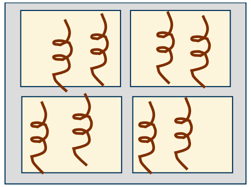
Parallel Programming … across nodes*

Tim Mattson

*Node: Large scale HPC systems are made from networked computers. A computer at a location in the network is called a **node**.

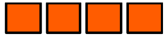
Hardware is diverse … and its only getting worse!!!

Write code with OpenMP

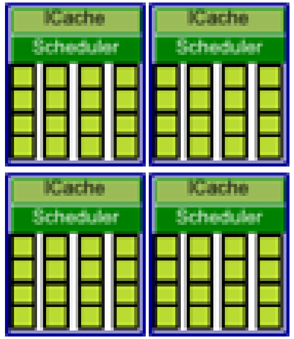


CPU

Work with the compiler to vectorize code



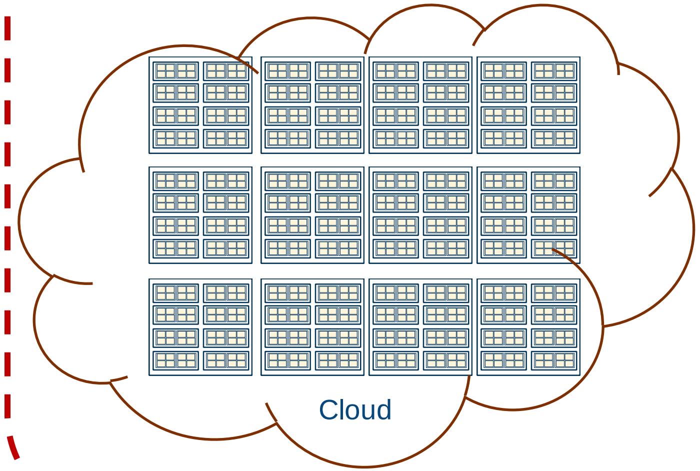
SIMD/Vector



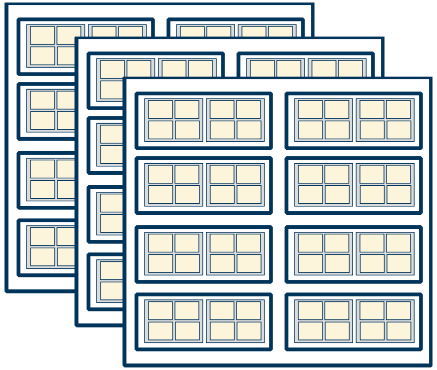
GPU

Use a portable API (OpenMP) but if you must, use CUDA. It's all the same model

Parallelism over disjoint address-spaces …. MPI

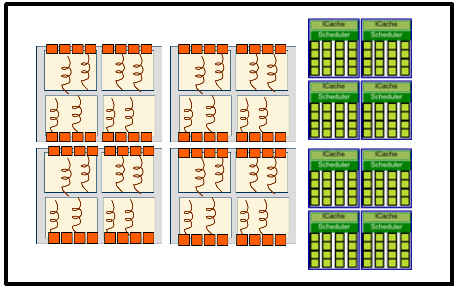


Cloud



Cluster

OpenMP lets you "do it all".



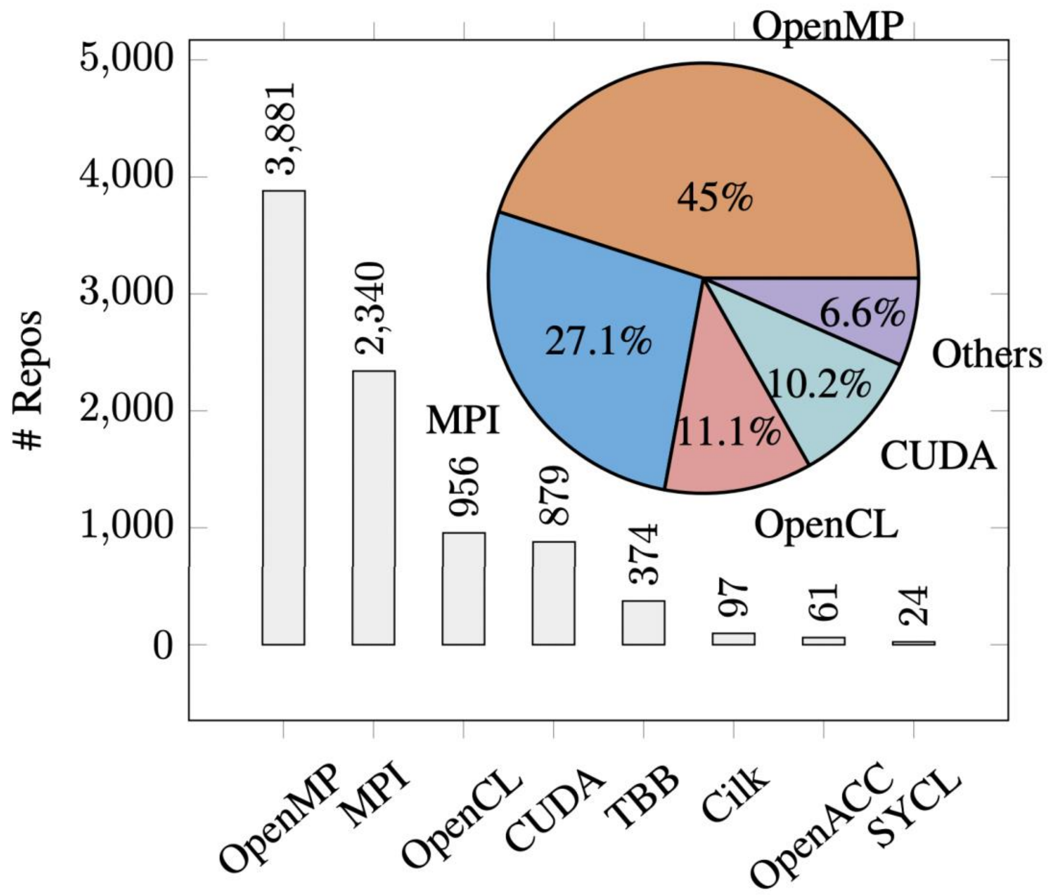
Heterogeneous node

Just learn three programming languages and you cover all of HPC

- 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.

OpenMP is the most popular parallel programing model in use today

In a dataset (HPCorpus) of all C/C++/Fortan github repositories from 2013-2023, OpenMP was found to be the most popular parallel programming model

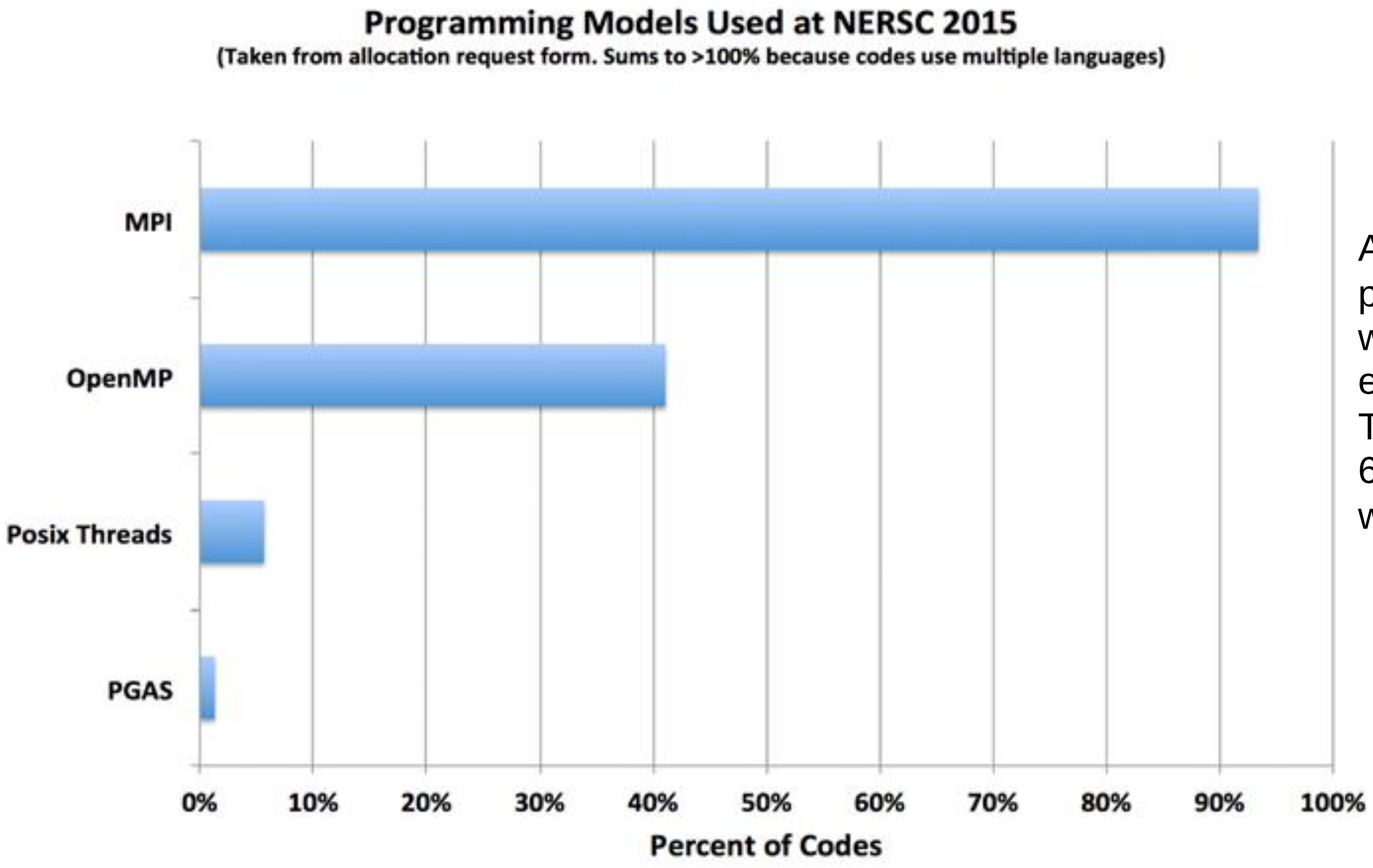


Aggregate numbers over all repositories from 2013 to 2023

Quantifying OpenMP: Statistical insights into usage and adoption, Tal Kadosh, et al., HPEC'2023, https://arxiv.org/abs/2308.08002

Note: since we did not collect files with .cu or .cuf suffices, we undercounted CUDA usage in HPCorpus.

For codes running at Supercomputer Centers, MPI is #1



At the time of this study, the production NERSC machines were large systems that emphasized the CPU:
The study was based on 6900 users, over 850 projects working with over 600 codes.

Source: https://www.nersc.gov/assets/Uploads/HelenHe-OpenMPCon-2015.pdf

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)

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

We’ll cover this case on Friday

- 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 Human Learning Group. tgmattso@gmail.com

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

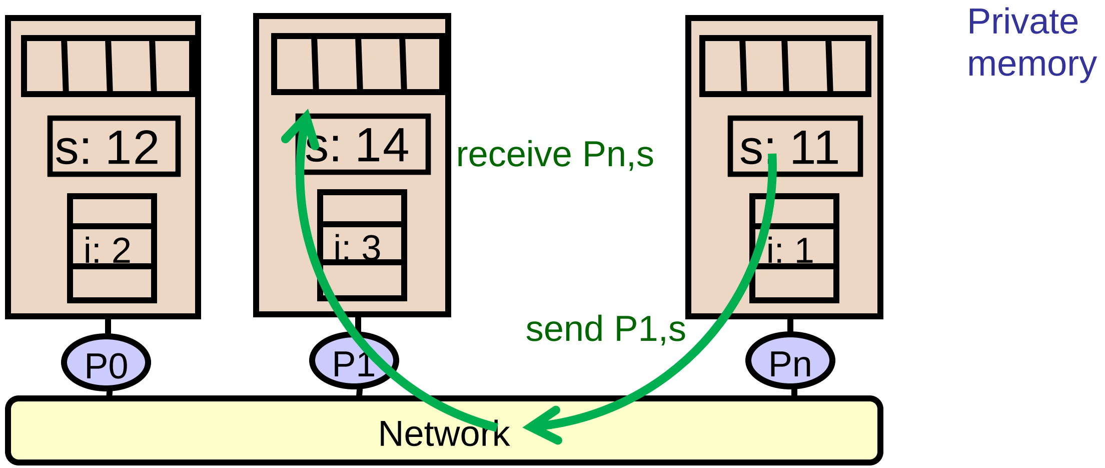
Outline

- ➡ 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

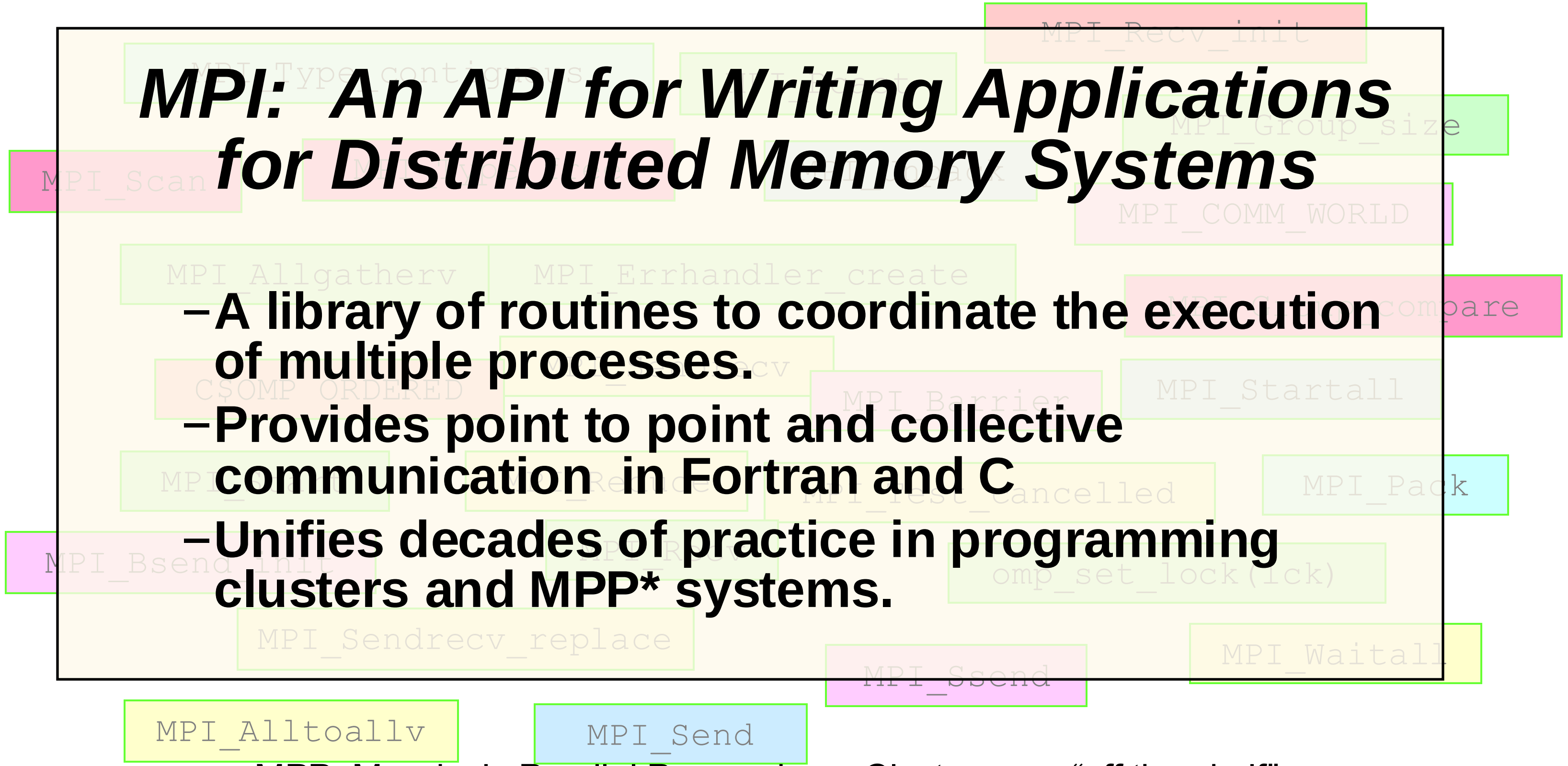
Programming Model for distributed memory systems

- Programs execute as a collection of processes.
  - **Number of processes almost always fixed at program startup time**
  - **Local address space per node -- NO physically shared memory.**
  - **Logically shared data is partitioned over local processes.**

- Processes communicate by explicit send/receive pairs
  - **Synchronization is implicit by communication events.**
  - **MPI (Message Passing Interface) is the most commonly used API**



Parallel API's: MPI, the Message Passing Interface



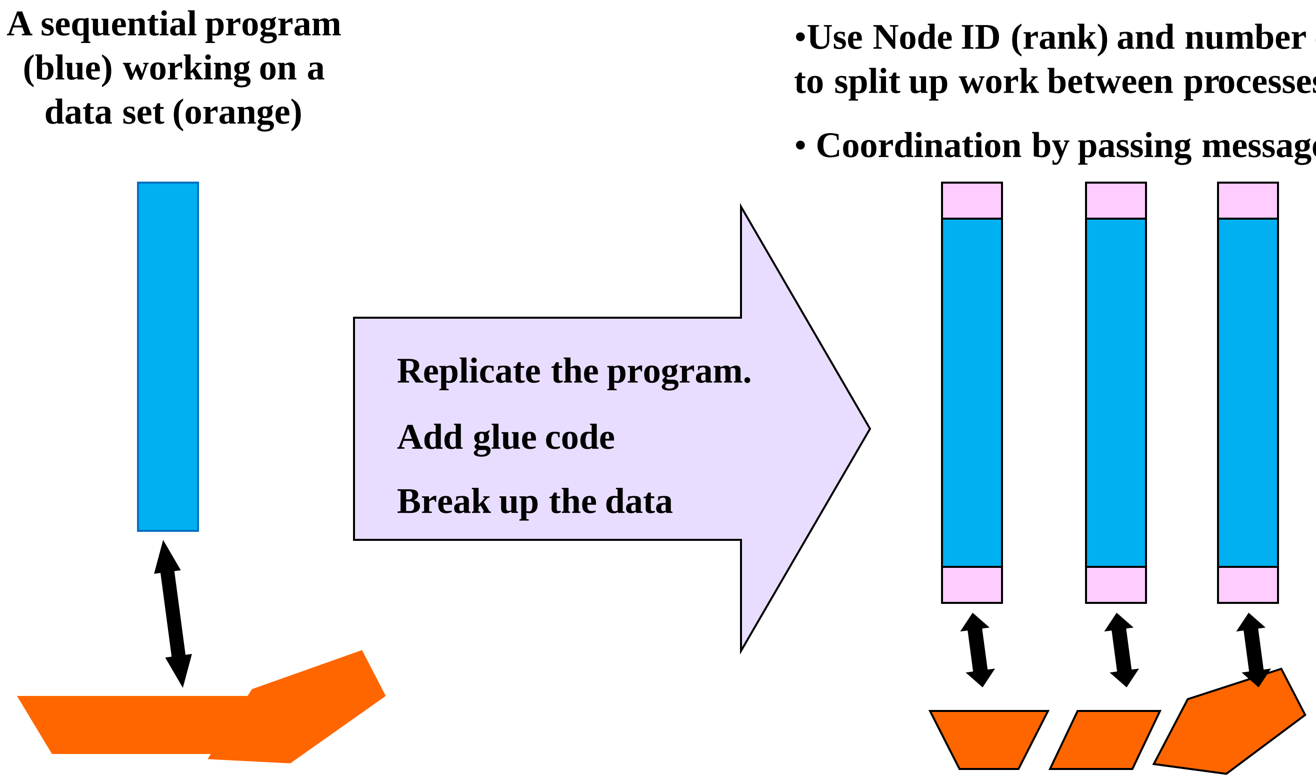
MPI: An API for Writing Applications for Distributed Memory Systems

- **A library of routines to coordinate the execution of multiple processes.**
- **Provides point to point and collective communication in Fortran and C**
- **Unifies decades of practice in programming clusters and MPP* systems.**

*MPP: Massively Parallel Processing. Clusters use "off the shelf" components. MPP systems include custom system integration.

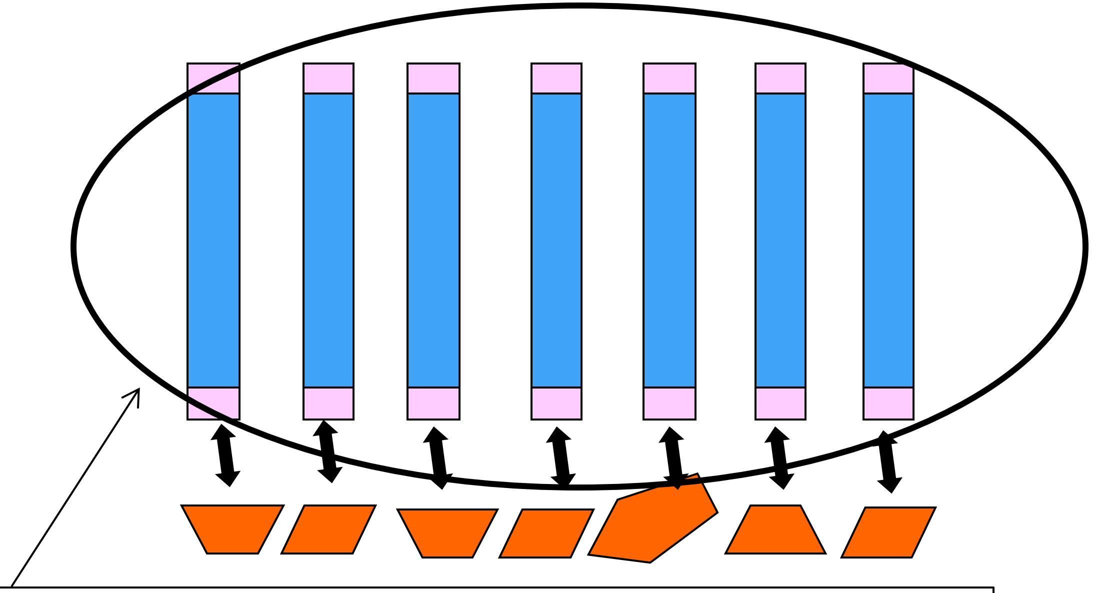
How do people use MPI? The SPMD Design Pattern

- A replicated single program working on a decomposed data set.
- Use Node ID (rank) and number of nodes to split up work between processes (ranks)
- Coordination by passing messages.



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**”

MPI Hello World Program

```c
#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)
```

- **`MPI_Comm`**, an *opaque data type called a communicator.* Default context: MPI_COMM_WORLD (all processes)
- **`MPI_Comm_size`** returns the number of processes in the process group associated with the communicator

```
#inclu
#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;
}
```

Communicators consist of two parts, a **context** and a **process group**.

The communicator lets one control how groups of messages interact.

Communicators support modular SW … i.e. I can give a library module its own communicator and know that it's messages can't collide with messages originating from outside the module

Which process "am I" (the rank)

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

- **`MPI_Comm`**, an *opaque data type,* **a** communicator. Default context: MPI_COMM_WORLD (all processes)
- **`MPI_Comm_rank`** An integer ranging from 0 to "(num of procs)-1"

```
#inclu
#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;
}
```

Note that other than init() and finalize(), every MPI function has a communicator.

This makes sense .. You need a context and group of processes that the MPI functions impact … and those come from the communicator.

Exercise: Hello world

- Goal
  - To setup your account in CloudVeneto and confirm that you can compiler and run an MPI.
- Program
  - Compile and run the mpi_hello.c program on the nodes of our cluster

```
$ ssh LoginName@gate.cloudveneto.it
```
ssh to the cloudveneto gateway with your account

```
$ ssh student##@10.67.19.47
```
ssh to our cluster … use the number (##) you were given

```
$ cd cloud/student##
```
Go to the home directory for your account

```
$ cp -r ../common/* .
```
Copy the shared MPI directory into your own account.

```
$ mpicc mpi_hello.c
```
Compile the hello world program we provide.

```
$ mpirun -hostfile exampleHost –n 2 a.out
```
Run the program with the provided hostfile

Running the program

```
#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;
}
```

- On a 4 node cluster, to run this program (hello):

```
> mpirun –np 4 –hostfile hostf 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
```

- Where “hostf” is a file with the names of the cluster nodes and the number of slots on the node, one to a line.

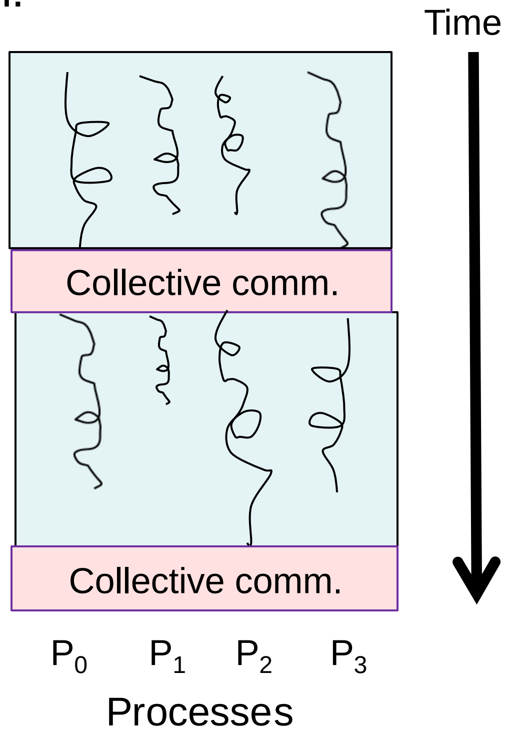
Outline

- 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

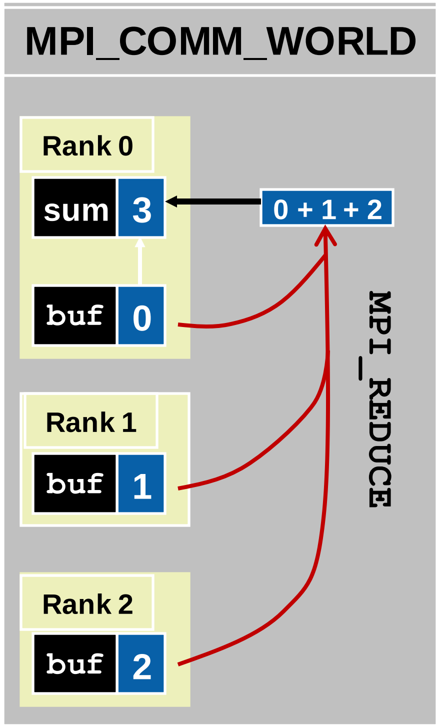
*This is a subset of available MPI types

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

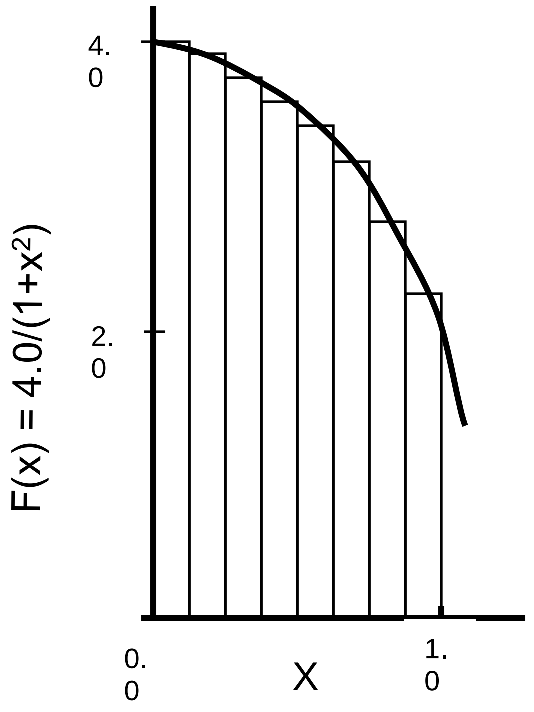
Operation	Function
`MPI_BAND`	Bitwise AND
`MPI_LOR`	Logical OR
`MPI_BOR`	Bitwise OR
`MPI_LXOR`	Logical exclusive OR
`MPI_BXOR`	Bitwise exclusive OR
User-defined	It is possible to define new reduction operations

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;
  buf = 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;
}
```

Timing MPI programs

- MPI added a function (which OpenMP copied) to time programs.
- **MPI_Wtime()** returns a double for the time (in seconds) for some arbitrary time in the past.
- As with omp_get_wtime(), call before and after a section of code of interest to get an elapsed time.

```
double init_time = MPI_Wtime();

// do a bunch of stuff

double elapsed_time = MPI_Wtime() - init_time;
```

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.

```
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);
Double MPI_Wtime();
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) ;
        double init_time = MPI_Wtime();
        my_steps = num_steps/numprocs ;
        for (i=id; i<num_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);
        if(my_id == 0) printf(" runtime = %lf\n",MPI_Wtime()-init_time);
        MPI_Finalize();
}
```

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

MPI Pi program performance (on my laptop)

```
#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) ;
        double init_time = MPI_Wtime();
        my_steps = num_steps/numprocs ;
        for (i=id; i<num_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);
        if(my_id == 0) printf(" runtime = %lf\n",MPI_Wtime()-init_time);
        MPI_Finalize();
}
```

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

*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.

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

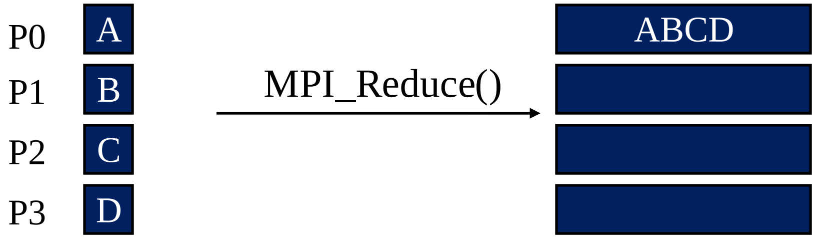
Operation	Function
`MPI_BAND`	Bitwise AND
`MPI_LOR`	Logical OR
`MPI_BOR`	Bitwise OR
`MPI_LXOR`	Logical exclusive OR
`MPI_BXOR`	Bitwise exclusive OR
User-defined	It is possible to define new reduction operations

Many operations beyond sum

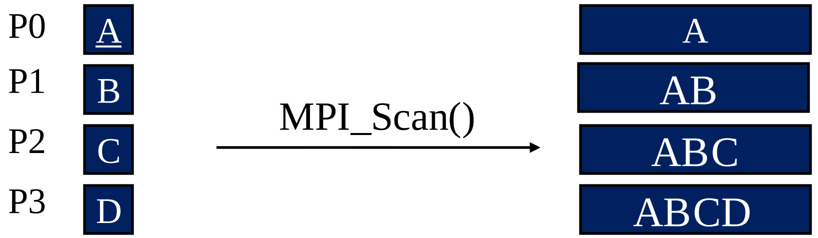
MPI defines a rich set of Collective operations

Collective Computations

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



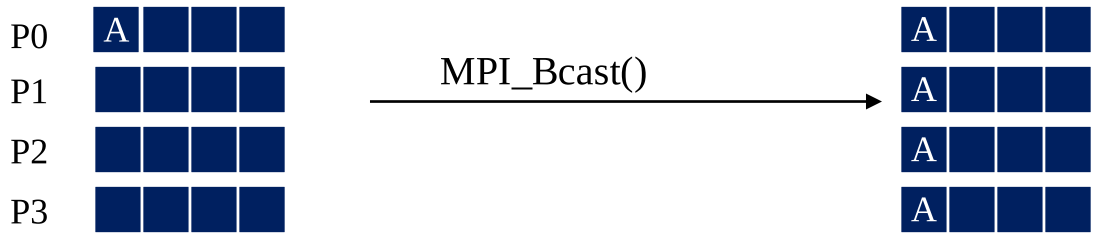
Scan: Take values on each P and combine them with a scan operation and spread the scan array out among all P.



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

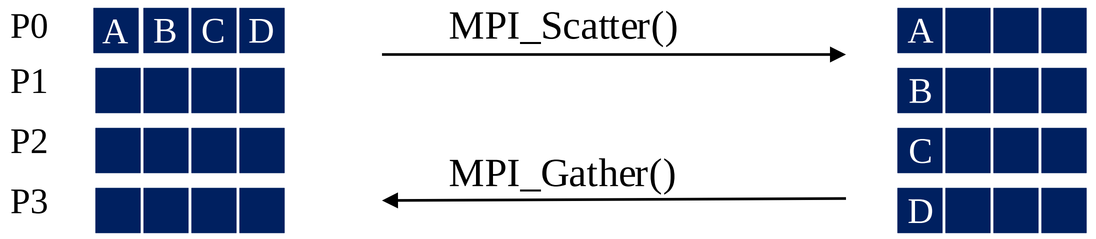
Collective Data Movement

Broadcast a value from P0 (the root) and give a copy to P1, P2 and P3



Scatter an array on P0 (the root) to P1, P2, and P3

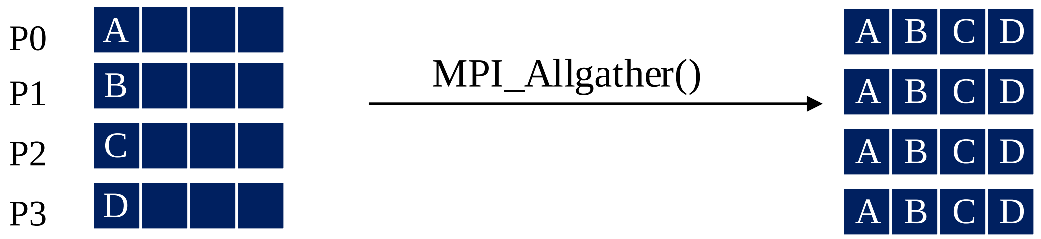
Gather values from P1, P2, and P3 into an array on P0 (the root)



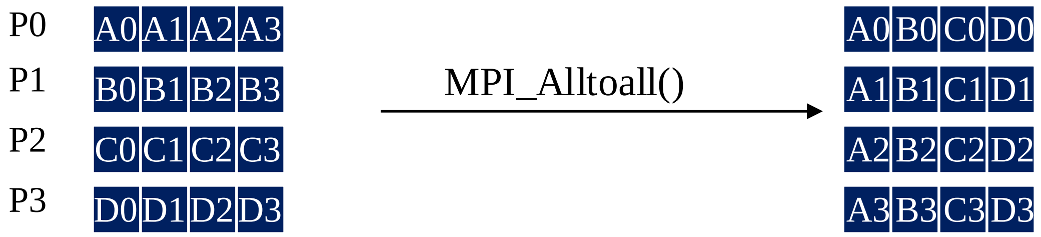
```
int MPI_Bcast( void *buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm )
int MPI_Gather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
int MPI_Scatter(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
```

More Collective Data Movement

Gather a chunk from each P and put it into a single array. Each P gets a copy of the resulting array.



All to All: Take chunks of data on each P and spread them out among the corresponding arrays on each P



```
int MPI_Allgather(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
int MPI_Alltoall(const void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
```

MPI Collectives: Summary

- 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 through a barrier which blocks until all the processes in the process group associated with the communicator call it.
  - **`MPI_Barrier( comm )`**

Collective operations are powerful … use them when you can

Do not implement them from scratch on your own. Think about how you’d implement, for example, a reduction.

It is MUCH harder than you might think.

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

Message passing: Basic ideas and jargon

- We need to coordinate the execution of processes … which may be spread out over a collection of independent computers
- Coordination:
  1. Process management (e.g., create and destroy)
  2. Synchronization … timing constraints for concurrent processes)
  3. Communication ... Passing a buffer from one machine to another

- A message passing interface builds coordination around messages (either explicitly or implicitly).

- The fundamental (and overly simple) timing model for a message:

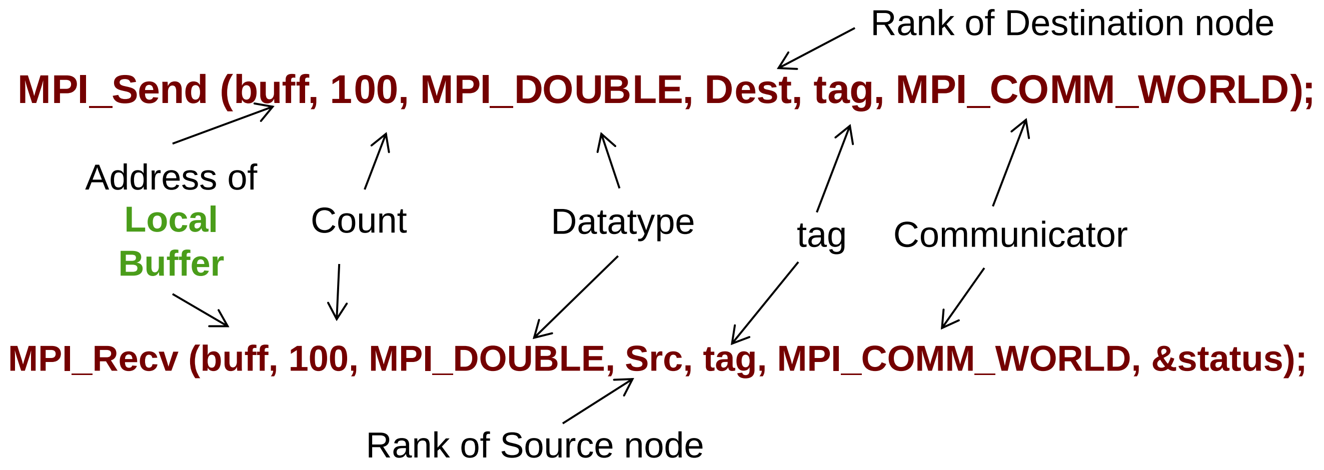
$$\text{Time}_{\text{communication}} = \text{latency} + N_{\text{bytes}}/\text{bandwidth}$$

Network fixed costs plus overheads

Network asymptotic bytes per second

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: a user-defined int to keep track of different messages from a single source



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

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);
```

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 sourc
     int tag, MPI_Comm comm,
     MPI_Status* status)
```

C language comments:

- **`void*`** says the argument can take a pointer to any type. The C compiler won't do any type checking … it just needs a valid address to a block of memory.
- A type with a * means the function expects a pointer to that type. So I would declare a variable as **`MPI_Status MyStat`** and then put the variable in the function call with an ampersand (**`&`**) … for example **`&MyStat`**

MPI_Status is a variable that contains information about the message that is received.
about the received message. The most common usage is to find out how many items w

```
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);
```

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 defines predefined data types that must be specified when passing messages.

Exercise: Ping-Pong Program

We provide a program you can start with called ping_pong.c … it has all the C code you need for this exercise other than the calls to the message passing functions.

- Goal
  - Measure the latency of our communication network.
- Program
  - Create a program to bounce a message (a single value) 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

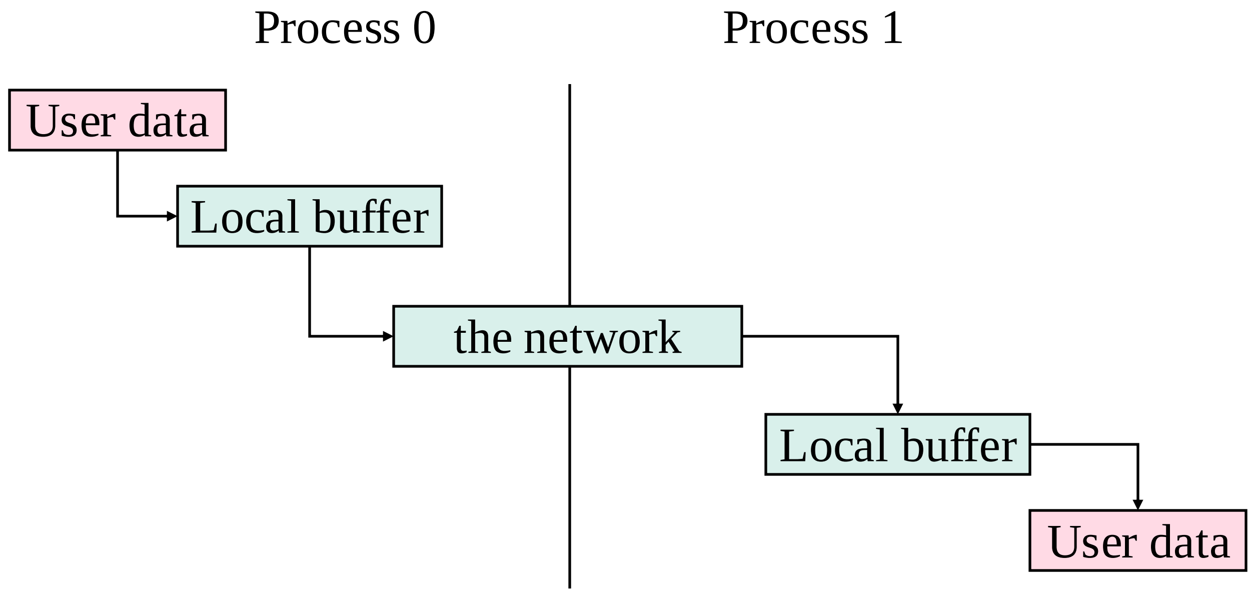
```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#define VAL 42
#define NREPS  10
#define TAG 5

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;
   MPI_Barrier(MPI_COMM_WORLD);
   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();
}
```

Outline

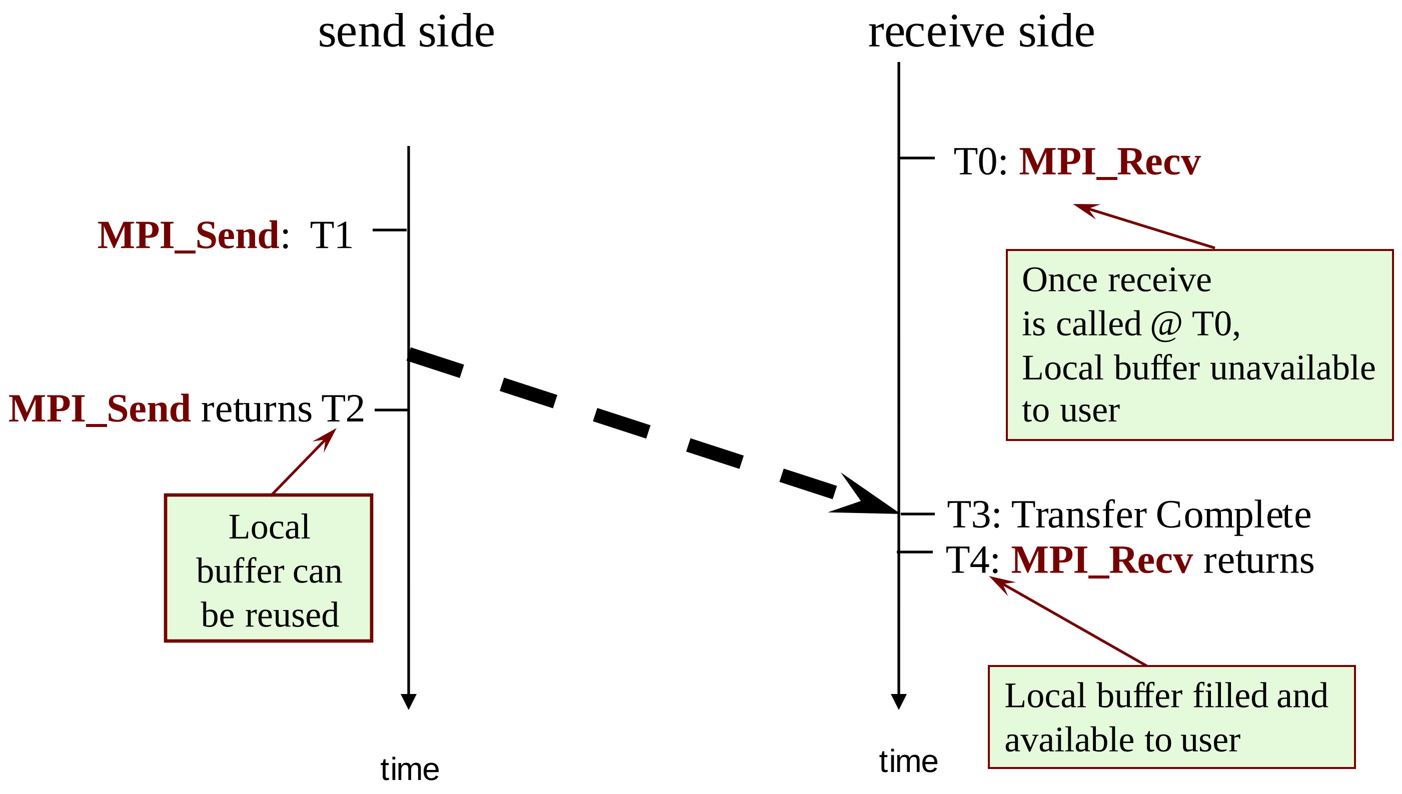
- MPI and distributed memory systems
- The Bulk Synchronous Pattern and MPI collective operations
- Introduction to message passing
- The diversity of message passing in MPI
- Geometric Decomposition and MPI
- Concluding Comments

Buffers

- Message passing is straightforward, but there are subtleties
  - Buffering and deadlock
  - Deterministic execution
  - Performance
- When you send data, where does it go? The following is the typical flow:



Blocking Send-Receive Timing Diagram
(Receive before Send)


send side
receive side
T0: MPI_Recv
MPI_Send: T1
MPI_Send returns T2
Once receive is called @ T0, Local buffer unavailable to user
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.

Non-Blocking Communication

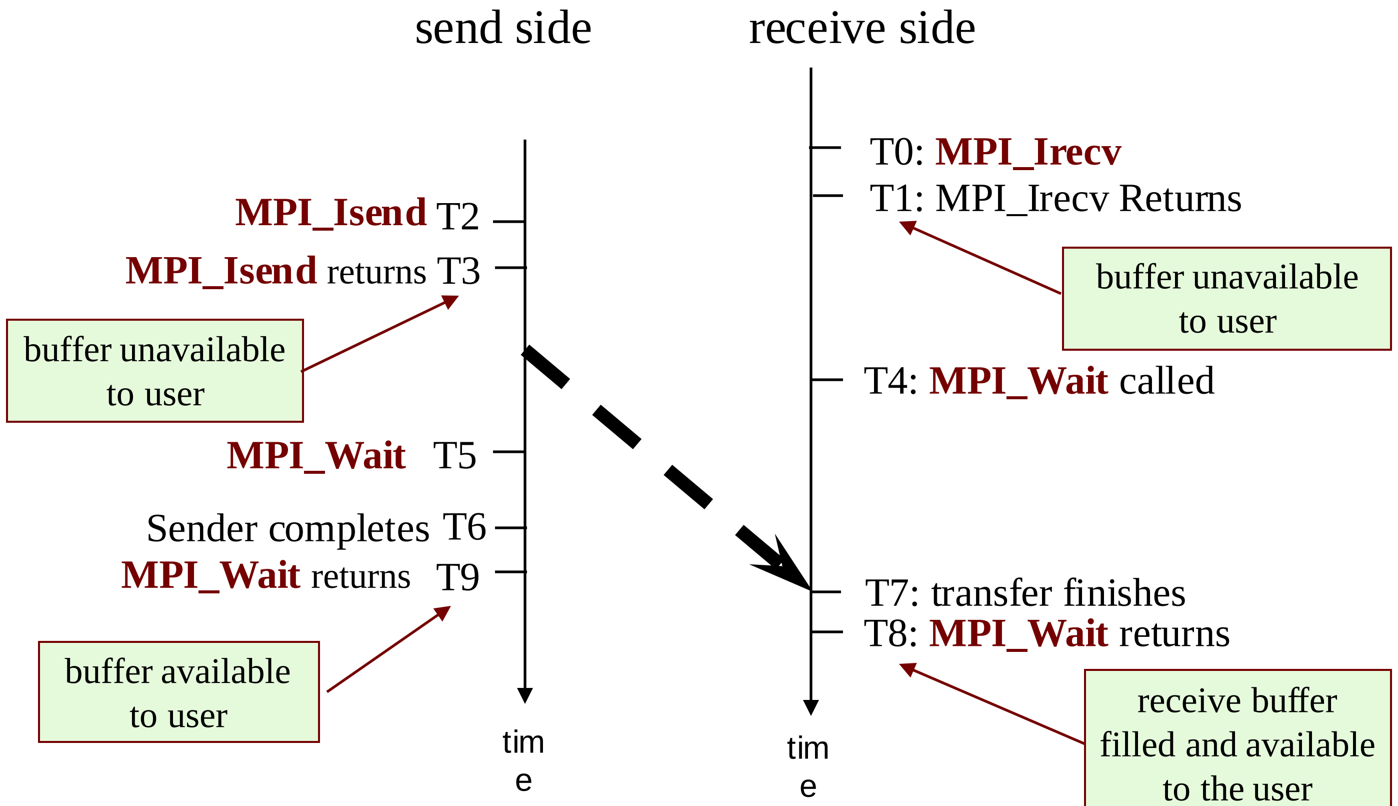
- Non-blocking operations return immediately and pass ‘‘request handles” that can be waited on and queried
  - **MPI_Isend(start, count, datatype, dest, tag, comm, request)**
  - **MPI_Irecv(start, count, datatype, src, tag, comm, request)**
  - **MPI_Wait(request, status)**
- One can also test without 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



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

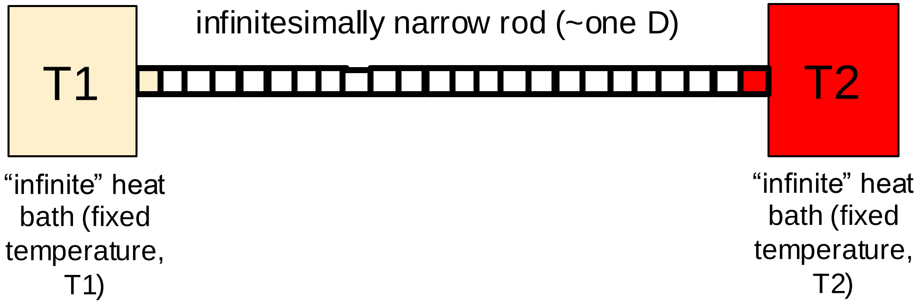
$$u_j^{n+1} = (1-2r)u_j^n + ru_{j-1}^n + ru_{j+1}^n \qquad r = k/h^2$$

- ▪ 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).

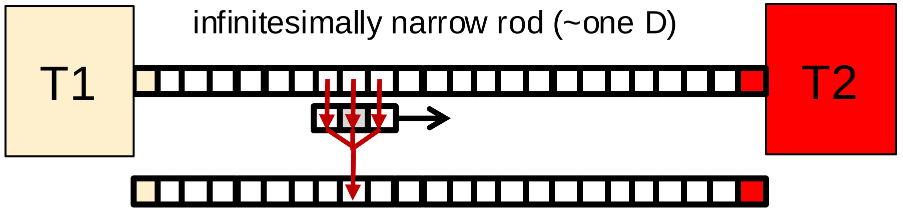
```
for (int t = 0; t < N_STEPS-1; ++t)
   for (int x = 1; x < N-1; ++x)
         u[t+1][x] = u[t][x] + r*(u[t][x+1] - 2*u[t][x] + u[t][x-1]);
```

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

Heat Diffusion equation

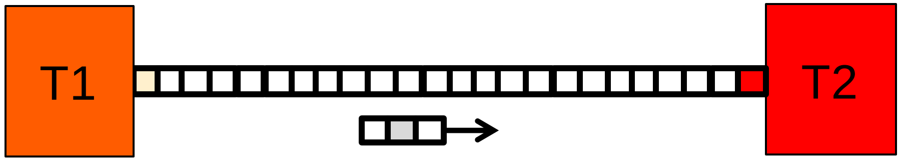

infinitesimally narrow rod (~one D)
T1
T2
“infinite” heat bath (fixed temperature, T1)
“infinite” heat bath (fixed temperature, T2)

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

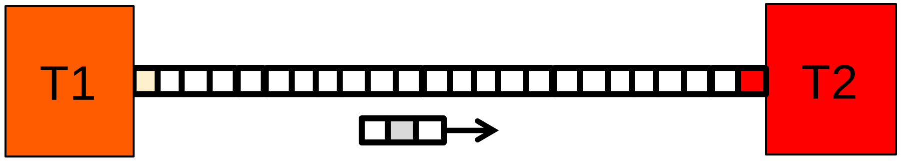


```
int main()
{
   double *u   = malloc (sizeof(double) * (N));
   double *up1 = malloc (sizeof(double) * (N));

   initialize_data(uk, ukp1, N, P); // initialize, 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

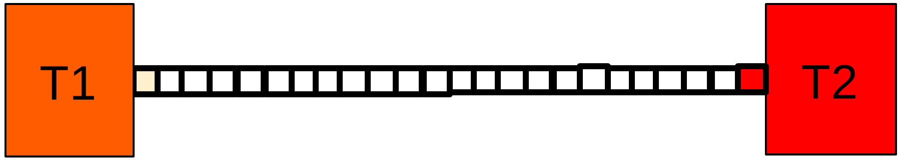


```
int main()
{
   double *u   = malloc (sizeof(double) * (N));
   double *up1 = malloc (sizeof(double) * (N));

   initialize_data(uk, ukp1, N, P); // initialize, 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?

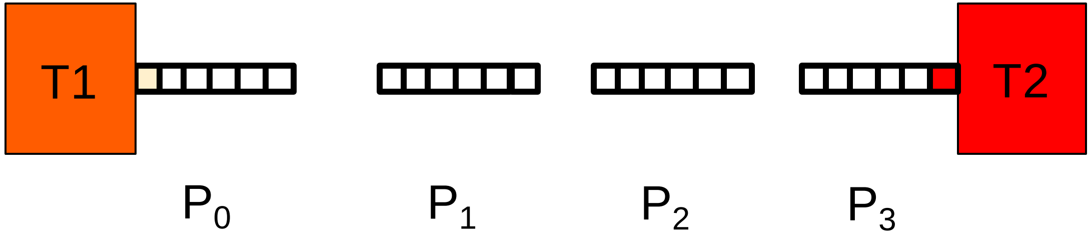
Heat Diffusion equation

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



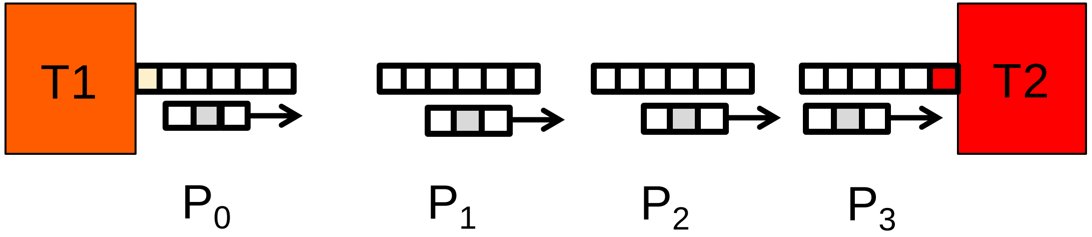
Heat Diffusion equation

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



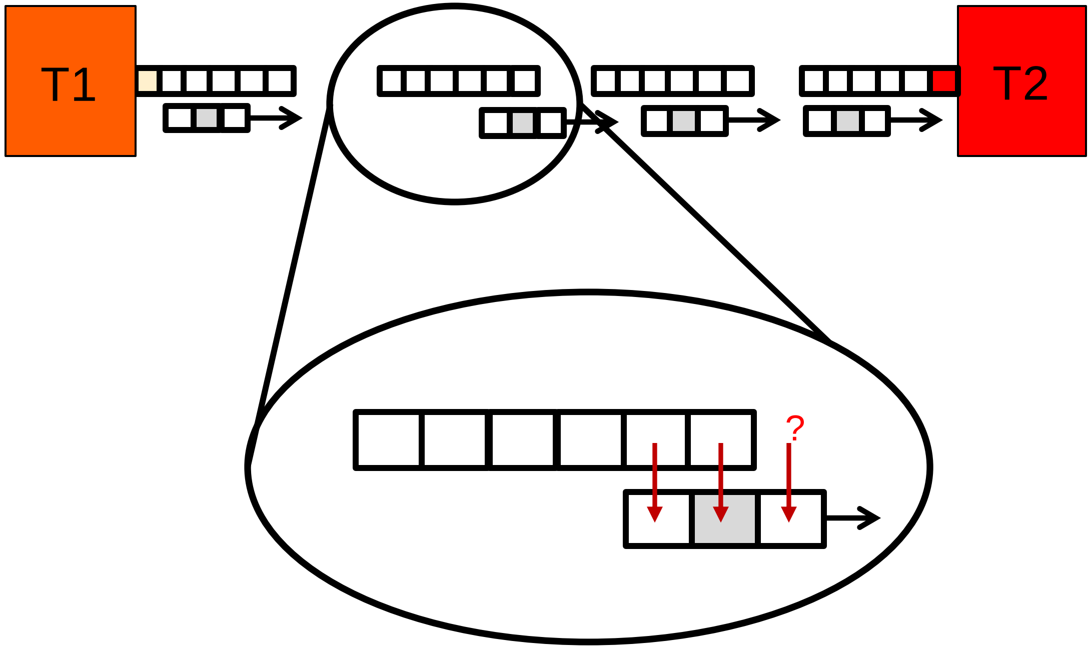
Heat Diffusion equation

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



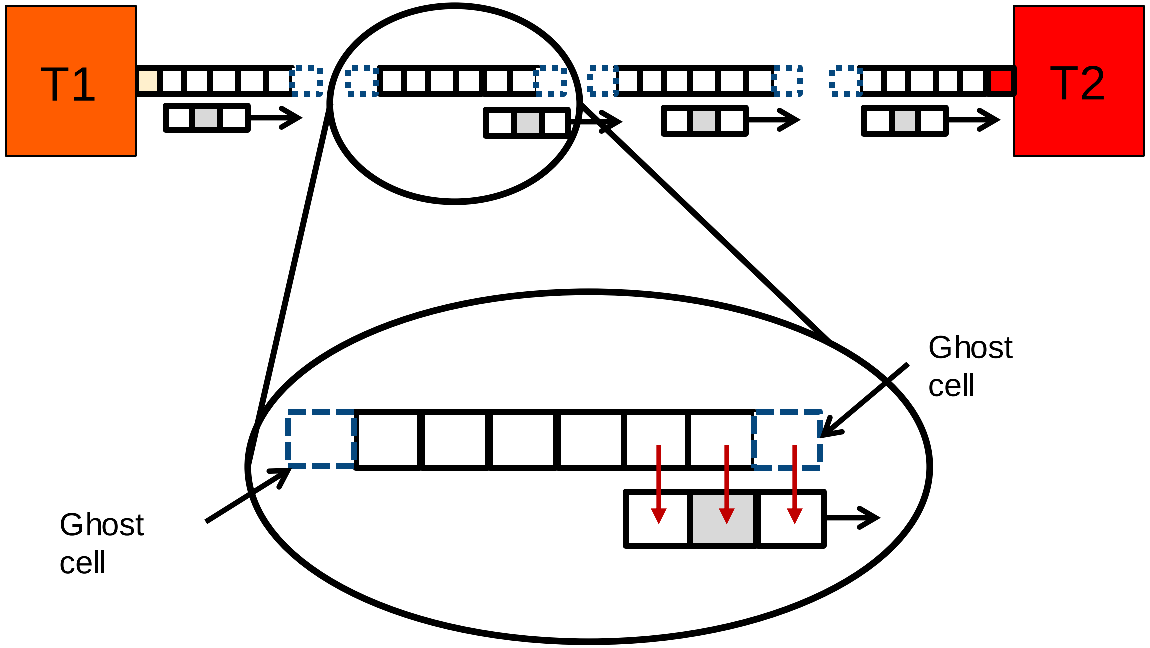
Heat Diffusion equation

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



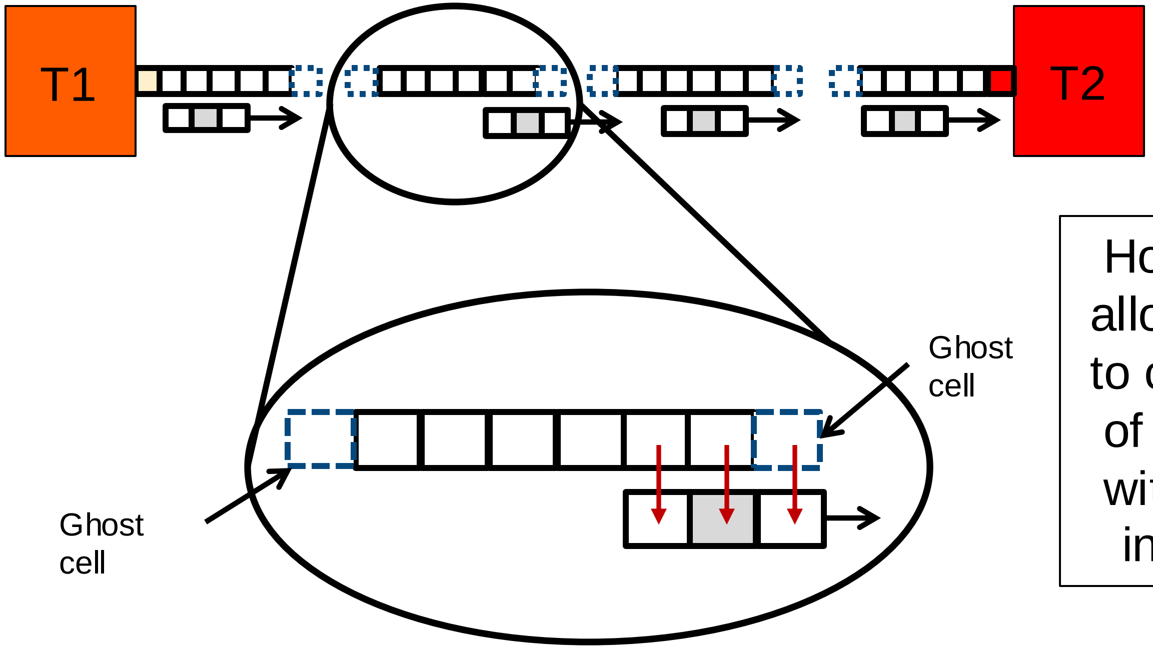
Heat Diffusion equation

- 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 equation

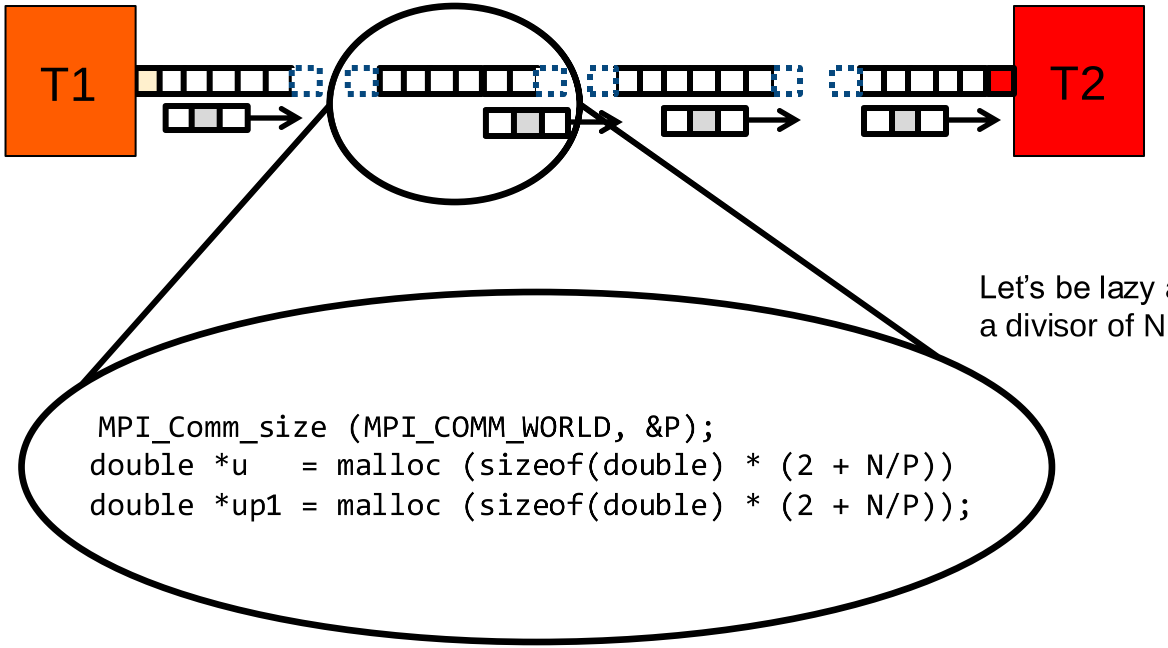
- We add ghost cells to the ends of each chunk, update them with the required values from neighbor chunks at each time step … hence giving the stencil everything it needs on any given chunk to update all of its values.



How would you allocate memory to create chunks of the right size with ghost cells in your code?

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.

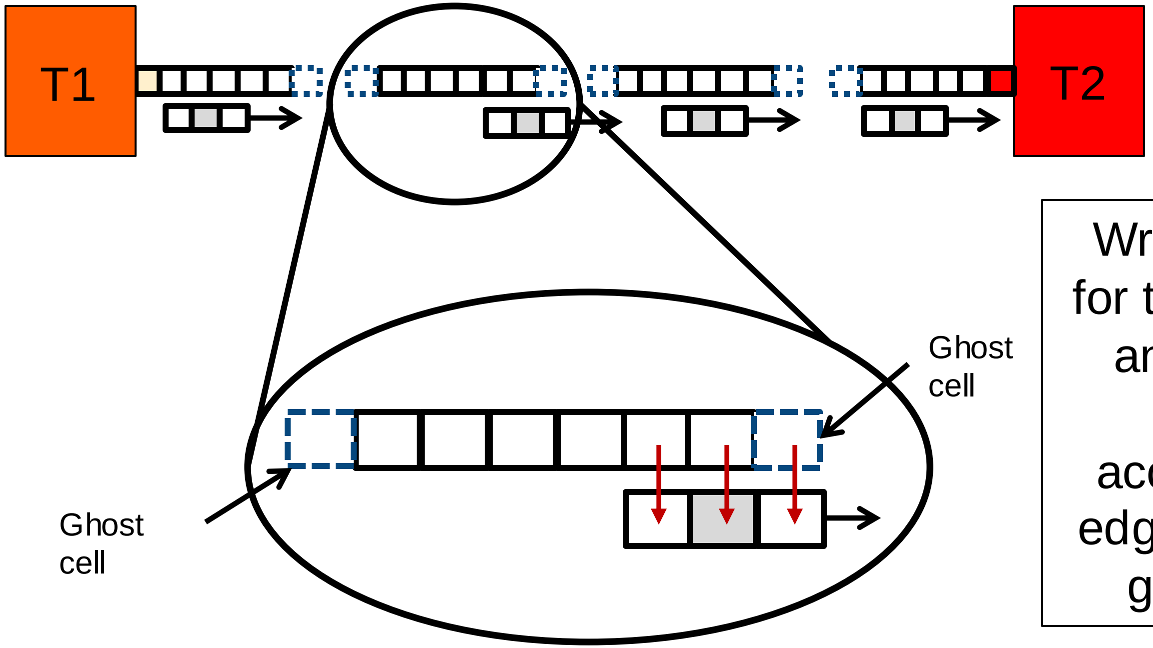


```
MPI_Comm_size (MPI_COMM_WORLD, &P);
double *u   = malloc (sizeof(double) * (2 + N/P))
double *up1 = malloc (sizeof(double) * (2 + N/P));
```

Let’s be lazy and assume P is a divisor of N (i.e.; N%P = 0)

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.



Write the code for the update of an individual chunk … accounting for edges using the ghost cells.

Heat Diffusion MPI Example: Updating a chunk

```
// 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;
```

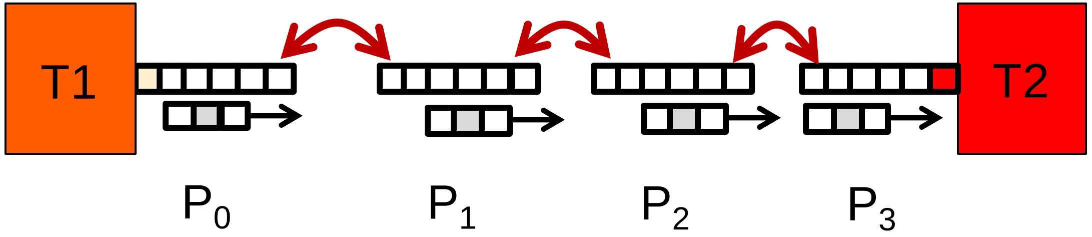
Update array values using local data and values from ghost cells.

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: Communication

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



Try to write the code for this communication pattern.

Heat Diffusion MPI Example

Note: the edges of domain are held at a fixed temperature.

- Node 0 has no neighbor to the left
- Node P has no neighbor to its right

```
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” to hold
double *up1 = malloc (sizeof(double) * (2 + N/P)); // values from my neighbors

initialize_data(uk, ukp1, N, P);
for (int t = 0; t < N_STEPS; ++t){
```

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

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

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

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

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

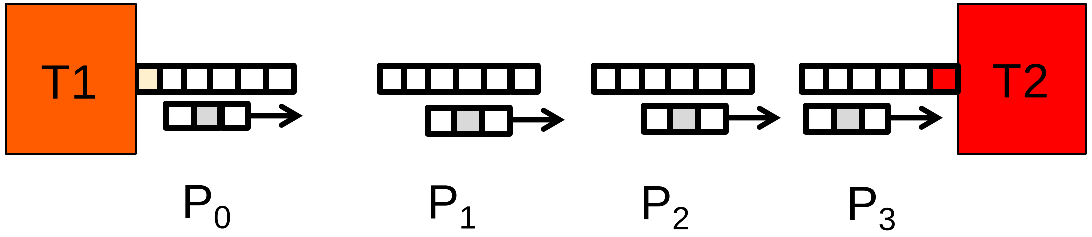
```
  if (myID != P-1) MPI_Send (&u[N/P], 1, MPI_DOUBLE, myID+1, 0, MPI_COMM_WORLD);
```

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

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

Heat Diffusion equation

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



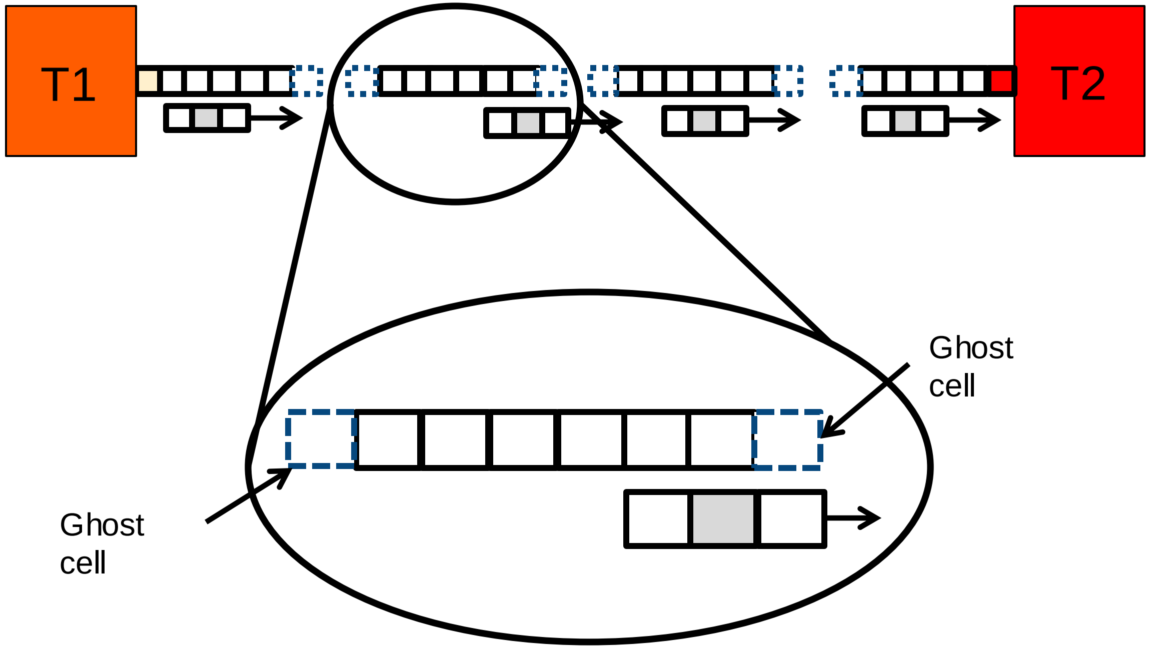
We now put all the pieces together for the full program

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” to hold
double *up1 = malloc (sizeof(double) * (2 + N/P)); // 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;
```

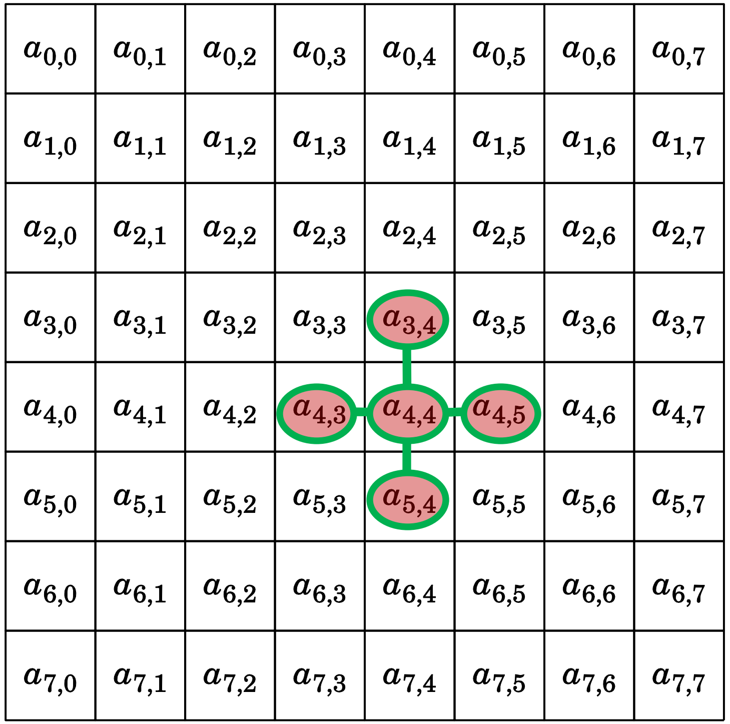
The Geometric Decomposition Pattern

- 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 and stencil →

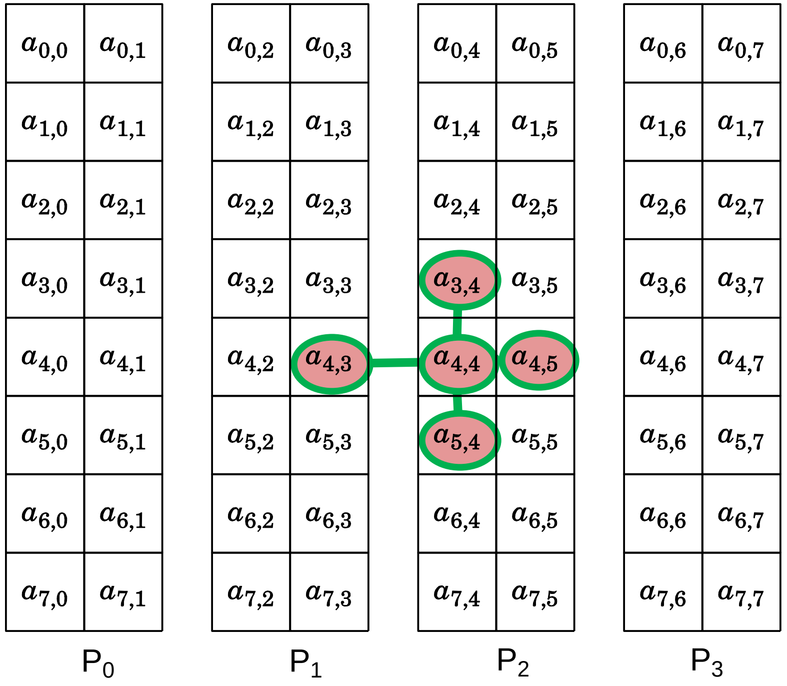

$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$ $a_{0,4}$ $a_{0,5}$ $a_{0,6}$ $a_{0,7}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,4}$ $a_{1,5}$ $a_{1,6}$ $a_{1,7}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{2,4}$ $a_{2,5}$ $a_{2,6}$ $a_{2,7}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$ $a_{3,4}$ $a_{3,5}$ $a_{3,6}$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

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) values** per processor

P is the # of processors

N is the order of our square matrix



$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$ $a_{0,4}$ $a_{0,5}$ $a_{0,6}$ $a_{0,7}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,4}$ $a_{1,5}$ $a_{1,6}$ $a_{1,7}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{2,4}$ $a_{2,5}$ $a_{2,6}$ $a_{2,7}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$ $a_{3,4}$ $a_{3,5}$ $a_{3,6}$ $a_{3,7}$
$a_{4,0}$ $a_{4,1}$ $a_{4,2}$ $a_{4,3}$ $a_{4,4}$ $a_{4,5}$ $a_{4,6}$ $a_{4,7}$
$a_{5,0}$ $a_{5,1}$ $a_{5,2}$ $a_{5,3}$ $a_{5,4}$ $a_{5,5}$ $a_{5,6}$ $a_{5,7}$
$a_{6,0}$ $a_{6,1}$ $a_{6,2}$ $a_{6,3}$ $a_{6,4}$ $a_{6,5}$ $a_{6,6}$ $a_{6,7}$
$a_{7,0}$ $a_{7,1}$ $a_{7,2}$ $a_{7,3}$ $a_{7,4}$ $a_{7,5}$ $a_{7,6}$ $a_{7,7}$
P_0 P_1 P_2 P_3

Partitioned Arrays: Block distribution

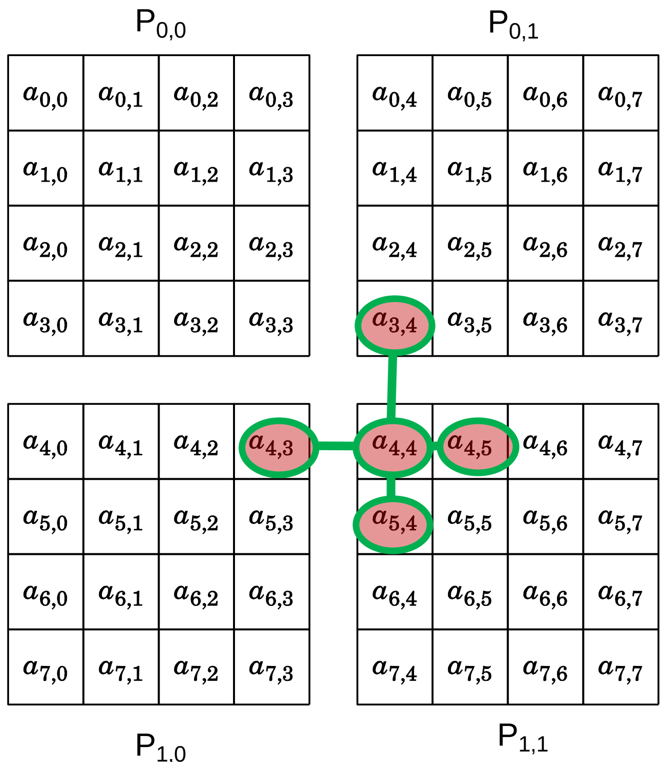
- If we parallelize in both dimensions, then we have $(N/P^{1/2})^2$ elements per processor, and we need to send **$O(N/P^{1/2})$ values** from each processor. Asymptotically better than O(N).

P is the # of processors

Assume a p by p square mesh … $p=P^{1/2}$

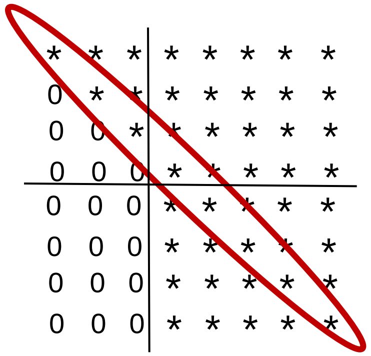
N is the order of our square matrix

Dimension of each block is $N/P^{1/2}$

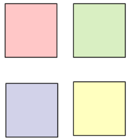

$P_{0,0}$
$P_{0,1}$
$a_{0,0}$ $a_{0,1}$ $a_{0,2}$ $a_{0,3}$ $a_{0,4}$ $a_{0,5}$ $a_{0,6}$ $a_{0,7}$
$a_{1,0}$ $a_{1,1}$ $a_{1,2}$ $a_{1,3}$ $a_{1,4}$ $a_{1,5}$ $a_{1,6}$ $a_{1,7}$
$a_{2,0}$ $a_{2,1}$ $a_{2,2}$ $a_{2,3}$ $a_{2,4}$ $a_{2,5}$ $a_{2,6}$ $a_{2,7}$
$a_{3,0}$ $a_{3,1}$ $a_{3,2}$ $a_{3,3}$ $a_{3,4}$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

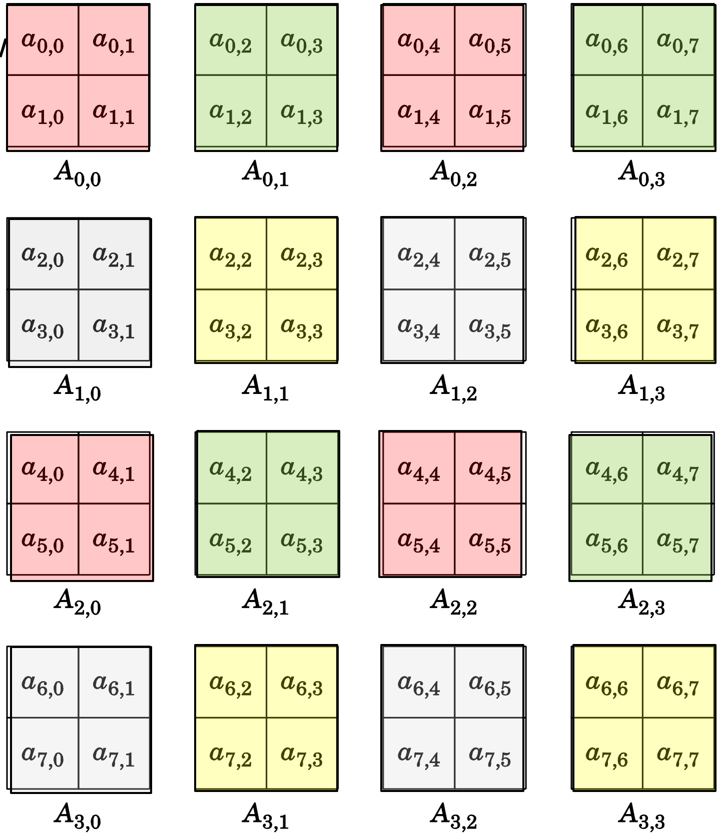
Partitioned Arrays: block cyclic distribution

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



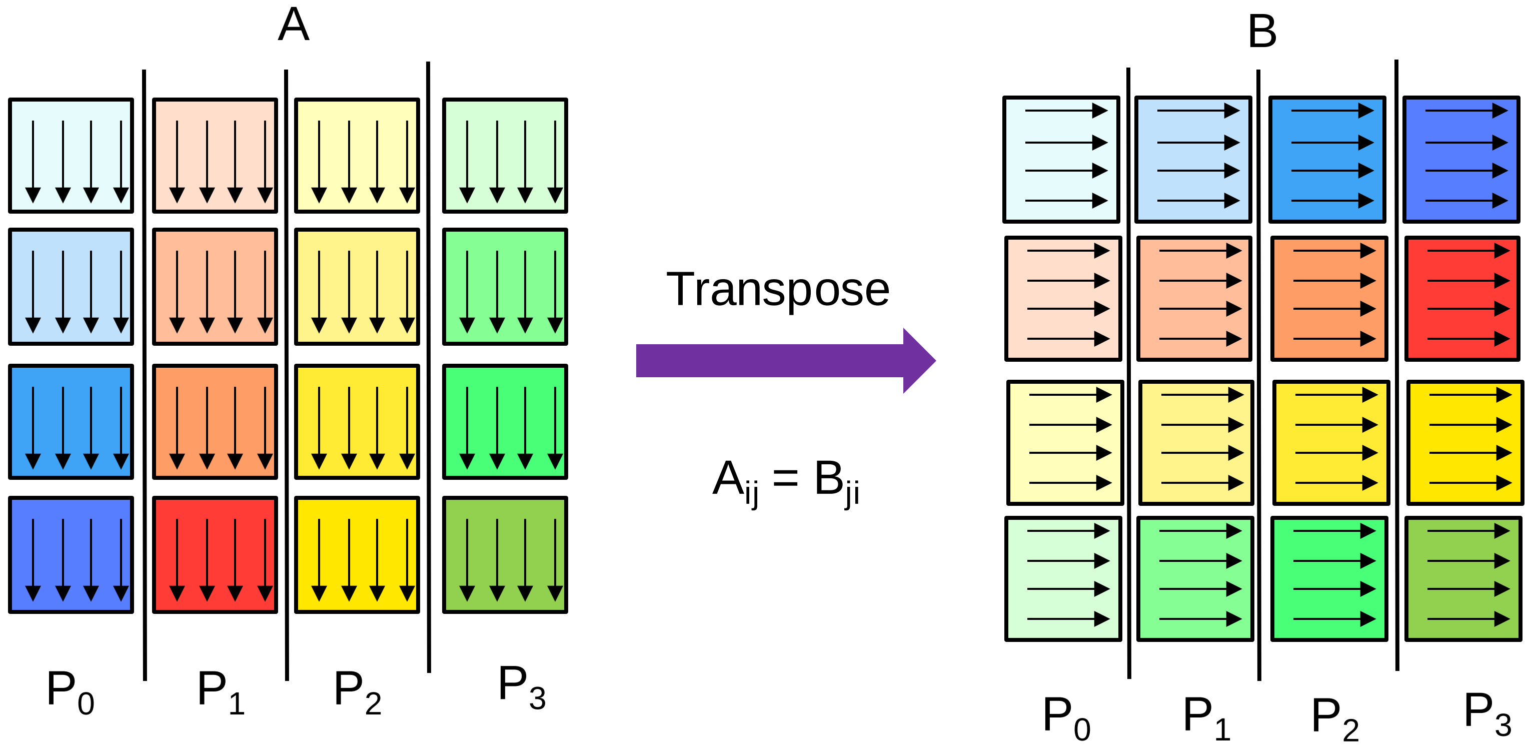
  - Zeros fill in the right lower triangle of the matrix … less work to do.
  - Balance load with cyclic distribution of blocks of A mapped onto a grid of nodes (2x2 in this case … colors show the mapping to nodes).




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

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. The order of A and B is N. N = blk*P where blk is the order of the square subblocks

Matrix Transposition

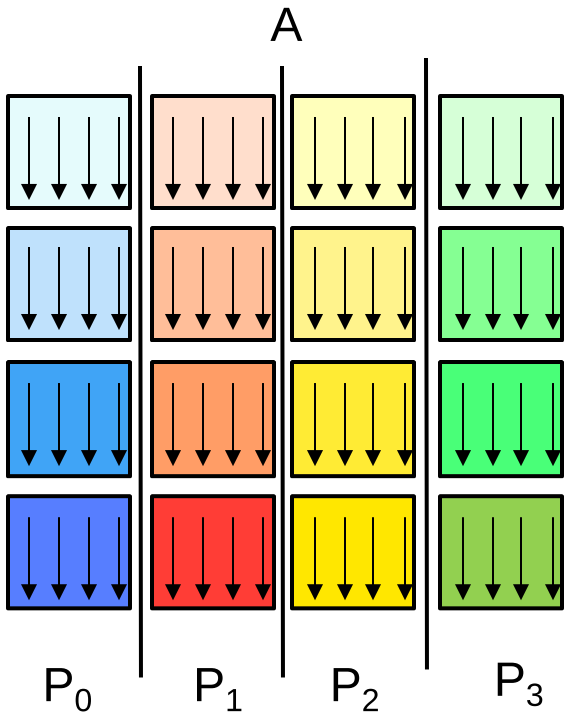
We are going to create a transpose program that uses the SPMD pattern.

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?



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

Matrix Transposition

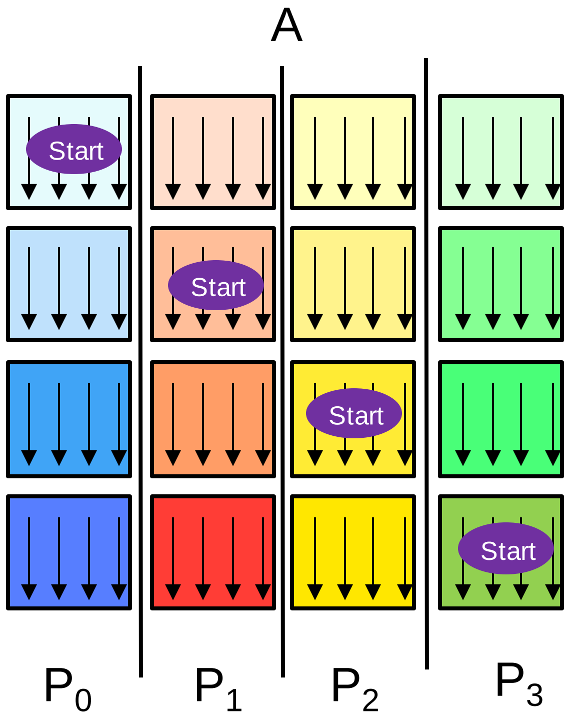
We are going to create a transpose program that uses the SPMD pattern.

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 until they hit the bottom of their column.

Phase 0 … transpose your diagonal

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

Matrix Transposition

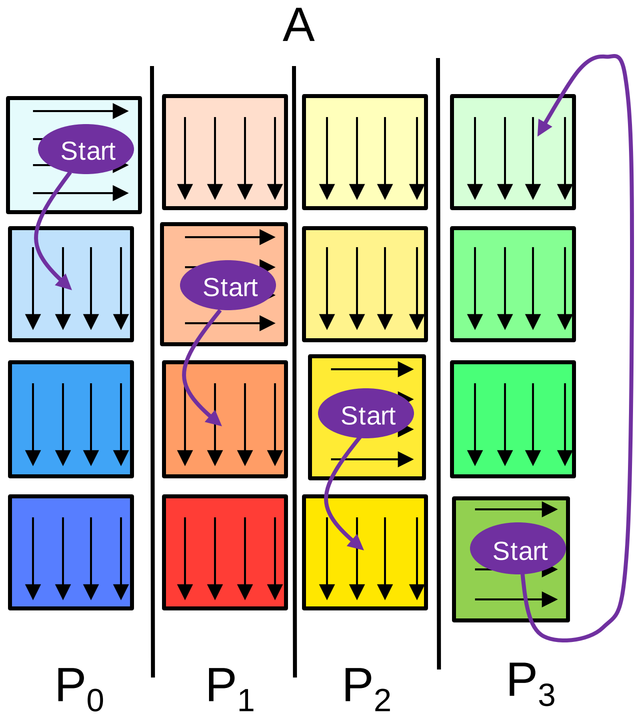
We are going to create a transpose program that uses the SPMD pattern.

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?



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"

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

Matrix Transposition

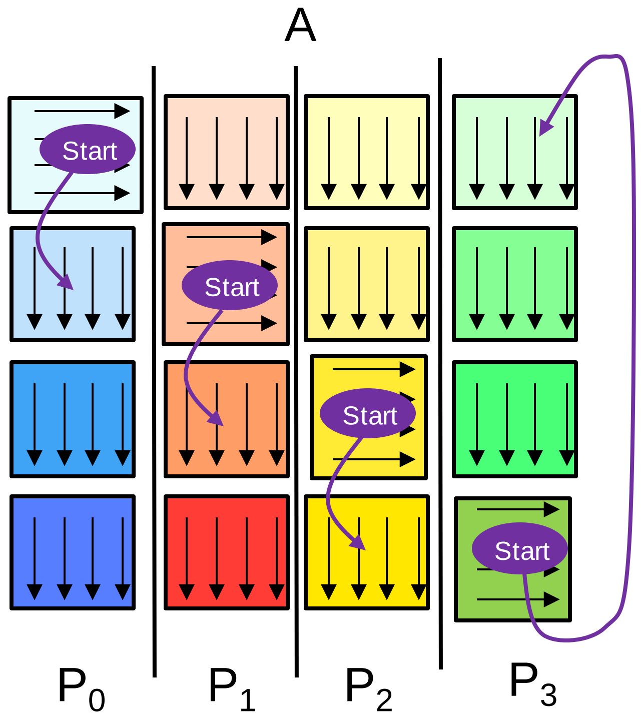
We are going to create a transpose program that uses the SPMD pattern.

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?



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"

We know the sender … who receives the block?

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

Matrix Transposition

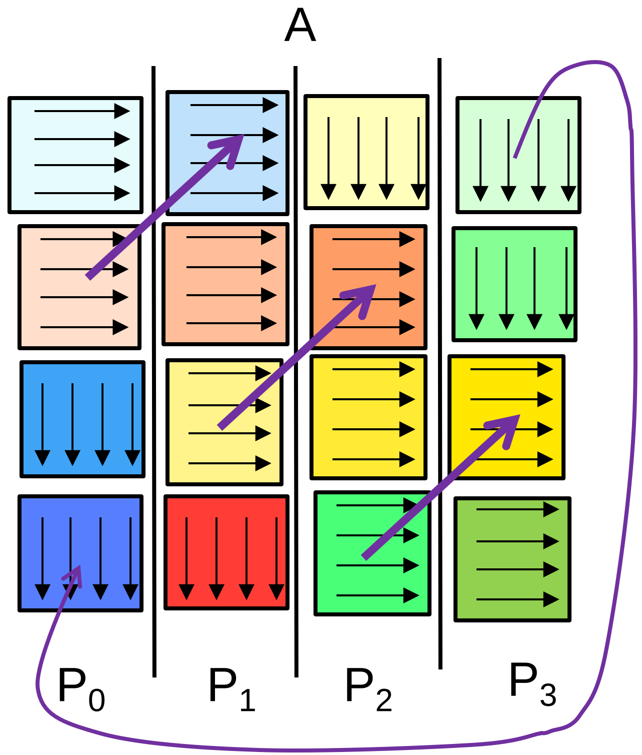
We are going to create a transpose program that uses the SPMD pattern.

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?



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"

We know the sender … who receives the block?

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

Exercise: Matrix Transpose Program

- Start with the basic transpose program we provide (transpose.c and trans_sendrcv.c functions).
- Your task … deduce a general expression for the sender and receiver (FROM and TO) for each phase.
- Go to trans_sendrcv.c and enter your definitions for the TO and FROM macros (**<u>what is there now is wrong</u>** … I just wanted something to show how macros work).
- 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_Request request;

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

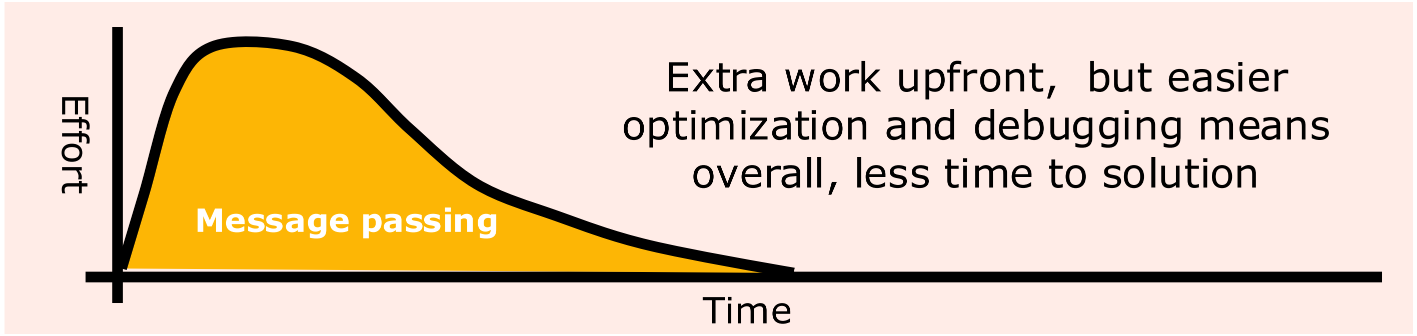
10

The ~~12~~ core functions in MPI

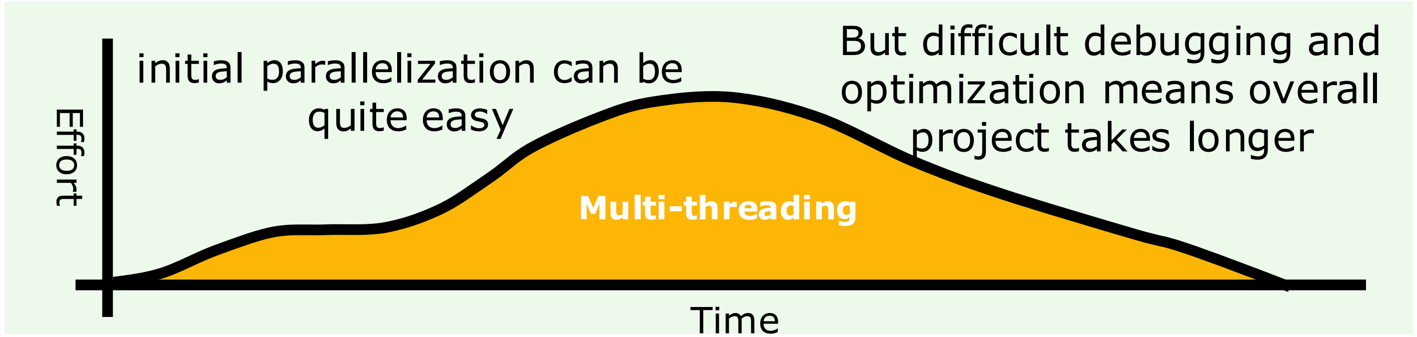
- MPI_Init
- MPI_Finish
- MPI_Comm_size
- MPI_Comm_rank
- ~~MPI_Send~~
- ~~MPI_Recv~~
- MPI_Reduce
- MPI_Isend
- MPI_Irecv
- MPI_Wait
- MPI_Wtime
- MPI_Bcast

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

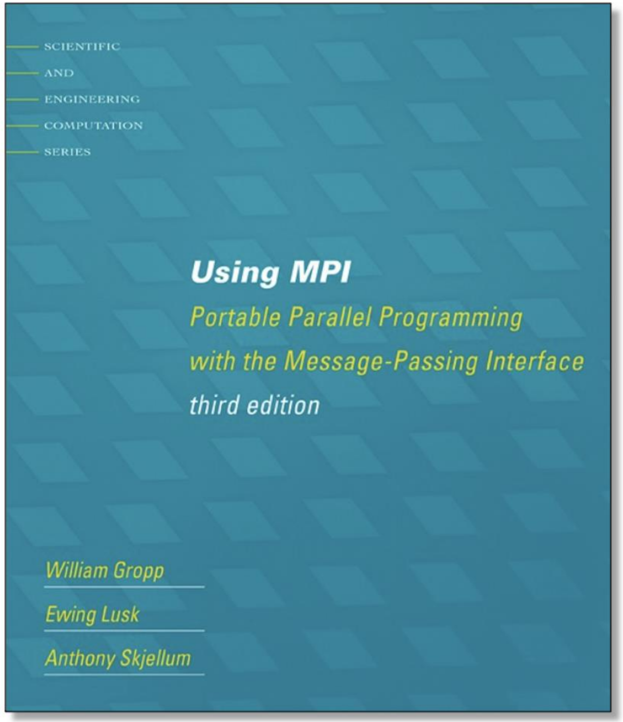
But difficult debugging and optimization means overall project takes longer

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

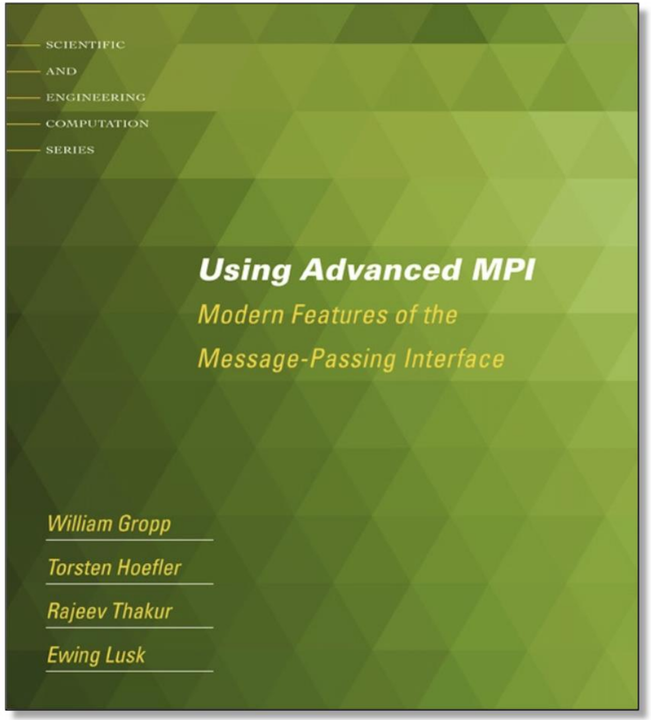
*P. N. Klein, H. Lu, and R. H. B. Netzer, Detecting Race Conditions in Parallel Programs that Use Semaphores, Algorithmica, vol. 35 pp. 321–345,

MPI References

- The Standard itself at http://www.mpi-forum.org
- Additional tutorial information at http://www.mcs.anl.gov/mpi
- The core reference books:



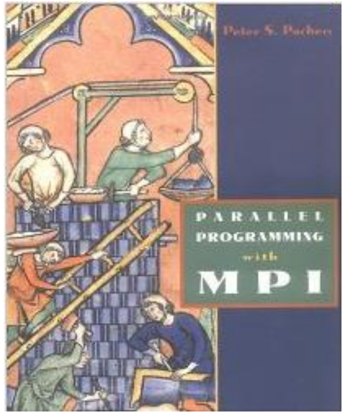
Basic MPI



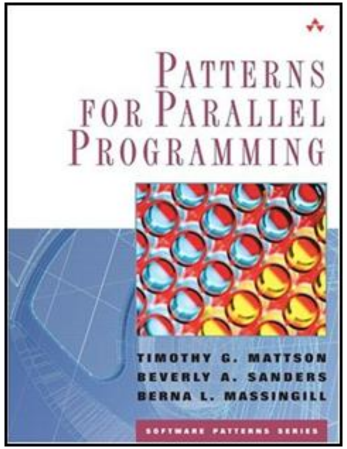
Advanced MPI, including MPI-3

Additional books to help you master MPI

- *Parallel Programming with MPI*, by Peter Pacheco, Morgan-Kaufmann, 1997.
  - Only covers MPI 1.0 so it's out of date, but it is a very friendly and gentle introduction.
  - Peter Pacheco is a teacher first and foremost and that shows in the way he organizes the material in this book.



- *Patterns for Parallel Programing*, by Tim Mattson, Beverly Sanders, and Berna Massingill.
  - Only covers MPI 1.0 so it's out of date.
  - Focusses on how to use MPI, not the structure of the standard itself.
  - Shows how patterns are expressed across MPI, OpenMP, and concurrent Java



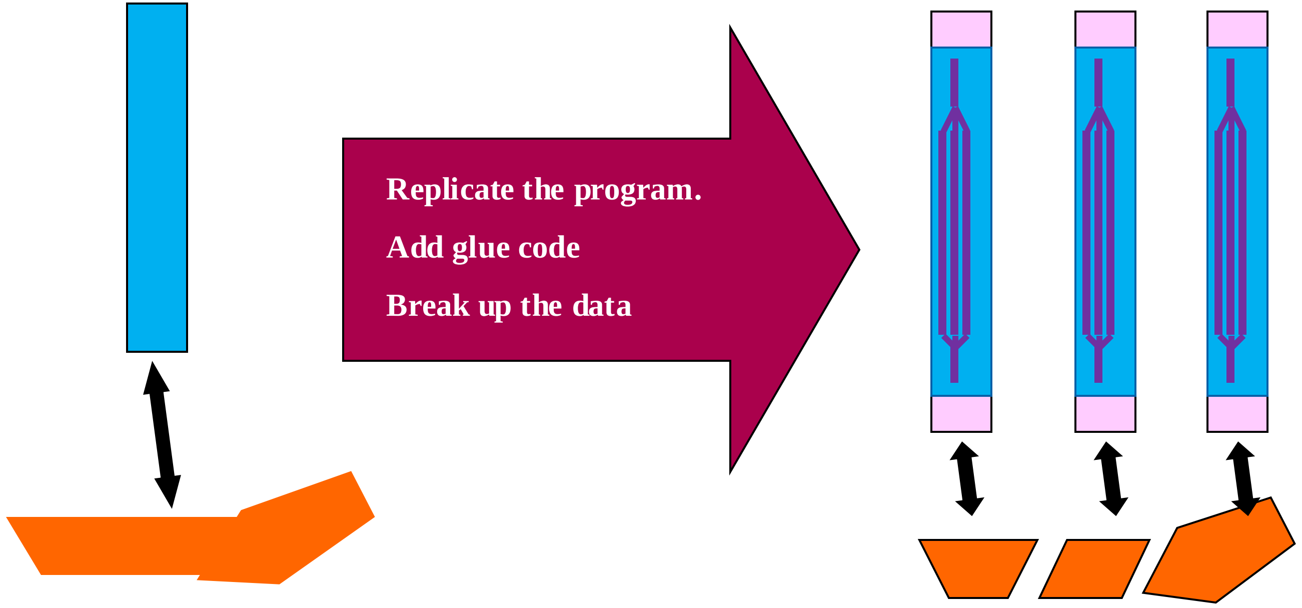
Backup

- 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.



Pi program with MPI and OpenMP

```
#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) ;
          MPI_Finalize();
}
```

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

For many years, this was all you needed to do to make OpenMP and MPI work together.

Don’t put MPI calls in a parallel region, and everything just works.

Technically, this doesn’t work anymore.

You must tell MPI at initialization about planned Thread use

- MPI includes a version of MPI_Init() that defines how to handle threads. If you are going to mix threads with MPI, you required to use this new initialization function.

int MPI_Init_thread(int *argc, char **argv, int required, int *provided)

- int *argc: number of values on the command line.
- char ***argv: Pointer to and array of pointers holding the arguments as character strings
- Int MPI threading mode that you require
- Int * provided: a pointer to an int that identifies the thread mode you got.

MPI defines four constants that represent the different thread modes

1. **MPI_THREAD_SINGLE:** Only one thread will execute.
2. **MPI_THREAD_FUNNELED:** The process may be multi-threaded, but only the initial thread will make MPI calls (all MPI calls are funneled to the initial thread).
3. **MPI_THREAD_SERIALIZED:** The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are serialized).
4. **MPI_THREAD_MULTIPLE:** Multiple threads may call MPI, with no restrictions.

The 4 constants are ordered integers of type int .. That is Multiple>Serialized>Funneled>Single

Pi program with MPI and OpenMP

```
#include <mpi.h>
#include “omp.h”
void main (int argc, char *argv[])
{
        int i, my_id, numprocs,got;  double x, pi, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init_thread(&argc, &argv,MPI_THREAD_FUNNELED, &got) ;
        if(got<MPI_THREAD_FUNNELED)  MPI_Abort();
        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) ;
        MPI_Finalize();
}
```

Funneled has never let me down.

… Stil, it is recommended that you always verify you actually got the level of thread support you requested

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

Use homebrew to install gnu compilers on your Apple laptop

I tested this on a new (July 2023) MacBook Air with an Apple M2 CPU

Warning: by default Xcode usese the name gcc for Apple's clang compiler.
Use Homebrew to load a real, gcc compiler.

- Go to the homebrew web site (brew.sh). Cut and paste the command near the top of the page to install homebrew (in /opt/homebrew):

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

- Add /opt/homebrew/bin to your path. I did this by adding the following line to .zshrc

```
% export PATH=/opt/homebrew/bin:$PATH
```

- Install the latest gcc compiler

```
% brew install gcc
```

- This will install the compiler in /opt/homebrew/bin. Check /opt/homebrew/bin to see which gcc compiler was installed. In my case, it installed gcc-13
- Test the compiler (and the openmp option) with a simple hello world program

```
% gcc-13 –fopenmp hello.c
```

OpenMP and MPI on Apple Laptops: MacPorts

I have not tested this in a long time.
I greatly prefer homebrew.

But if you prefer MacPorts, this procedure should work.

- To use OpenMP and MPI on your Apple laptop:
- Download Xcode. Be sure to choose the command line tools that match your OS.
- Download and use MacPorts to install the latest gnu compilers.

`sudo port selfupdate`	Update to latest version of MacPorts
`sudo port install gcc14`	Grab version 13 gnu compilers
`port select --list gcc`	List versions of gcc on your system
`sudo port select --set gcc mp-gcc14`	Select the mp enabled version of the most recent gcc release
`sudo port install mpich-gcc14`	Grab the library that matches the version of your gcc compiler.
`mpicc –fopenmp hello.c`	Test the installation with a simple program
`mpiexec –n 4 ./a.out`	

MPI History

- MPI 4.1 specification (the latest as of December 2024).
  - Approved November 2, 2023 by the MPI Forum
  - Number of pages (not counting intro-text, appendices, or tool interfaces): 686 pages
- History
  - MPI-1.0: May 5, 1994
  - MPI-1.1: June 12, 1995
  - MPI-1.2: July 18, 1997
  - MPI-2.0: July 18, 1997
  - MPI-1.3: May 30, 2008
  - MPI-2.1: June 23, 2008
  - MPI-3.0: September 21, 2012
  - MPI-4.0: June 9, 2021
  - MPI-4.1: November 2, 2023

Currently they are working on a 4.2 spec (a clean-up and refine effort) and a more ambitious spec with new functionality that will be called 5.0.

Follow the work at: https://www.mpi-forum.org/