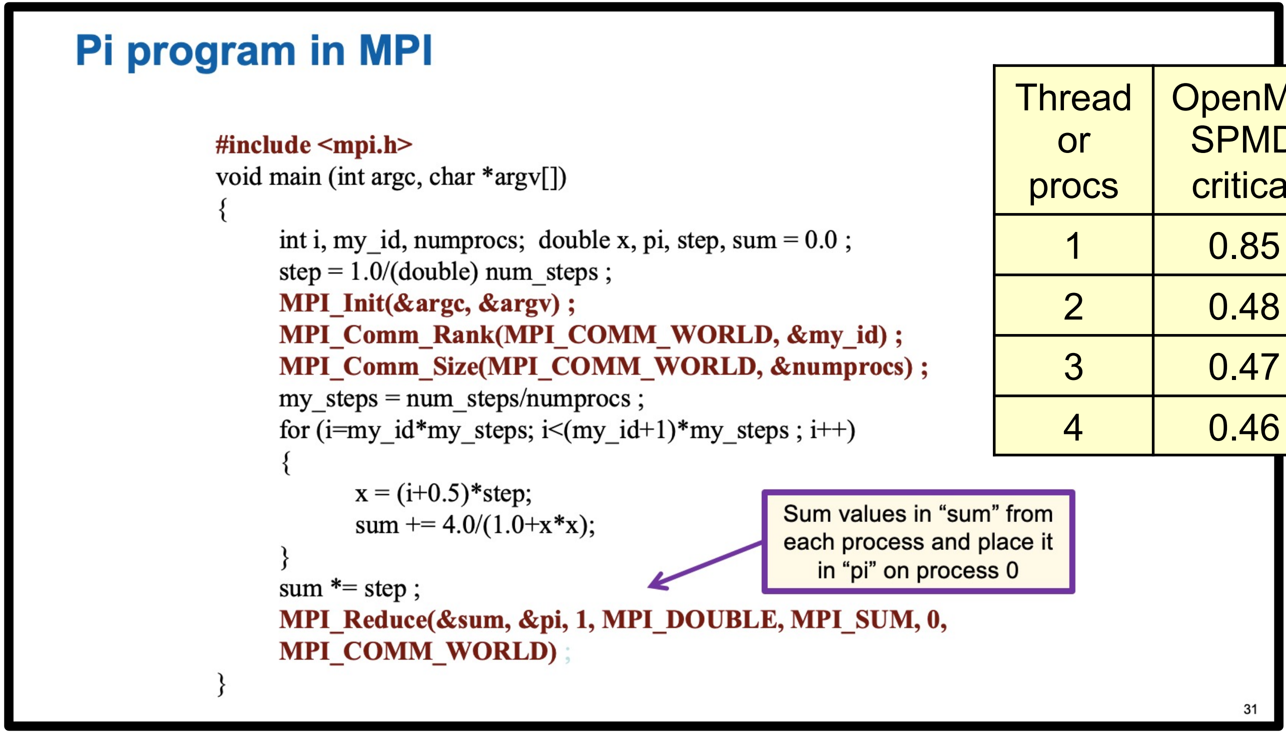
MPI Data Type	C Data Type
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	int
MPI_LONG	long

Pi program in MPI

```
#include <mpi.h>
void main (int argc, char *argv[])
{
        int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
        MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
        my_steps = num_steps/numprocs ;
        for (i=my_id*my_steps; i<(my_id+1)*my_steps ; i++)
        {
                x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
        sum *= step ;
        MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
        MPI_COMM_WORLD) ;
}
```

Sum values in “sum” from each process and place it in “pi” on process 0

MPI Pi program performance



Pi program in MPI

```
#include <mpi.h>
void main (int argc, char *argv[])
{
        int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
        MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
        my_steps = num_steps/numprocs ;
        for (i=my_id*my_steps; i<(my_id+1)*my_steps ; i++)
        {
                x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
        sum *= step ;
        MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
        MPI_COMM_WORLD) ;
}
```

Sum values in “sum” from each process and place it in “pi” on process 0

Thread or procs	OpenMP SPMD critical	OpenMP PI Loop	MPI
1	0.85	0.43	0.84
2	0.48	0.23	0.48
3	0.47	0.23	0.46
4	0.46	0.23	0.46

Note: OMP loop used a Blocked loop distribution. The others used a cyclic distribution. Serial .. 0.43.

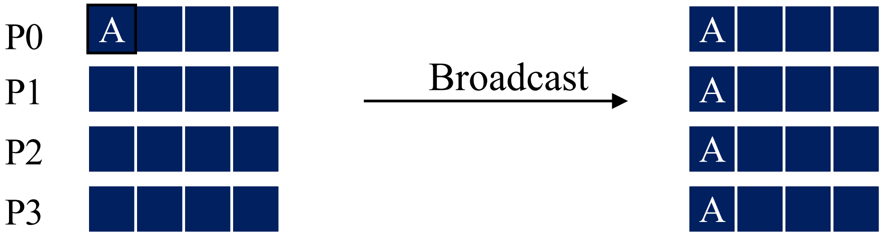
*Intel compiler (icpc) with –O3 on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core^{TM} i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

MPI Collective Routines

- Collective communications: called by all processes in the group to create a global result and share with all participating processes.
  - **`Allgather, Allgatherv, Allreduce, Alltoall, Alltoallv, Bcast, Gather, Gatherv, Reduce, Reduce_scatter, Scan, Scatter, Scatterv`**
- Notes:
  - **`Allreduce, Reduce, Reduce_scatter`**, and **`Scan`** use the same set of built-in or user-defined combiner functions.
  - Routines with the "**`All`**" prefix deliver results to all participating processes
  - Routines with the "**`v`**" suffix allow chunks to have different sizes
- Global synchronization is available in MPI
  - **`MPI_Barrier( comm )`**
- Blocks until all processes in the group of the communicator **`comm`** call it.

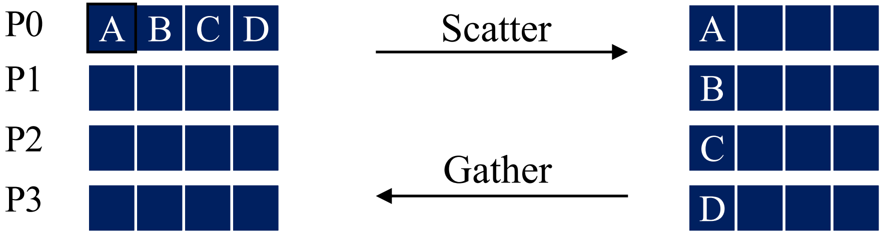
Collective Data Movement

Take a value from P0 and give a copy to P1, P2 and P3



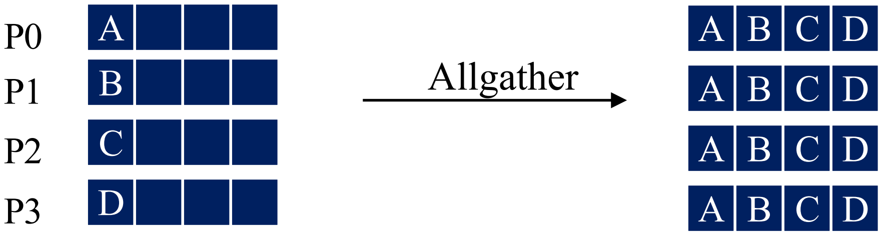
Scatter an array on P0 to P1, P2, and P3

Gather values from P1, P2, and P3 into an array on P0



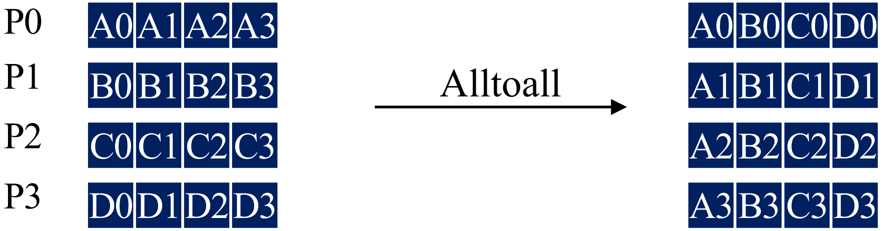
More Collective Data Movement

Take a chunk from each P and gather into a single array on each P



P0	A				A	B	C	D
P1	B				A	B	C	D
P2	C				A	B	C	D
P3	D				A	B	C	D

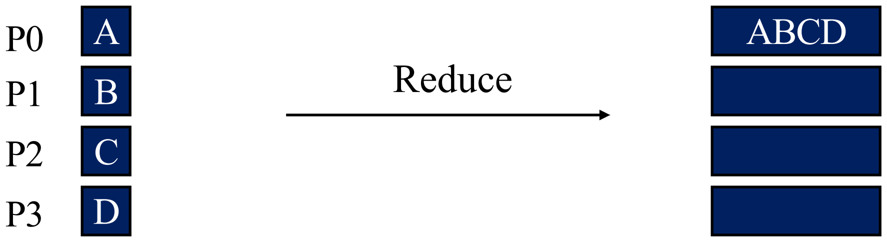
Take arrays on each P and spread them out to arrays on each P



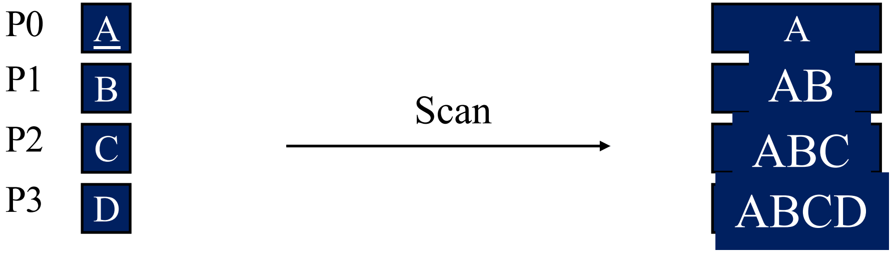
P0	A0	A1	A2	A3	A0	B0	C0	D0
P1	B0	B1	B2	B3	A1	B1	C1	D1
P2	C0	C1	C2	C3	A2	B2	C2	D2
P3	D0	D1	D2	D3	A3	B3	C3	D3

Collective Computation

Take values on each P and combine them with an op (such as add) into a single value on one P.



Take values on each P and combine them with a scan operation and spread the scan array out among all P.



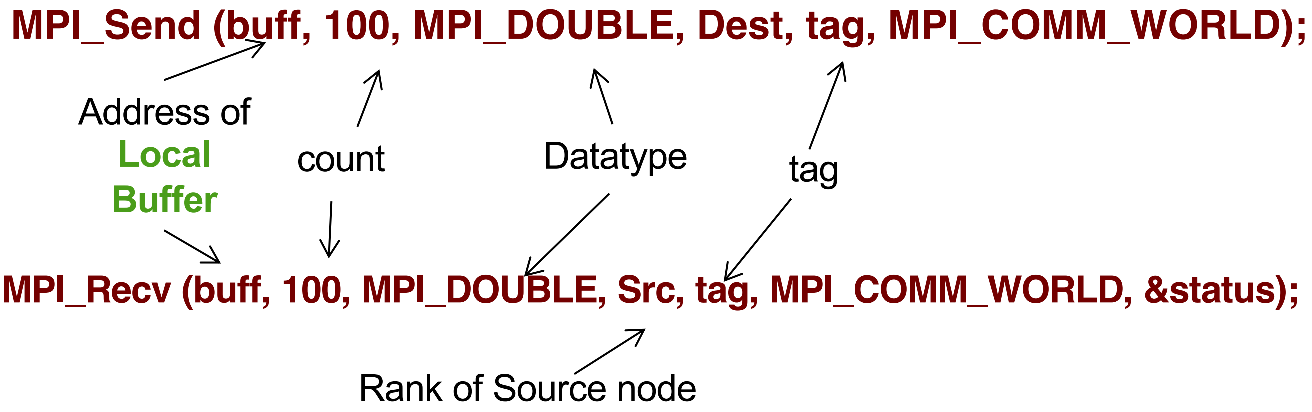
Outline

- MPI and distributed memory systems
- The Bulk Synchronous Pattern and MPI collective operations
- ➡ Introduction to message passing
- The diversity of message passing in MPI
- Geometric Decomposition and MPI
- Concluding Comments

Sending and receiving messages

- Pass a buffer which holds “count” values of MPI_TYPE
- The data in a message to send or receive is described by a triple:
  - **(address, count, datatype)**

- The receiving process identifies messages with the double :
  - **(source, tag)**
- Where:
  - Source is the rank of the sending process
  - Tag is a user-defined integer to help the receiver keep track of different messages from a single source



MPI_Send (buff, 100, MPI_DOUBLE, Dest, tag, MPI_COMM_WORLD);

Address of **Local Buffer**

count

Datatype

tag

MPI_Recv (buff, 100, MPI_DOUBLE, Src, tag, MPI_COMM_WORLD, &status);

Rank of Source node

Sending and Receiving messages: More Details

```
int MPI_Send (void* buf, int count,
    MPI_Datatype datatype, int dest,
    int tag, MPI_Comm comm)

int MPI_Recv (void* buf, int count,
    MPI_Datatype datatype, int source,
    int tag, MPI_Comm comm,
    MPI_Status* status)
```

MPI_Status is a variable that contains information about the message that is received. We can use it to find out information about the received message. The most common usage is to find out how many items were in the message:

```
MPI_Status MyStat;      int count;      float buff[4];
int ierr = MPI_Recv(buf, 4, MPI_FLOAT, 2, 0, MPI_COMM_WORLD, &MyStat);   // receive message from node=2 with message tag = 0
If(ierr == MPI_SUCCESS) MPI_Get_Count(MyStat, MPI_FLOAT, &count);
```

For messages of a known size, we typically ignore the status, in which case use the parameter MPI_STATUS_IGNORE

```
int ierr = MPI_Recv(&buf, 4, MPI_FLOAT, 2, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

Exercise: Ping-Pong Program

- Goal
  - Measure the latency of our communication network.
- Program
  - Create a program to bounce a message between a pair of processes. Bounce the message back and forth multiple times and report the average one-way communication time. Figure out how to use this so called “ping-pong” program to measure the latency of communication on your system.

```
int MPI_Send (void* buf, int count,MPI_Datatype datatype, int dest,int tag, MPI_Comm comm)

int MPI_Recv (void* buf, int count,MPI_Datatype datatype, int source,int tag,
     MPI_Comm comm, MPI_Status* status)
```

```
#include <mpi.h>
int size, rank, argc;   char **argv;
MPI_Init (&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Comm_size (MPI_COMM_WORLD, &size);
double MPI_Wtime();
MPI_Finalize();
```

MPI Data Type	C Data Type
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	int
MPI_LONG	long

Solution: Ping-Pong Program

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define VAL 42
#define NREPS  10
#define TAG 5

int main(int argc, char **argv)  {
   int rank, size;
   double t0;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);

   int bsend = VAL;
   int brecv = 0;
   MPI_Status stat;
   if(rank == 0) t0 = MPI_Wtime();
   for(int i=0;i<NREPS; i++){
      if(rank == 0){
        MPI_Send(&bsend, 1, MPI_INT, 1, TAG, MPI_COMM_WORLD);
        MPI_Recv(&brecv, 1, MPI_INT, 1, TAG, MPI_COMM_WORLD, &stat);
        if(brecv != VAL)printf("error: interation %d %d != %d\n",i,brecv,VAL);
        brecv = 0;
      }
      else if(rank == 1){
        MPI_Recv(&brecv, 1, MPI_INT, 0, TAG, MPI_COMM_WORLD, &stat);
        MPI_Send(&bsend, 1, MPI_INT, 0, TAG, MPI_COMM_WORLD);
        if(brecv != VAL)printf("error: interation %d %d != %d\n",i,brecv,VAL);
        brecv = 0;
      }
   }
   if(rank == 0){
      double t = MPI_Wtime() - t0;
      double lat = t/(2*NREPS);
      printf(" lat = %f seconds\n",(float)lat);
   }
   MPI_Finalize();
}
```

MPI Data Types for C

MPI Data Type	C Data Type
MPI_BYTE	
MPI_CHAR	signed char
MPI_DOUBLE	double
MPI_FLOAT	float
MPI_INT	int
MPI_LONG	long
MPI_LONG_DOUBLE	long double
MPI_PACKED	
MPI_SHORT	short
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_UNSIGNED_CHAR	unsigned char

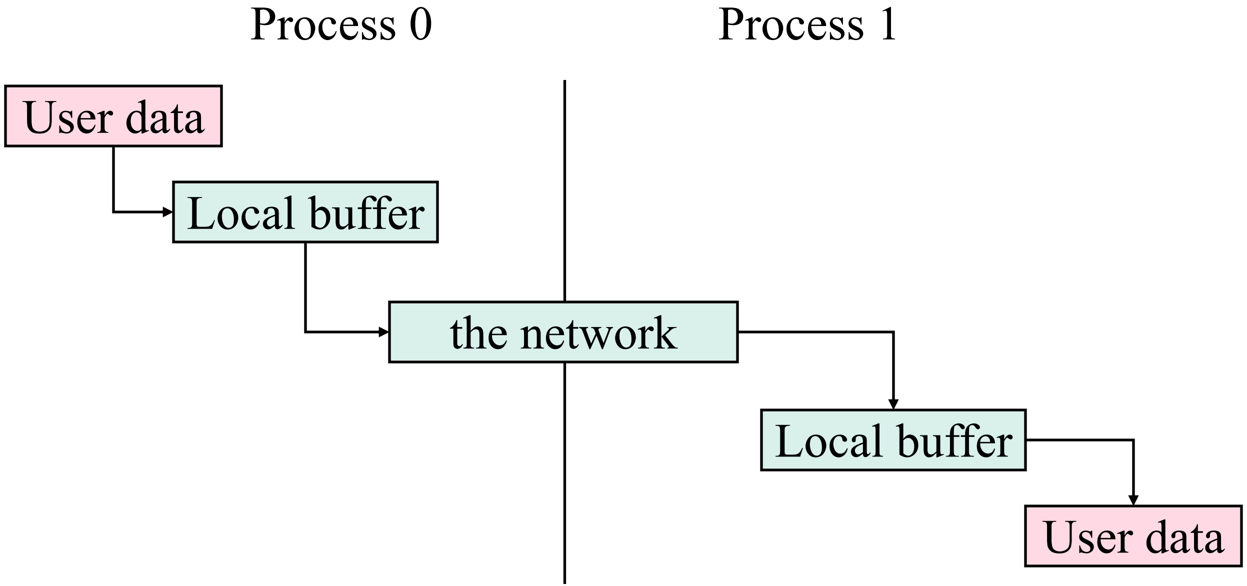
MPI provides predefined data types that must be specified when passing messages.

Outline

- MPI and distributed memory systems
- The Bulk Synchronous Pattern and MPI collective operations
- Introduction to message passing
- The diversity of message passing in MPI
- Geometric Decomposition and MPI
- Concluding Comments

Buffers

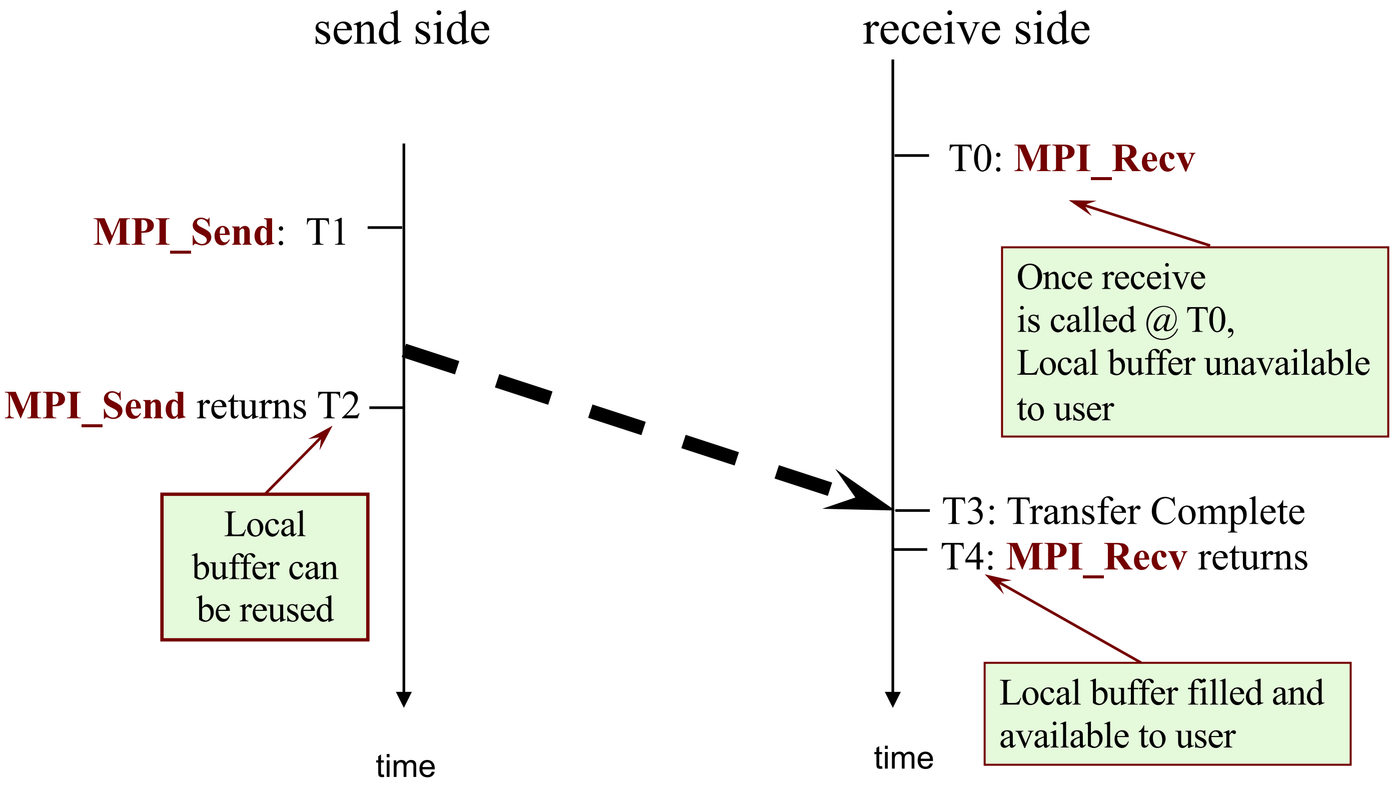
- Message passing is straightforward, but there are subtleties
  - Buffering and deadlock
  - Deterministic execution
  - Performance
- When you send data, where does it go? One possibility is:



Derived from slides provided by Bill Gropp of UIUC

Blocking Send-Receive Timing Diagram

(Receive before Send)



send side

receive side

T0: **MPI_Recv**

MPI_Send: T1

Once receive is called @ T0, Local buffer unavailable to user

MPI_Send returns T2

Local buffer can be reused

T3: Transfer Complete

T4: **MPI_Recv** returns

Local buffer filled and available to user

time

time

It is important to post the receive before sending, for highest performance.

Sources of Deadlocks

- Send a large message from process 0 to process 1
  - If there is insufficient storage at the destination, the send must wait for the user to provide the memory space (through a receive)
- What happens with this code?

Process 0	Process 1
`Send(1)`	`Send(0)`
`Recv(1)`	`Recv(0)`

- This code could deadlock … it depends on the availability of system buffers in which to store the data sent until it can be received

Slide source: based on slides from Bill Gropp, UIUC

Some Solutions to the “deadlock” Problem

- Order the operations more carefully:

Process 0	Process 1
`Send(1)`	**`Recv(0)`**
`Recv(1)`	**`Send(0)`**

- Supply receive buffer at same time as send:

Process 0	Process 1
`Sendrecv(1)`	**`Sendrecv(0)`**

Slide source: Bill Gropp, UIUC

More Solutions to the “unsafe” Problem

- Supply a sufficiently large buffer in the send function

Process 0	Process 1
`Bsend(1)`	**`Bsend(0)`**
`Recv(1)`	**`Recv(0)`**

- Use non-blocking operations:

Process 0	Process 1
`Isend(1)`	**`Isend(0)`**
`Irecv(1)`	**`Irecv(0)`**
`Waitall`	**`Waitall`**

Slide source: Bill Gropp, UIUC

Non-Blocking Communication

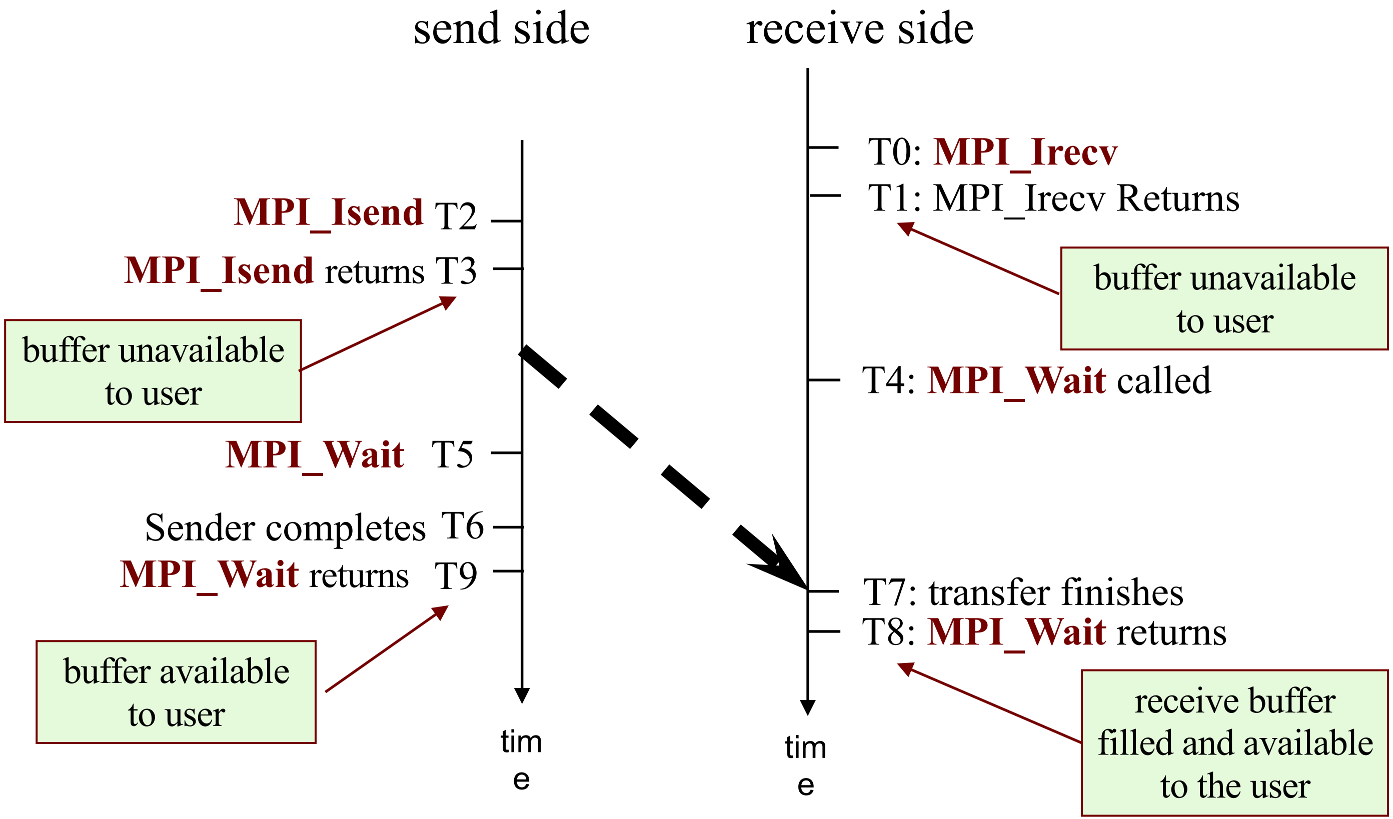
- Non-blocking operations return immediately and pass “request handles” that can be waited on and queried
  - **MPI_Isend(start, count, datatype, dest, tag, comm, request)**
  - **MPI_Irecv(start, count, datatype, src, tag, comm, request)**
  - **MPI_Wait(request, status)**
- One can also test without waiting using MPI_TEST
  - **MPI_Test(request, flag, status)**
- Anywhere you use MPI_Send or MPI_Recv, you can use the pair of MPI_Isend/MPI_Wait or MPI_Irecv/MPI_Wait
- Note the MPI types:

MPI_Status status; // type used with the status output from recv

MPI_Request request; // the type of the handle used with isend/ircv

Non-blocking operations are extremely important … they allow you to overlap computation and communication.

Non-Blocking Send-Receive Diagram



send side

receive side

T0: **MPI_Irecv**

T1: MPI_Irecv Returns

buffer unavailable to user

MPI_Isend T2

MPI_Isend returns T3

buffer unavailable to user

T4: **MPI_Wait** called

MPI_Wait T5

Sender completes T6

T7: transfer finishes

T8: **MPI_Wait** returns

MPI_Wait returns T9

buffer available to user

receive buffer filled and available to the user

Exercise: Ring program

- Start with the basic ring program we provide. Run it for a range of message sizes and notes what happens for large messages.
  - It may deadlock if the network stalls due to there being no place to put a message (i.e. no receives in place so the send blocking on when its buffer can be reused hangs).
- Try to make it more stable for large messages by:
  - Split-phase … have the nodes "send than receive" while the other half "receive then send".
  - Sendrecv … a collective communication send/receive.
  - Isend/Irecv … nonblocking send receive

```
double *buff;     int buff_count, to, from, tag=3;   MPI_Status stat;

MPI_Recv (buff, buff_count, MPI_DOUBLE, from, tag, MPI_COMM_WORLD, &stat);
MPI_Send (buff, buff_count, MPI_DOUBLE, to,     tag,  MPI_COMM_WORLD);
MPI_Isend( Buff, count, datatype, dest, tag, comm, request )
MPI_Irecv( Buff, count, datatype, src, tag, comm, request )
MPI_Wait( request, status )
MPI_Sendrecv (snd_buff,  buff_count, MPI_DOUBLE, to, tag,
                 rcv_buf,     buff_count, MPI_DOUBLE, to, tag, MPI_COMM_WORLD, &stat);
```

Example: shift messages around a ring (part 1 of 2)

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

int main(int argc, char **argv)
{
  int num, rank, size, tag, next, from;
  MPI_Status status1, status2;
  MPI_Request req1, req2;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank( MPI_COMM_WORLD, &rank);
  MPI_Comm_size( MPI_COMM_WORLD, &size);
  tag = 201;
  next = (rank+1) % size;
  from = (rank + size - 1) % size;
  if (rank == 0) {
    printf("Enter the number of times around the ring: ");
    scanf("%d", &num);

    printf("Process %d sending %d to %d\n", rank, num, next);
    MPI_Isend(&num, 1, MPI_INT, next, tag,
                                         MPI_COMM_WORLD,&req1);
    MPI_Wait(&req1, &status1);
  }
  do {
    MPI_Irecv(&num, 1, MPI_INT, from, tag,
                                         MPI_COMM_WORLD, &req2);
    MPI_Wait(&req2, &status2);

    if (rank == 0) {
      num--;
      printf("Process 0 decremented number\n");
    }

    printf("Process %d sending %d to %d\n", rank, num, next);
    MPI_Isend(&num, 1, MPI_INT, next, tag,
                                          MPI_COMM_WORLD, &req1);
    MPI_Wait(&req1, &status1);
  } while (num != 0);

  if (rank == 0) {
    MPI_Irecv(&num, 1, MPI_INT, from, tag,
                                          MPI_COMM_WORLD, &req2);
    MPI_Wait(&req2, &status2);
  }
  MPI_Finalize();
  return 0;
}
```

Outline

- MPI and distributed memory systems
- The Bulk Synchronous Pattern and MPI collective operations
- Introduction to message passing
- The diversity of message passing in MPI
- ➡ Geometric Decomposition and MPI
- Concluding Comments

Example: finite difference methods

- Solve the heat diffusion equation in 1 D:
  - u(x,t) describes the temperature field
  - We set the heat diffusion constant to one
  - Boundary conditions, constant u at endpoints.

$$\frac{\partial^2 u}{\partial x^2} = \frac{\partial u}{\partial t}$$

- map onto a mesh with stepsize h and k

$$x_i = x_0 + ih \qquad t_i = t_0 + ik$$

- Central difference approximation for spatial derivative (at fixed time)

$$\frac{\partial^2 u}{\partial x^2} = \frac{u_{j+1} - 2u_j + u_{j-1}}{h^2}$$

- Time derivative at t = t^{n+1}

$$\frac{du}{dt} = \frac{u^{n+1} - u^n}{k}$$

Example: Explicit finite differences

- Combining time derivative expression using spatial derivative at t = t^n

$$\frac{u_j^{n+1} - u_j^n}{k} = \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{h^2}$$

  - Solve for u at time n+1 and step j

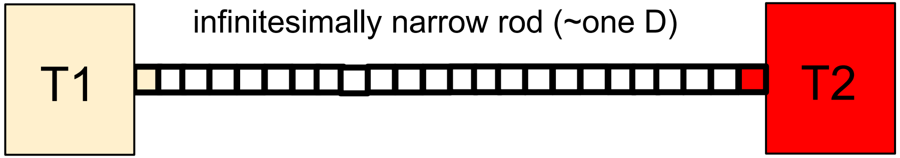
$$u_j^{n+1} = (1-2r)u_j^n + ru_{j-1}^n + ru_{j+1}^n \qquad r = {}^{k}\!/_{h^2}$$

  - The solution at t = t_{n+1} is determined explicitly from the solution at t = t_n (assume u[t][0] = u[t][N] = Constant for all t).

```
for (int t = 0; t < N_STEPS-1; ++t)
   for (int x = 1; x < N-1; ++x)
         u[t+1][x] = u[t][x] + r*(u[t][x+1] - 2*u[t][x] + u[t][x-1]);
```

  - Explicit methods are easy to compute … each point updated based on nearest neighbors. Converges for r<1/2.

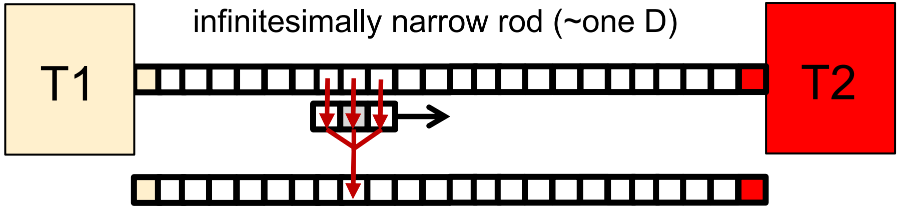
Heat Diffusion equation



“infinite” heat bath (fixed temperature, T1)

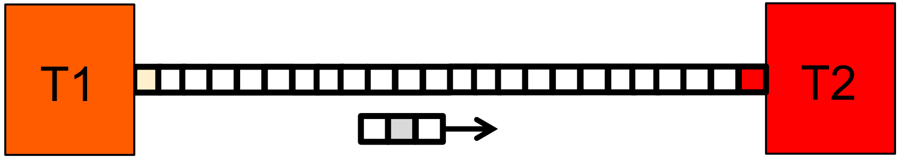
“infinite” heat bath (fixed temperature, T2)

Heat Diffusion equation



Pictorially, you are sliding a three point "stencil" across the domain (u[t]) and computing a new value of the center point (u[t+1]) at each stop.

Heat Diffusion equation

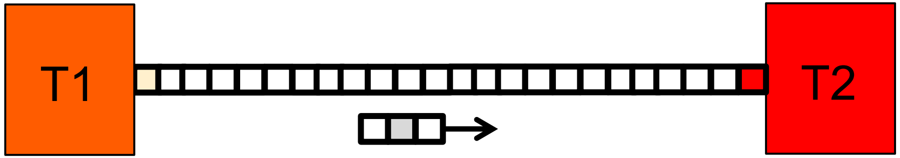


```
int main()
{
   double *u   = malloc (sizeof(double) * (N));
   double *up1 = malloc (sizeof(double) * (N));

   initialize_data(uk, ukp1, N, P); // init to zero, set end temperatures
   for (int t = 0; t < N_STEPS; ++t){
      for (int x = 1; x < N-1; ++x)
          up1[x] = u[x] + (k / (h*h)) * (u[x+1] - 2*u[x] + u[x-1]);

      temp = up1; up1 = u; u = temp;
    }
return 0;
```

Note: I don't need the intermediate "u[t]" values hence "u" is just indexed by x.

A well known trick with 2 arrays so I don't overwrite values from step k-1 as I fill in for step k

Heat Diffusion equation

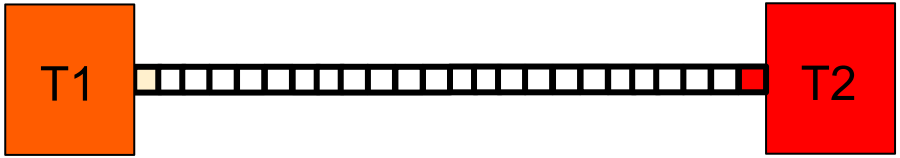


```
int main()
{
   double *u   = malloc (sizeof(double) * (N));
   double *up1 = malloc (sizeof(double) * (N));

   initialize_data(uk, ukp1, N, P); // init to zero, set end temperatures
   for (int t = 0; t < N_STEPS; ++t){
      for (int x = 1; x < N-1; ++x)
          up1[x] = u[x] + (k / (h*h)) * (u[x+1] - 2*u[x] + u[x-1]);

      temp = up1; up1 = u; u = temp;
    }
return 0;
```

How would you parallelize this program?

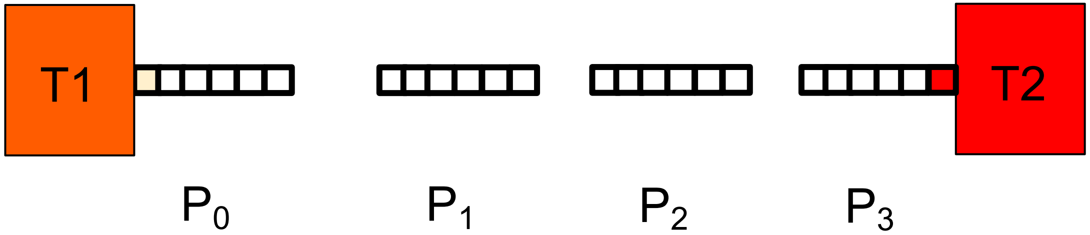
Heat Diffusion equation

- Start with our original picture of the problem … a one dimensional domain with end points set at a fixed temperature.



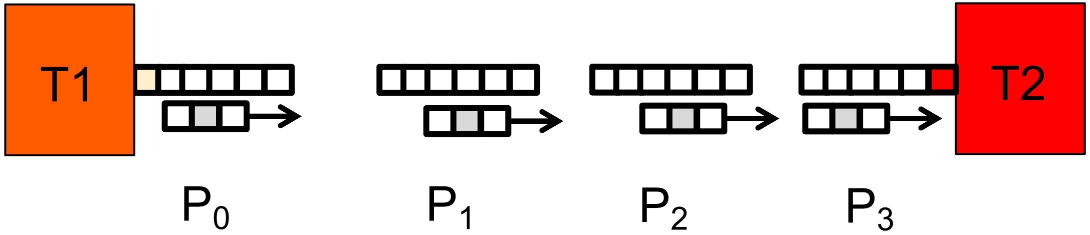
Heat Diffusion equation

- Break it into chunks assigning one chunk to each process.



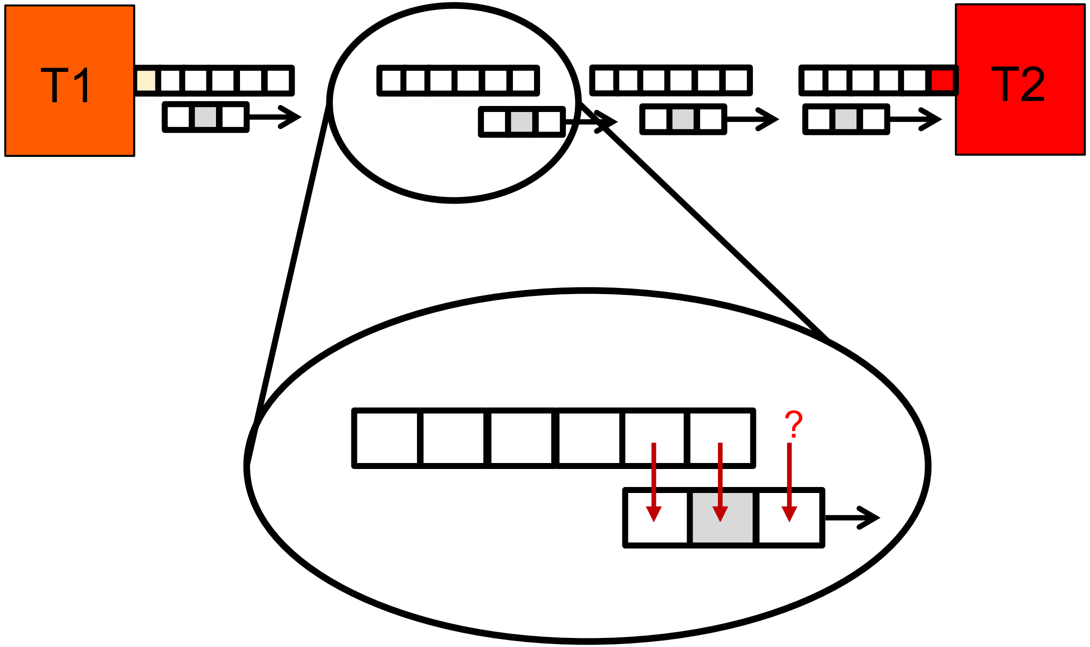
Heat Diffusion equation

- Each process works on it’s own chunk … sliding the stencil across the domain to updates its own data.



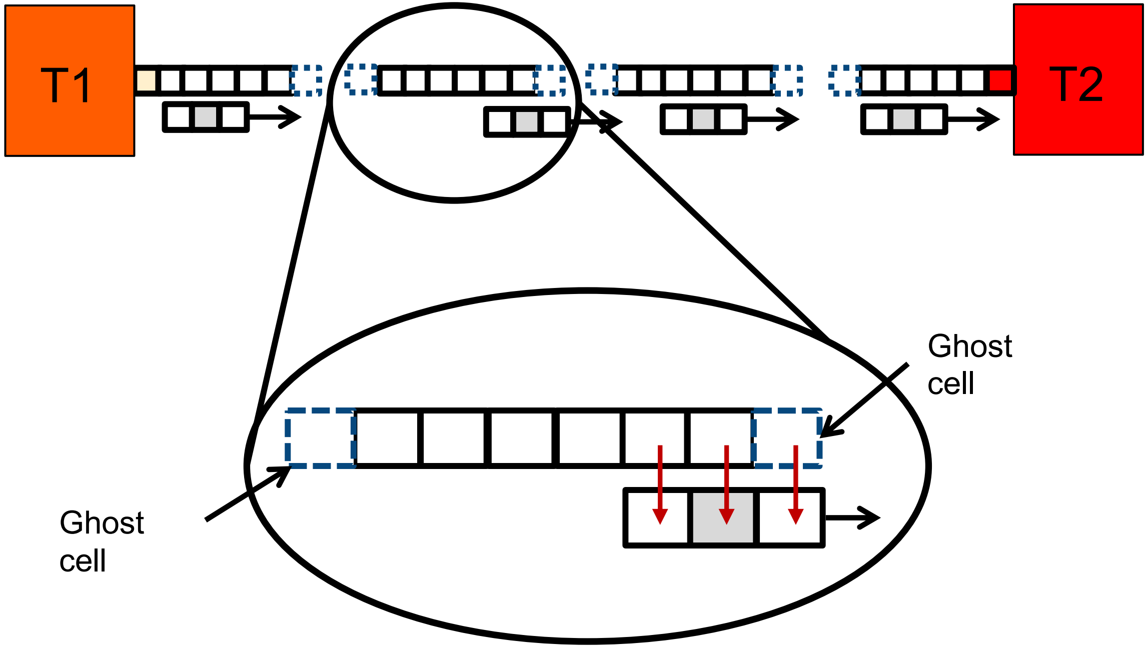
Heat Diffusion equation

- What about the ends of each chunk … where the stencil will run off the end and hence have missing values for the computation?



Heat Diffusion equation

- We add ghost cells to the ends of each chunk, update them with the required values from neighbor chunks at each time step … hence giving the stencil everything it needs on any given chunk to update all of its values.



Heat Diffusion MPI Example

```
MPI_Init (&argc, &argv);
MPI_Comm_size (MPI_COMM_WORLD, &P);
MPI_Comm_rank (MPI_COMM_WORLD, &myID);
double *u   = malloc (sizeof(double) * (2 + N/P))  // include "Ghost Cells"
double *up1 = malloc (sizeof(double) * (2 + N/P)); // to hold values
                                                    // from my neighbors
initialize_data(uk, ukp1, N, P);
for (int t = 0; t < N_STEPS; ++t){
  if (myID != 0)  MPI_Send (&u[1], 1, MPI_DOUBLE, myID-1, 0, MPI_COMM_WORLD);
  if (myID != P-1) MPI_Recv (&u[N/P+1], 1, MPI_DOUBLE, myID+1, 0, MPI_COMM_WORLD, &status);
  if (myID != P-1) MPI_Send (&u[N/P], 1, MPI_DOUBLE, myID+1, 0, MPI_COMM_WORLD);
  if (myID != 0)   MPI_Recv (&u[0], 1, MPI_DOUBLE, myID-1, 0,MPI_COMM_WORLD, &status);

  for (int x = 2; x < N/P; ++x)
    up1[x] = u[x] + (k / (h*h)) * (u[x+1] - 2*u[x] + u[x-1]);
  if (myID != 0)
    up1[1] = u[1] + (k / (h*h)) * (u[1+1] - 2*u[1] + u[1-1]);
  if (myID != P-1)
    up1[N/P] = u[N/P] + (k/(h*h)) * (u[N/P+1] - 2*u[N/P] + u[N/P-1]);
  temp = up1; up1 = u; u = temp;

} // End of for (int t ...) loop

MPI_Finalize();
return 0;
```

We write/explain this part first and then address the communication and data structures

Heat Diffusion MPI Example

```
// Compute interior of each “chunk”
  for (int x = 2; x < N/P; ++x)
    up1[x] = u[x] + (k / (h*h)) * (u[x+1] - 2*u[x] + u[x-1]);

// update edges of each chunk keeping the two far ends fixed
// (first element on Process 0 and the last element on process P-1).
  if (myID != 0)
    up1[1] = u[1] + (k / (h*h)) * (u[1+1] - 2*u[1] + u[1-1]);

  if (myID != P-1)
    up1[N/P] = u[N/P] + (k/(h*h)) * (u[N/P+1] - 2*u[N/P] + u[N/P-1]);

// Swap pointers to prepare for next iterations
  temp = up1; up1 = u; u = temp;

} // End of for (int t ...) loop

MPI_Finalize();
return 0;
```

Update array values using local data and values from ghost cells.

u[0] and u[N/P+1] are the ghost cells

Note I was lazy and assumed N was evenly divided by P. Clearly, I’d never do this in a “real” program.

Heat Diffusion MPI Example

1D PDE solver … the simplest “real” message passing code I can think of. Note: edges of domain held at a fixed temperature

```
MPI_Init (&argc, &argv);
MPI_Comm_size (MPI_COMM_WORLD, &P);
MPI_Comm_rank (MPI_COMM_WORLD, &myID);
double *u   = malloc (sizeof(double) * (2 + N/P))  // include "Ghost Cells"
double *up1 = malloc (sizeof(double) * (2 + N/P)); // to hold values
                                                   // from my neighbors
initialize_data(uk, ukp1, N, P);
for (int t = 0; t < N_STEPS; ++t){
  if (myID != 0)
    MPI_Send (&u[1], 1, MPI_DOUBLE, myID-1, 0, MPI_COMM_WORLD);
```

Send my “left” boundary value to the neighbor on my “left’

```
  if (myID != P-1)
    MPI_Recv (&u[N/P+1], 1, MPI_DOUBLE, myID+1, 0, MPI_COMM_WORLD, &status);
```

Receive my “right” ghost cell from the neighbor to my “right’

```
  if (myID != P-1)
    MPI_Send (&u[N/P], 1, MPI_DOUBLE, myID+1, 0, MPI_COMM_WORLD);
```

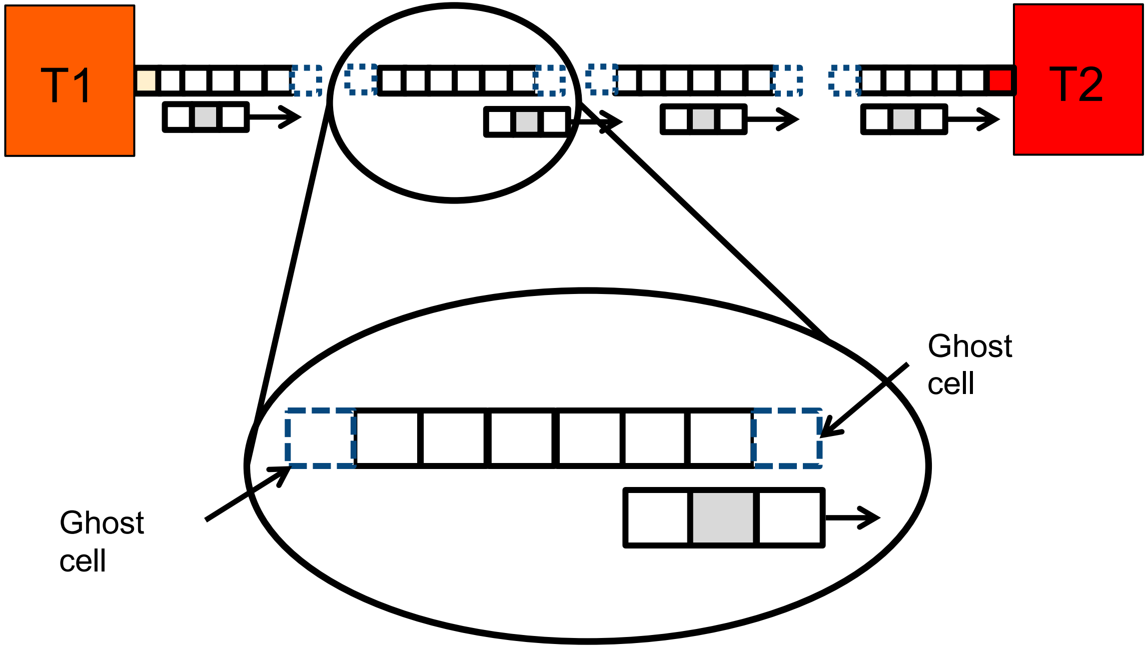
Send my “right” boundary value to the neighbor to my “right’

```
  if (myID != 0)
    MPI_Recv (&u[0], 1, MPI_DOUBLE, myID-1, 0,MPI_COMM_WORLD, &status);
/* continued on previous slide */
```

Receive my “left” ghost cell from the neighbor to my “left”

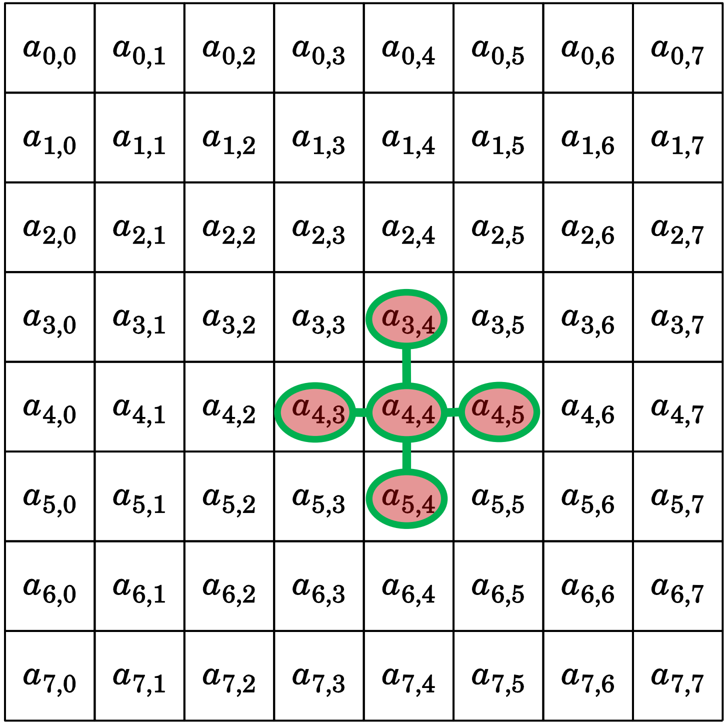
The Geometric Decomposition Pattern

- This is an instance of a very important design pattern … the Geometric decomposition pattern.



Partitioned Arrays

- Realistic problems are 2D or 3D; require more complex data distributions.
- We need to parallelize the computation by partitioning this index space
- Example: Consider a 2D domain over which we wish to solve a PDE using an explicit finite difference solver . The figure shows a five point stencil … update a value based on its value and its 4 neighbors.
- Start with an array →



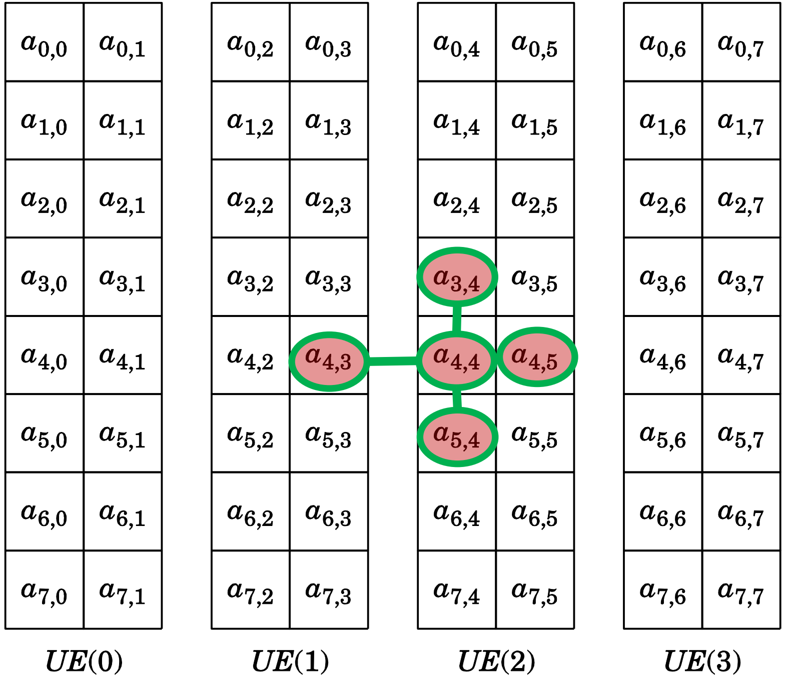
$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$
$a_{4,0}$	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	$a_{4,6}$	$a_{4,7}$
$a_{5,0}$	$a_{5,1}$	$a_{5,2}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$	$a_{5,6}$	$a_{5,7}$
$a_{6,0}$	$a_{6,1}$	$a_{6,2}$	$a_{6,3}$	$a_{6,4}$	$a_{6,5}$	$a_{6,6}$	$a_{6,7}$
$a_{7,0}$	$a_{7,1}$	$a_{7,2}$	$a_{7,3}$	$a_{7,4}$	$a_{7,5}$	$a_{7,6}$	$a_{7,7}$

Partitioned Arrays: Column block distribution

- Split the non-unit-stride dimension (P-1) times to produce P chunks, assign the ith chunk to P$_{i.}$ …. To keep things simple, assume N%P = 0
- In a 2D finite-differencing program (exchange edges), how much do we have to communicate? **O(N/P) messages** per processor

P is the # of processors

N is the order of our square matrix



$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$
$a_{4,0}$	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$	$a_{4,4}$	$a_{4,5}$	$a_{4,6}$	$a_{4,7}$
$a_{5,0}$	$a_{5,1}$	$a_{5,2}$	$a_{5,3}$	$a_{5,4}$	$a_{5,5}$	$a_{5,6}$	$a_{5,7}$
$a_{6,0}$	$a_{6,1}$	$a_{6,2}$	$a_{6,3}$	$a_{6,4}$	$a_{6,5}$	$a_{6,6}$	$a_{6,7}$
$a_{7,0}$	$a_{7,1}$	$a_{7,2}$	$a_{7,3}$	$a_{7,4}$	$a_{7,5}$	$a_{7,6}$	$a_{7,7}$
$UE(0)$		$UE(1)$		$UE(2)$		$UE(3)$	

UE = unit of execution … think of it as a generic term for “process or thread”

Partitioned Arrays: Block distribution

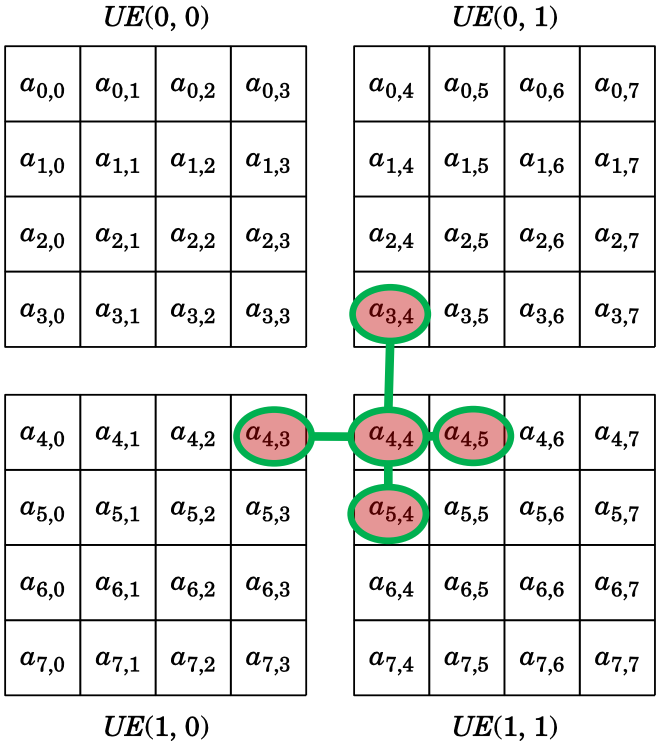
- If we parallelize in both dimensions, then we have $(N/p)^2$ elements per processor, and we need to send **~4*(n/p) messages** from each processor. Asymptotically better than 2*sqrt(N).

P is the # of processors

Assume a p by p square mesh … p=sqrt(P)

N is the order of our square matrix

Dimension of each block is N/P



$UE(0, 0)$

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

$UE(0, 1)$

$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
$a_{3,4}$	$a_{3,5}$	$a_{3,6}$	$a_{3,7}$

$UE(1, 0)$

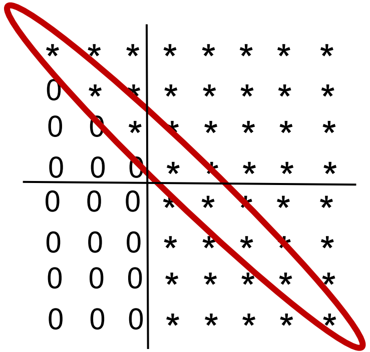
$a_{4,0}$	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$
$a_{5,0}$	$a_{5,1}$	$a_{5,2}$	$a_{5,3}$
$a_{6,0}$	$a_{6,1}$	$a_{6,2}$	$a_{6,3}$
$a_{7,0}$	$a_{7,1}$	$a_{7,2}$	$a_{7,3}$

$UE(1, 1)$

$a_{4,4}$	$a_{4,5}$	$a_{4,6}$	$a_{4,7}$
$a_{5,4}$	$a_{5,5}$	$a_{5,6}$	$a_{5,7}$
$a_{6,4}$	$a_{6,5}$	$a_{6,6}$	$a_{6,7}$
$a_{7,4}$	$a_{7,5}$	$a_{7,6}$	$a_{7,7}$

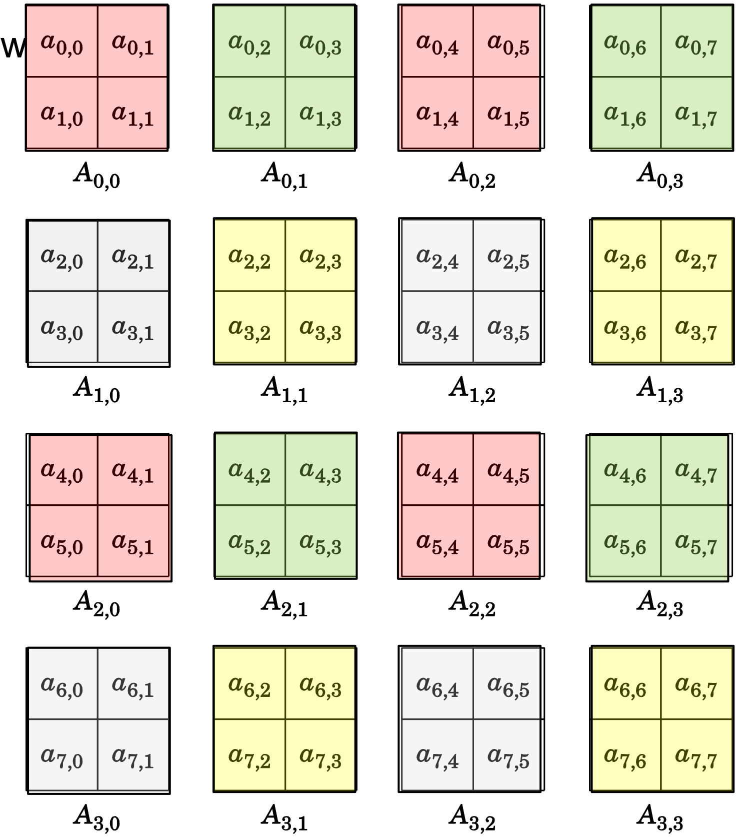
Partitioned Arrays: block cyclic distribution

- LU decomposition (A= LU) .. Move down the diagonal transform rows to “zero the column” below the diagonal.



$$\left[\begin{array}{ccc|ccccc}
* & * & * & * & * & * & * & * \\
0 & * & * & * & * & * & * & * \\
0 & 0 & * & * & * & * & * & * \\
0 & 0 & 0 & * & * & * & * & * \\
\hline
0 & 0 & 0 & * & * & * & * & * \\
0 & 0 & 0 & * & * & * & * & * \\
0 & 0 & 0 & * & * & * & * & * \\
0 & 0 & 0 & * & * & * & * & *
\end{array}\right]$$

  - Zeros fill in the right lower triangle of the matrix … less work to do.
  - Balance load with cyclic distribution of blocks of A mapped onto a grid of nodes (2x2 in this case … colors show the mapping to nodes).

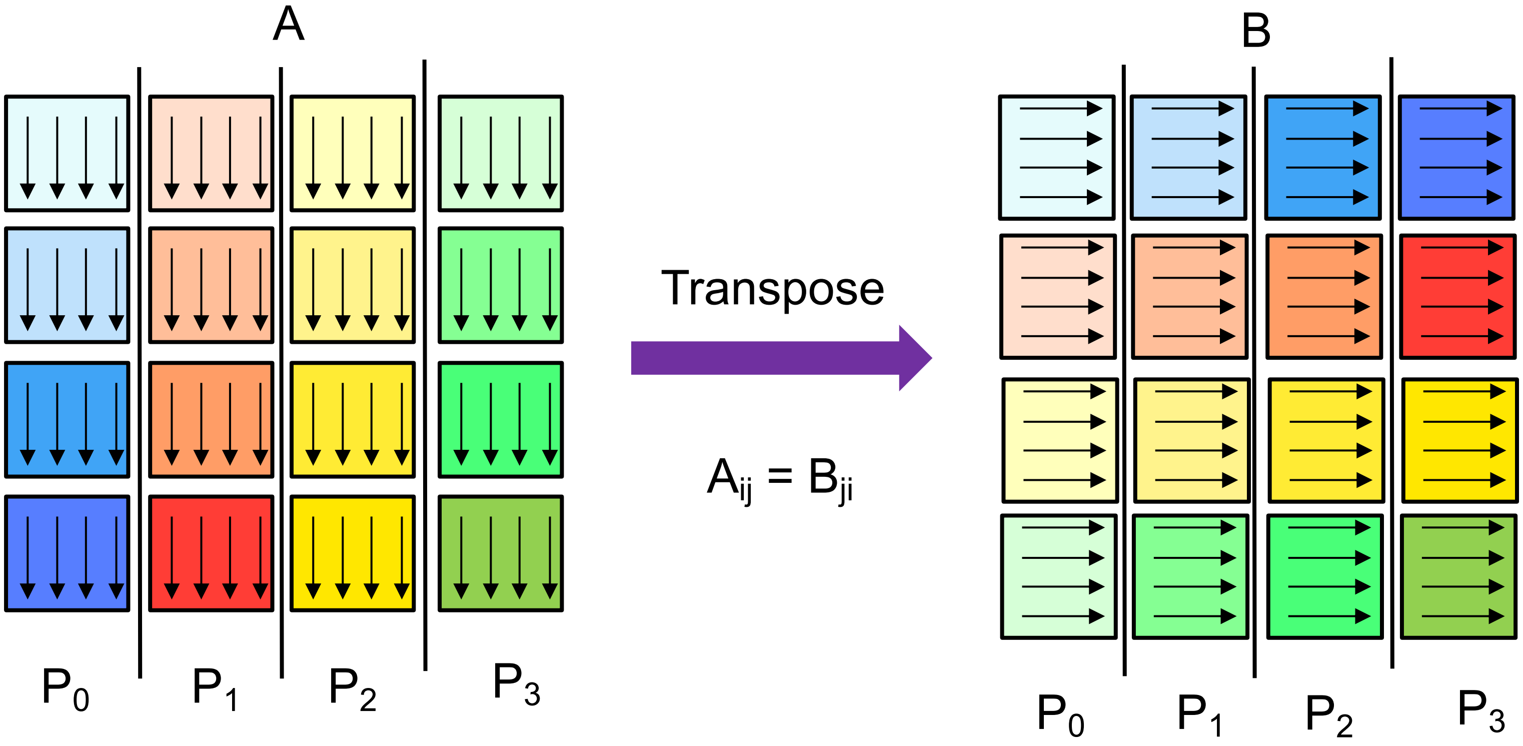


$a_{0,0}$ $a_{0,1}$ $a_{1,0}$ $a_{1,1}$ $A_{0,0}$	$a_{0,2}$ $a_{0,3}$ $a_{1,2}$ $a_{1,3}$ $A_{0,1}$	$a_{0,4}$ $a_{0,5}$ $a_{1,4}$ $a_{1,5}$ $A_{0,2}$	$a_{0,6}$ $a_{0,7}$ $a_{1,6}$ $a_{1,7}$ $A_{0,3}$
$a_{2,0}$ $a_{2,1}$ $a_{3,0}$ $a_{3,1}$ $A_{1,0}$	$a_{2,2}$ $a_{2,3}$ $a_{3,2}$ $a_{3,3}$ $A_{1,1}$	$a_{2,4}$ $a_{2,5}$ $a_{3,4}$ $a_{3,5}$ $A_{1,2}$	$a_{2,6}$ $a_{2,7}$ $a_{3,6}$ $a_{3,7}$ $A_{1,3}$
$a_{4,0}$ $a_{4,1}$ $a_{5,0}$ $a_{5,1}$ $A_{2,0}$	$a_{4,2}$ $a_{4,3}$ $a_{5,2}$ $a_{5,3}$ $A_{2,1}$	$a_{4,4}$ $a_{4,5}$ $a_{5,4}$ $a_{5,5}$ $A_{2,2}$	$a_{4,6}$ $a_{4,7}$ $a_{5,6}$ $a_{5,7}$ $A_{2,3}$
$a_{6,0}$ $a_{6,1}$ $a_{7,0}$ $a_{7,1}$ $A_{3,0}$	$a_{6,2}$ $a_{6,3}$ $a_{7,2}$ $a_{7,3}$ $A_{3,1}$	$a_{6,4}$ $a_{6,5}$ $a_{7,4}$ $a_{7,5}$ $A_{3,2}$	$a_{6,6}$ $a_{6,7}$ $a_{7,6}$ $a_{7,7}$ $A_{3,3}$

🟥 🟩
🟪 🟨

Matrix Transpose:
Column block decomposition

You can only learn this stuff by doing it so we're going to design an algorithm to transpose a matrix using a partitioned array model based on column blocks.



Let's keep things simple. N = blk*P where blk is the order of the square subblocks

Exercise/discussion

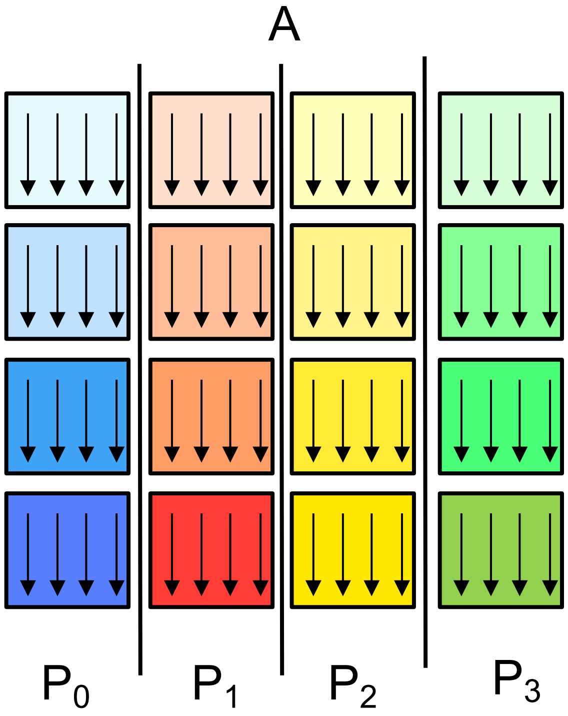
We are going to create a transpose program that uses the SPMD pattern.

That's Single Program Multiple Data.

We'll run the same program on each node.

What is the high level structure of this algorithm?

That is … who will each Processor march through its set of blocks?



Let's keep things simple. N = blk*P where blk is the order of the square subblocks

Exercise/discussion

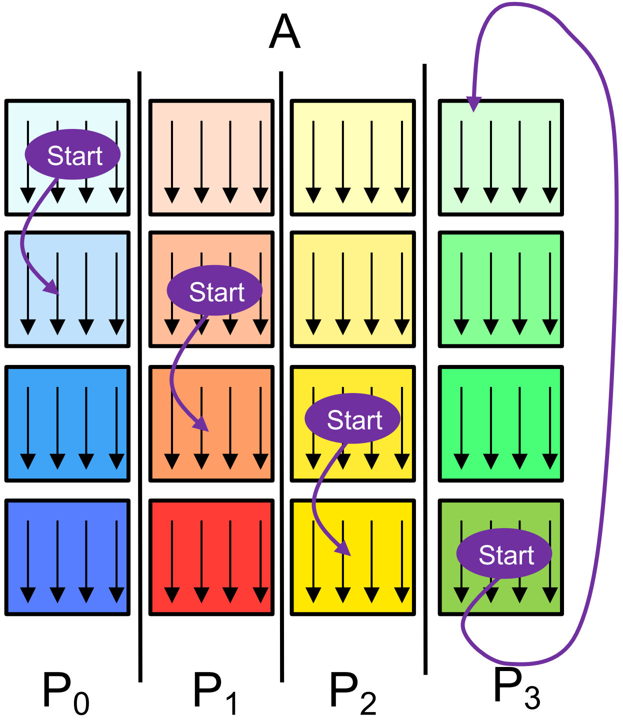
We are going to create a transpose program that uses the SPMD pattern.

That's Single Program Multiple Data.

We'll run the same program on each node.

What is the high level structure of this algorithm?

That is … How will each Processor march through its set of blocks?



There is no one way to do this.

Since its an SPMD program, you want a symmetric path through the blocks on each processor.

A great approach is for everyone to start from their diagonal and shift down (with a circular shift pattern … i.e. when you run off an edge, wrap around to the opposite edge.

Phase 0 … transpose your diagonal
Phase 1 … deal with next block "down"

How many phases for our case? **3**

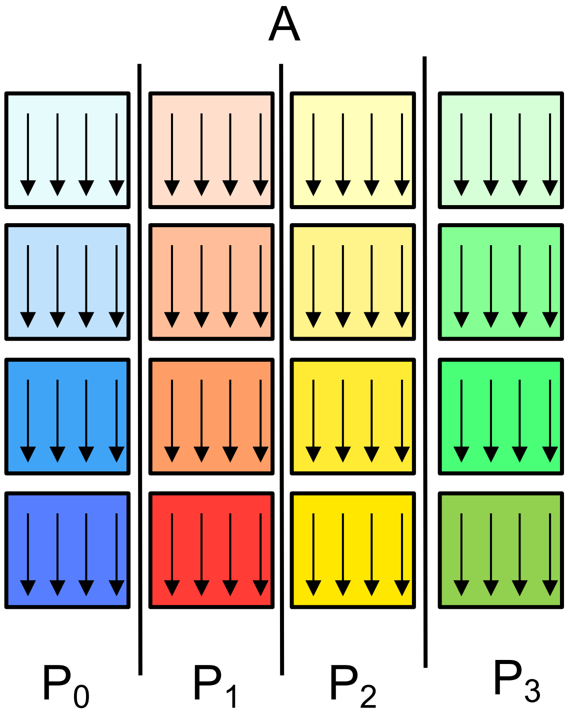
Let's keep things simple. N = blk*P where blk is the order of the square subblocks

Exercise/discussion

What is the communication pattern for each phase?

Which block is sent?
Who receives that block?
Where do they put it?

Remember, this is SPMD. You have a single program so how will you structure it so each processor does the right thing for each block.



Let’s keep things simple. N = blk*P where blk is the order of the square subblocks

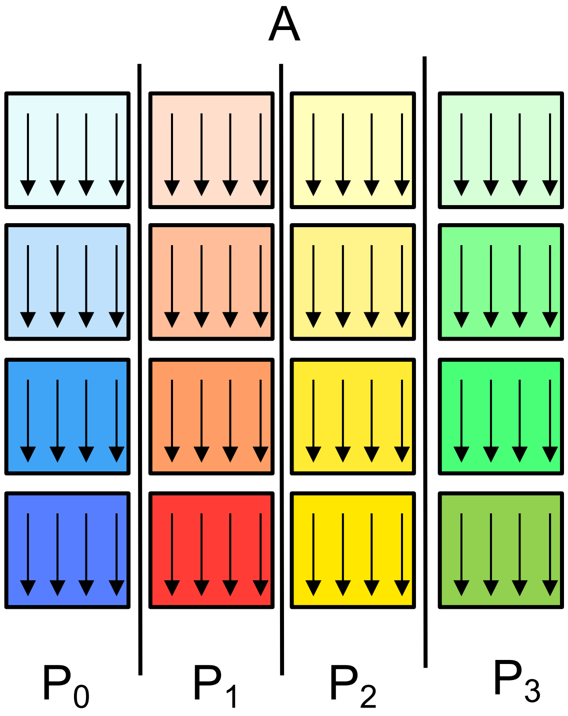
Exercise/discussion

The three phases shown for P_1

What is the communication pattern for each phase?

Which block is sent?
Who receives that block?
Where do they put it?

Remember, this is SPMD. You have a single program so how will you structure it so each processor does the right thing for each block.



Communication pattern

Phase 0 … no communication … just a local transpose on block ID (the diagonal)

Phase k … Send block (ID+k) to your k^{th} neighbor

On the receiving end, the block **from** neighbor ID goes to your row-block number ID. Why?

Let’s keep things simple. N = blk*P where blk is the order of the square subblocks

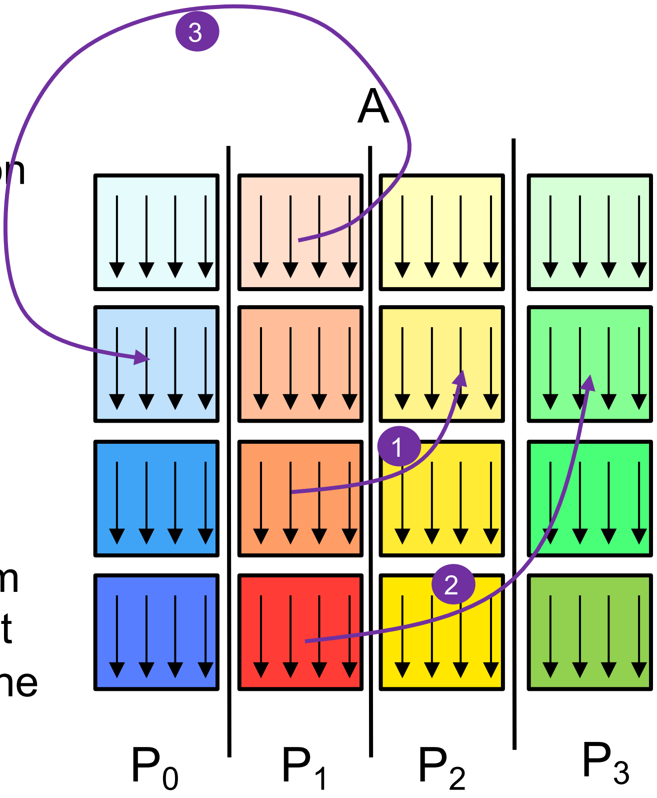
Exercise/discussion

What is the communication pattern for each phase?

Which block is sent?
Who receives that block?
Where do they put it?

Remember, this is SPMD. You have a single program so how will you structure it so each processor does the right thing for each block.

The three phases shown for P_1



Communication pattern

Phase 0 … no communication … just a local transpose on block ID (the diagonal)

Phase k … Send block (ID+k) to your k^th neighbor

On the receiving end, the block **from** neighbor ID goes to your row-block number ID. Why?

We have a column block decomposition so column block indices are the rank (ID). Plus in a transpose you map column blocks to row blocks

Remember to transpose the block … either before you send it or after it arrives.

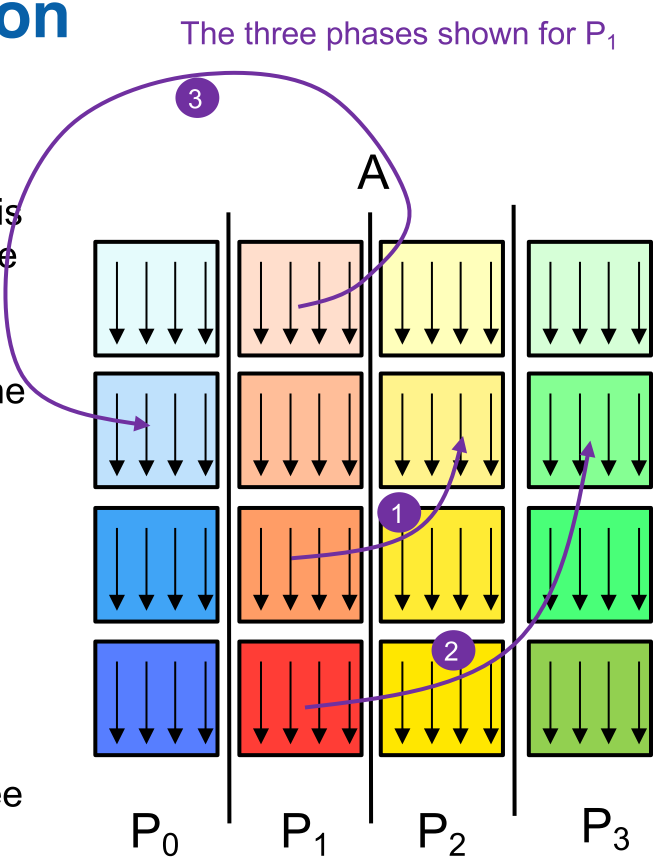
Let's keep things simple. N = blk*P where blk is the order of the square subblocks

Exercise/discussion

Now for the tricky part. This is an SPMD pattern. Every node will run the same program.

So using just the rank (ID), the phase, and the number of processes (P) … write expressions for the communication patterns for each phase and each processor.

Hint: you have to account for wrap-around (e.g. phase three in the figure)



Let's keep things simple. N = blk*P where blk is the order of the square subblocks

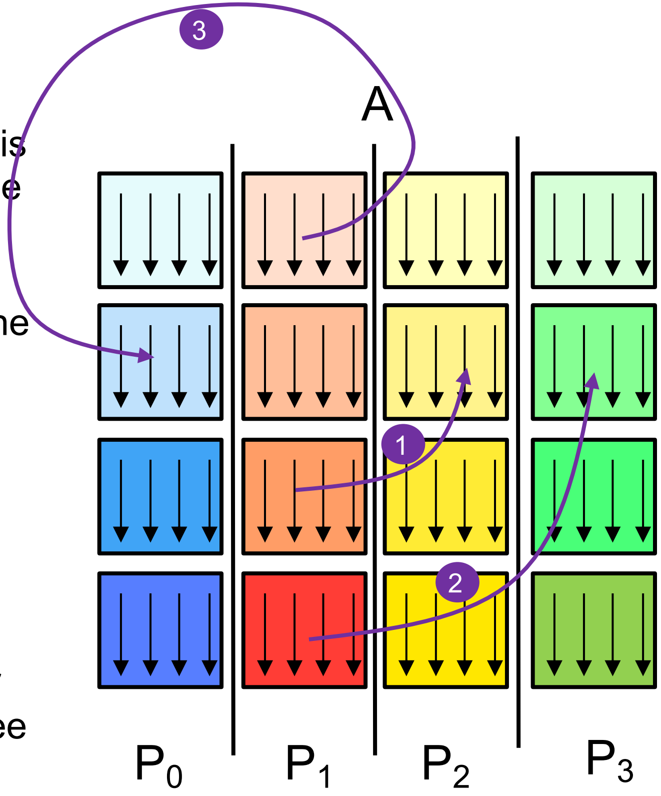
Exercise/discussion

Now for the tricky part. This is an SPMD pattern. Every node will run the same program.

So using just the rank (ID), the phase, and the number of processes (P) … write expressions for the communication patterns for each phase and each processor.

Hint: you have to account for wrap-around (e.g. phase three in the figure)

The three phases shown for P_1



In a given phase, each process will need to send a block TO another process and receive a block FROM another process.

You need expressions for TO and FROM.

We will put this in a C macro.

A macro in the C programming language replaces code in the program text BEFORE compilation.

Example: (note: this is NOT the right answer … I don’t want to make this too easy)

```
#define TO(ID, Phase) (ID/PHASE)%N
```

Let’s keep things simple. N = blk*P where blk is the order of the square subblocks

Exercise: transpose program

- Start with the basic transpose program we provide.
- Go to trans_sendrcv.c and enter your definitions for the TO and FROM macros.
- Test and verify correctness
- Try different message passing approaches.
- Can you overlap the local transpose and the communication between nodes?

```
double *buff;     int buff_count, to, from, tag=3;   MPI_Status stat;

MPI_Recv (buff, buff_count, MPI_DOUBLE, from, tag, MPI_COMM_WORLD, &stat);
MPI_Send (buff, buff_count, MPI_DOUBLE, to,     tag,  MPI_COMM_WORLD);
MPI_Isend( Buff, count, datatype, dest, tag, comm, request )
MPI_Irecv( Buff, count, datatype, src, tag, comm, request )
MPI_Wait( request, status )
MPI_Sendrecv (snd_buff,  buff_count, MPI_DOUBLE, to, tag,
              rcv_buf,     buff_count, MPI_DOUBLE, to, tag, MPI_COMM_WORLD, &stat);
```

Outline

- MPI and distributed memory systems
- The Bulk Synchronous Pattern and MPI collective operations
- Introduction to message passing
- The diversity of message passing in MPI
- Geometric Decomposition and MPI
- Concluding Comments

The 12 core functions in MPI

- MPI_Init
- MPI_Finish
- MPI_Comm_size
- MPI_Comm_rank
- MPI_Send
- MPI_Recv
- MPI_Reduce
- MPI_Isend
- MPI_Irecv
- MPI_Wait
- MPI_Wtime
- MPI_Bcast

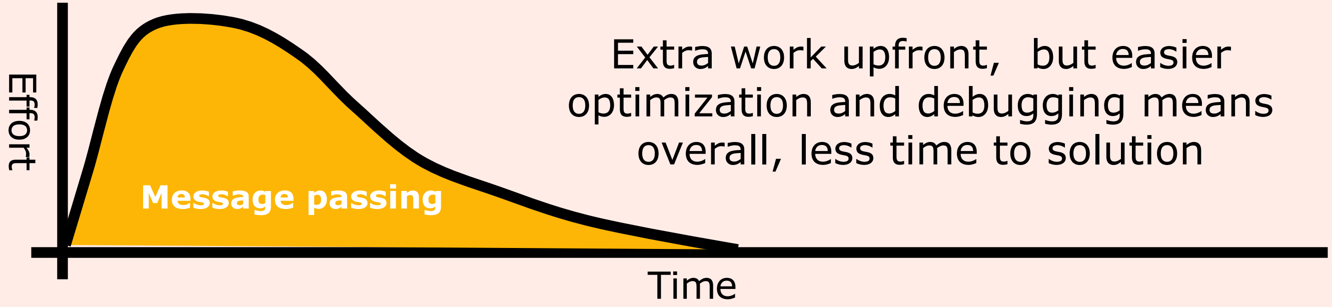
10

The ~~12~~ core functions in MPI

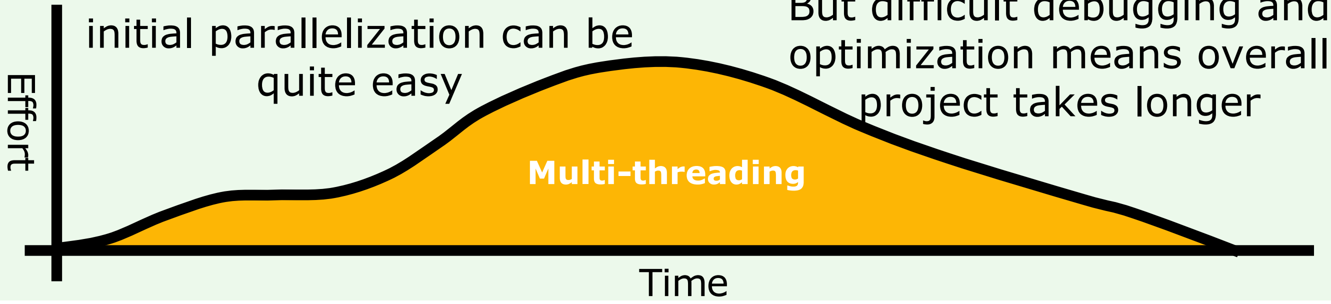
- MPI_Init
- MPI_Finish
- MPI_Comm_size
- MPI_Comm_rank
- ~~MPI_Send~~
- ~~MPI_Recv~~
- MPI_Reduce
- MPI_Isend
- MPI_Irecv
- MPI_Wait
- MPI_Wtime
- MPI_Bcast

Real Programmers always try to overlap communication and computation .. Post your receives using MPI_Irecv() then where appropriate, MPI_Isend().

Does a shared address space make programming easier?



Extra work upfront, but easier optimization and debugging means overall, less time to solution



initial parallelization can be quite easy

But difficult debugging and optimization means overall project takes longer

Proving that a shared address space program using semaphores is race free is an NP-complete problem*

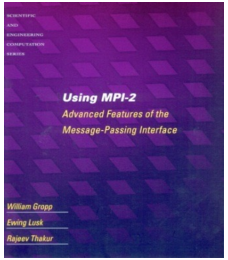
*P. N. Klein, H. Lu, and R. H. B. Netzer, Detecting Race Conditions in Parallel Programs that Use Semaphores, Algorithmica, vol. 35 pp. 321–345,

MPI References

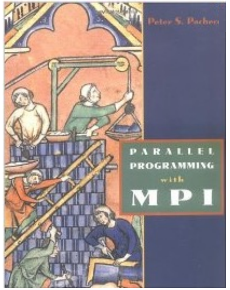
- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Other information on Web:
  - at http://www.mcs.anl.gov/mpi
  - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages

Books for learning MPI

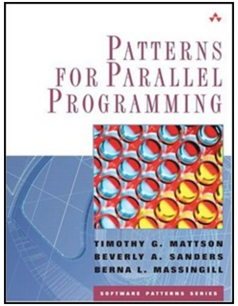
- *Using MPI-2: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Thakur, MIT Press, 1999..


Scientific and Engineering Computation Series
Using MPI-2
Advanced Features of the Message-Passing Interface
William Gropp
Ewing Lusk
Rajeev Thakur

- *Parallel Programming with MPI*, by Peter Pacheco, Morgan-Kaufmann, 1997.



- *Patterns for Parallel Programing*, by Tim Mattson, Beverly Sanders, and Berna Massingill.



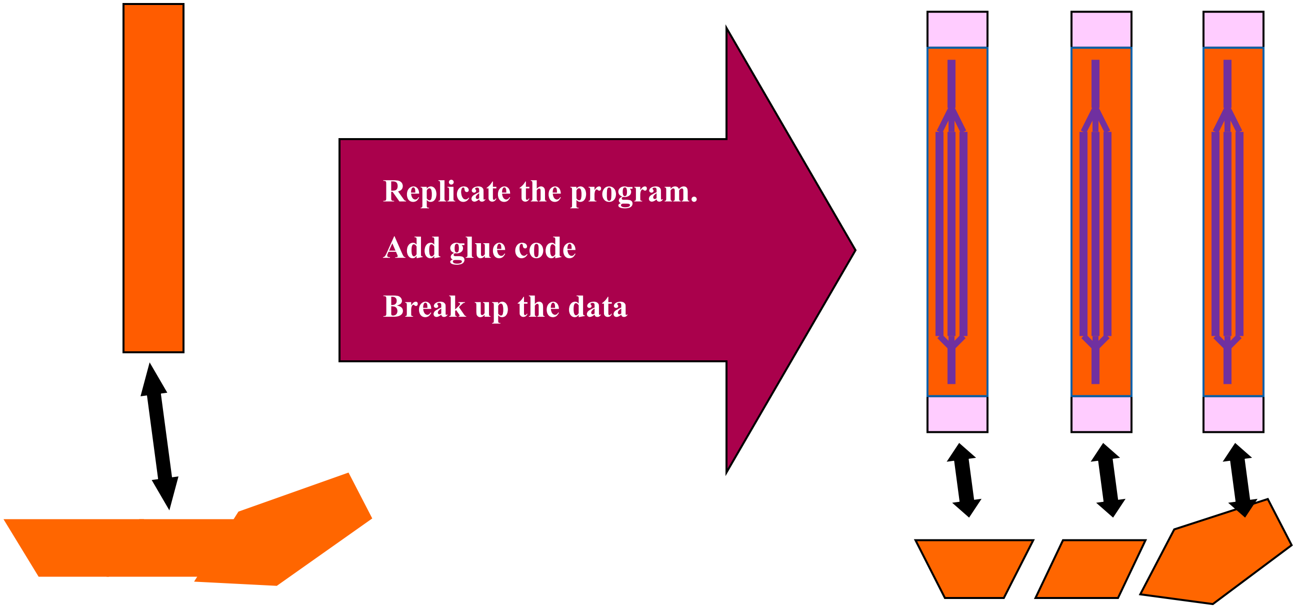
Backup

- Mixing OpenMP and MPI
- Loading MPI on your system

How do people mix MPI and OpenMP?

A sequential program
working on a data set

- Create the MPI program with its data decomposition.
- Use OpenMP inside each MPI process.



Replicate the program.

Add glue code

Break up the data

Pi program with MPI and OpenMP

Get the MPI part done first, then add OpenMP pragma where it makes sense to do so

```
#include <mpi.h>
#include “omp.h”
void main (int argc, char *argv[])
{
        int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
        MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
        my_steps = num_steps/numprocs ;
#pragma omp parallel for reduction(+:sum) private(x)
        for (i=my_id*my_steps; i<(m_id+1)*my_steps ; i++)
        {
                x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
        sum *= step ;
        MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
        MPI_COMM_WORLD) ;
}
```

Key issues when mixing OpenMP and MPI

1. Messages are sent to a process not to a particular thread.
   - Not all MPIs are threadsafe. MPI 2.0 defines threading modes:
     - MPI_Thread_Single: no support for multiple threads
     - MPI_Thread_Funneled: Mult threads, only master calls MPI
     - MPI_Thread_Serialized: Mult threads each calling MPI, but they do it one at a time.
     - MPI_Thread_Multiple: Multiple threads without any restrictions
   - Request and test thread modes with the function:

     MPI_init_thread(desired_mode, delivered_mode, ierr)
2. Environment variables are not propagated by mpirun. You'll need to broadcast OpenMP parameters and set them with the library routines.

Dangerous Mixing of MPI and OpenMP

- The following will work only if MPI_Thread_Multiple is supported … a level of support I wouldn't depend on.

```
MPI_Comm_Rank(MPI_COMM_WORLD, &mpi_id) ;
#pragma omp parallel
{
     int tag, swap_neigh, stat, omp_id = omp_thread_num();
     long buffer [BUFF_SIZE], incoming [BUFF_SIZE];
     big_ugly_calc1(omp_id, mpi_id, buffer);
                                                                              // Finds MPI id and tag so
     neighbor(omp_id, mpi_id, &swap_neigh, &tag);  // messages don't conflict

     MPI_Send (buffer,   BUFF_SIZE, MPI_LONG, swap_neigh,
                    tag, MPI_COMM_WORLD);
     MPI_Recv (incoming, buffer_count, MPI_LONG, swap_neigh,
                    tag,  MPI_COMM_WORLD, &stat);

     big_ugly_calc2(omp_id, mpi_id, incoming, buffer);
#pragma critical
    consume(buffer, omp_id, mpi_id);
}
```

Messages and threads

- Keep message passing and threaded sections of your program separate:
  - Setup message passing outside OpenMP parallel regions (MPI_Thread_funneled)
  - Surround with appropriate directives (e.g. critical section or master) (MPI_Thread_Serialized)
  - For certain applications depending on how it is designed it may not matter which thread handles a message. (MPI_Thread_Multiple)
    - Beware of race conditions though if two threads are probing on the same message and then racing to receive it.

Safe Mixing of MPI and OpenMP

Put MPI in sequential regions

```
MPI_Init(&argc, &argv) ;     MPI_Comm_Rank(MPI_COMM_WORLD, &mpi_id) ;

// a whole bunch of initializations

#pragma omp parallel for
for (I=0;I<N;I++) {
    U[I] =  big_calc(I);
}

    MPI_Send (U,   BUFF_SIZE, MPI_DOUBLE, swap_neigh,
                tag, MPI_COMM_WORLD);
    MPI_Recv (incoming, buffer_count, MPI_DOUBLE, swap_neigh,
                tag,  MPI_COMM_WORLD, &stat);

#pragma omp parallel for
for (I=0;I<N;I++) {
    U[I] =  other_big_calc(I, incoming);
}

consume(U, mpi_id);
```

Technically Requires MPI_Thread_funneled, but I have never had a problem with this approach … even with pre-MPI-2.0 libraries.

Safe Mixing of MPI and OpenMP

Protect MPI calls inside a parallel region

```
MPI_Init(&argc, &argv) ;     MPI_Comm_Rank(MPI_COMM_WORLD, &mpi_id) ;

// a whole bunch of initializations

#pragma omp parallel
{
#pragma omp for
    for (I=0;I<N;I++)    U[I] =  big_calc(I);

#pragma master
{
     MPI_Send (U,   BUFF_SIZE, MPI_DOUBLE, neigh, tag,  MPI_COMM_WORLD);
     MPI_Recv (incoming, count, MPI_DOUBLE, neigh,  tag,  MPI_COMM_WORLD,  &stat);
}
#pragma omp barrier
#pragma omp for
    for (I=0;I<N;I++)   U[I] =  other_big_calc(I, incoming);

#pragma omp master
    consume(U, mpi_id);
}
```

Technically Requires MPI_Thread_funneled, but I have never had a problem with this approach … even with pre-MPI-2.0 libraries.

Hybrid OpenMP/MPI works, but is it worth it?

- Literature* is mixed on the hybrid model: sometimes its better, sometimes MPI alone is best.
- There is potential for benefit to the hybrid model
  - MPI algorithms often require replicated data making them less memory efficient.
  - Fewer total MPI communicating agents means fewer messages and less overhead from message conflicts.
  - Algorithms with good cache efficiency should benefit from shared caches of multi-threaded programs.
  - The model maps perfectly with clusters of SMP nodes.
- But really, it’s a case by case basis and to large extent depends on the particular application.

*L. Adhianto and Chapman, 2007

Backup

- Mixing OpenMP and MPI
- Loading MPI on your system

MPIch library on Apple Laptops: MacPorts

- To use MPI on your Apple laptop:
  - Download Xcode. Be sure to choose the command line tools that match your OS.
  - Install MacPorts (if you haven't already … use the installer for your OS from macports.org).

```
sudo port selfupdate
```

Update to latest version of MacPorts

```
sudo port install mpich-gcc9
```

Graph the library that matches the version of your gcc compiler.

Test the installation with a simple program

```
mpicc hello.c
mpiexec –n 4 ./a.out
```