Parallel Programming … Programming across nodes[*]

Tim Mattson

Human Learning Group[**]

tgmattso@gmail.com

tim@timmattson.com

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

**a made-up company. Sometime I'm required to name an institution I belong to. I like "human learning" not "machine learning"

Disclaimer

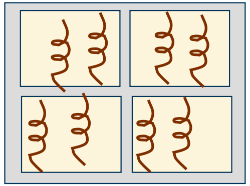
- The views expressed in this talk are those of the speaker.

- 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

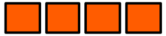
Hardware is diverse … and its only getting worse!!!

Write code with TBB or OpenMP

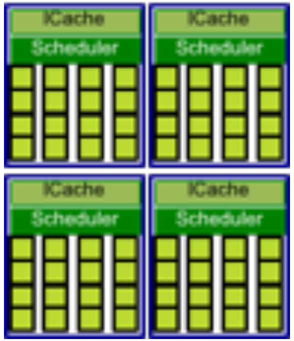


CPU

Work with the compiler to vectorize code



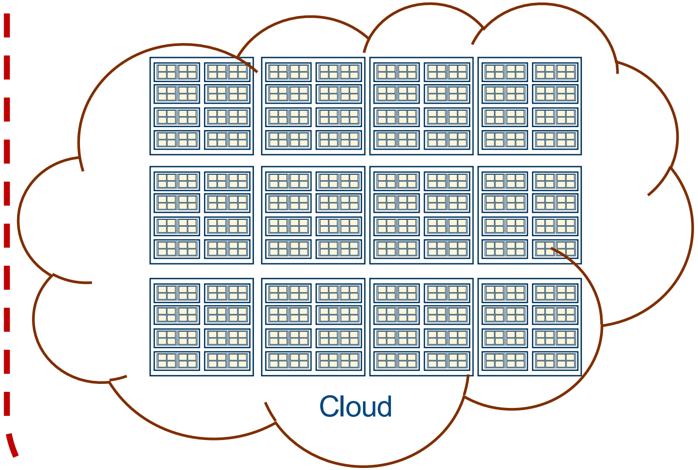
SIMD/Vector



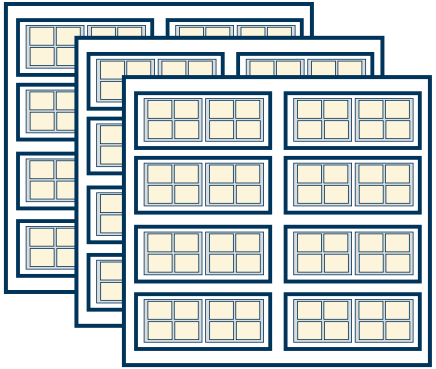
GPU

Use a portable API but if you must, use CUDA. It’s all the same model

Parallelism over disjoint address-spaces …. MPI

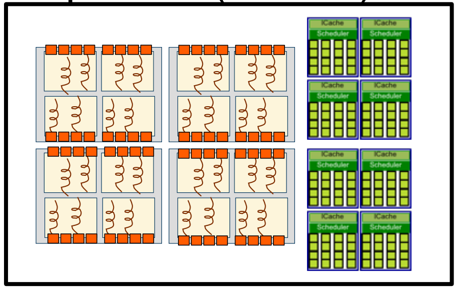


Cloud



Cluster

OpenMP lets you “do it all”. Or combine CUDA and OpenMP (or TBB).



Heterogeneous node

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).

Tim Mattson surfing at La Push Washington. The Beach Boys were right … ♫ Catch a wave and you’re sitting on top of the world. ♫

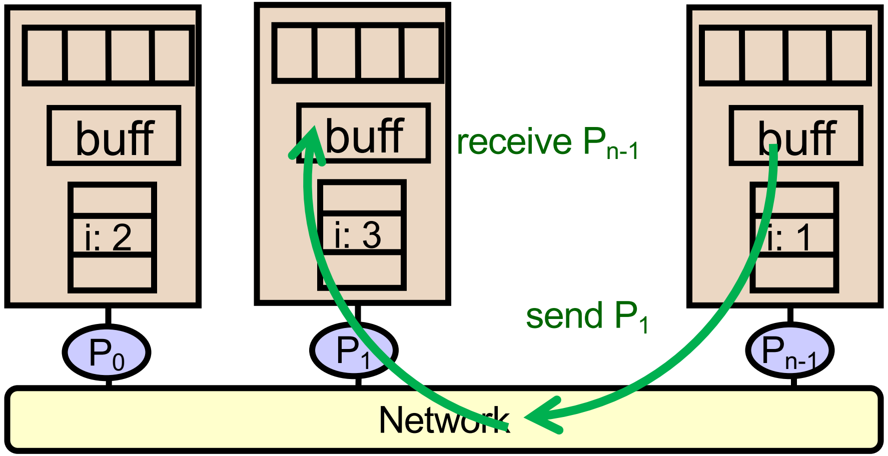
Outline

- ➡ 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

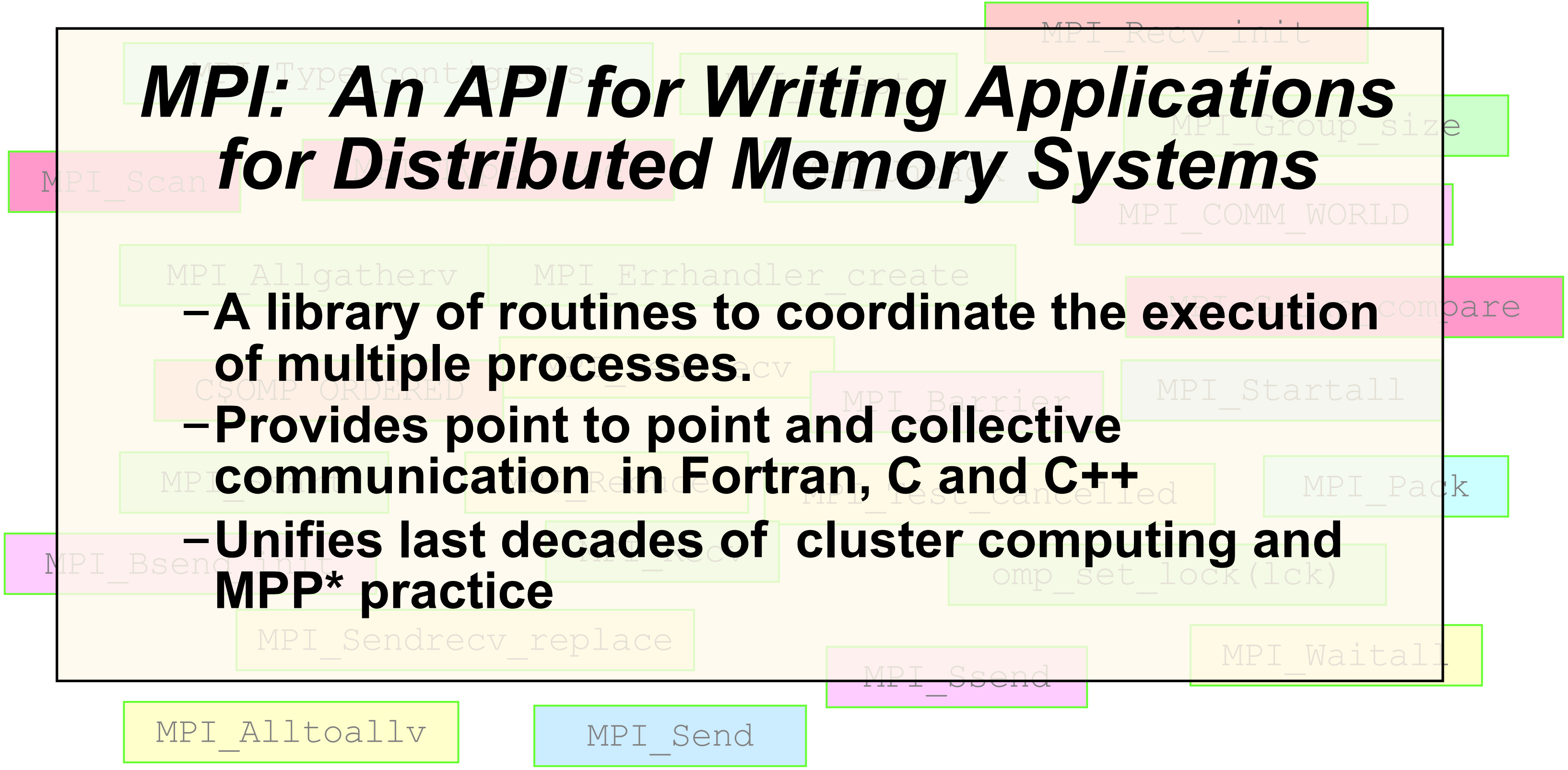
- Programs execute as a collection of processes.
  - Number of processes usually fixed at program startup time
  - Local address space per node -- **NO physically shared memory**.
  - **Logically** shared data is partitioned over local processes.

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



A collection of n MPI processes (P_0 to P_{n-1}) running on n nodes

MPI, the Message Passing Interface



MPI: An API for Writing Applications for Distributed Memory Systems

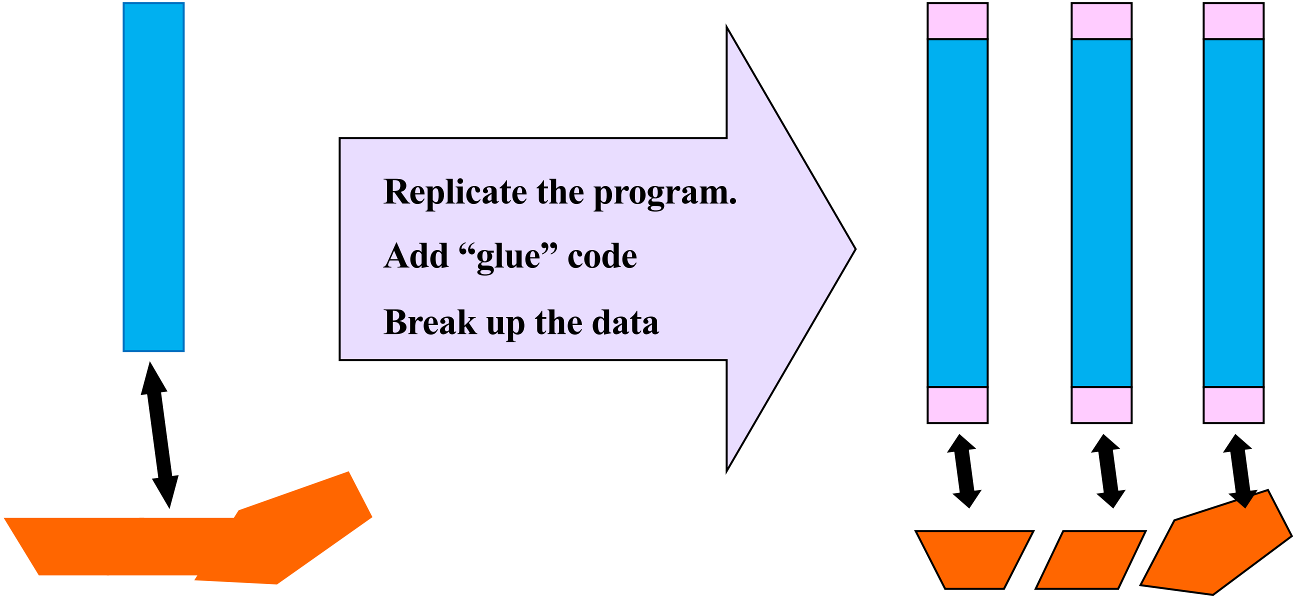
- **A library of routines to coordinate the execution of multiple processes.**
- **Provides point to point and collective communication in Fortran, C and C++**
- **Unifies last decades of cluster computing and MPP* practice**

*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
- Coordinate processes by passing messages.

A sequential program (blue) working on a data set (orange)



Replicate the program.

Add “glue” code

Break up the data

Glue code is what I call the code to initialize/finalize MPI and establish the communication context, rank, and process count.

Running MPI programs

MPI uses **mpirun** or **mpiexec** (or both) to launch programs on a cluster. They are largely equivalent. Just figure out which one is preferred on the system you are using.

- MPI implementations need a way to start “P processes” on the system.
- We do this with the mpirun command:

```
> mpirun –n P ./a.out
```

Run the program locally on P processes

Exercise: Hello world part 1

```
module load compilers/openmpi–4–1–5_gcc12.3
module load compilers/gcc–12.3_sl7
```

- Goal
  - To confirm that you can run a program on our cluster.
- Program
  - Write a program that prints “hello world” to the screen.
  - Execute across the nodes of our cluster using mpirun

- For our MPI work, we will use the following nodes: hpc-200-06-06, hpc-200-06-17, and hpc-200-06-18. Log into one of those nodes.
- MPI and the right gcc are present by default. If not, see ESC24 school-environment instructions for the modules you need to load.
- Then write your program (hello.c) and do the following:

```
$ gcc hello.c
$ mpirun –n 2 ./a.out
```

Once into the system, we drop the cr.cnaf.infn.it from the host name so we would reference node hpc-200-06-06.cr.cnaf.infn.it as hpc-200-06-06

Running MPI programs

MPI uses **mpirun** or **mpiexec** (or both) to launch programs on a cluster. They are largely equivalent. Just figure out which one is preferred on the system you are using.

- MPI implementations need a way to start “P processes” on the system.
- We do this with the mpirun command:

```
> mpirun –n P ./a.out
```

Run the program locally on P processes

- To run on different nodes, use a hostfile.

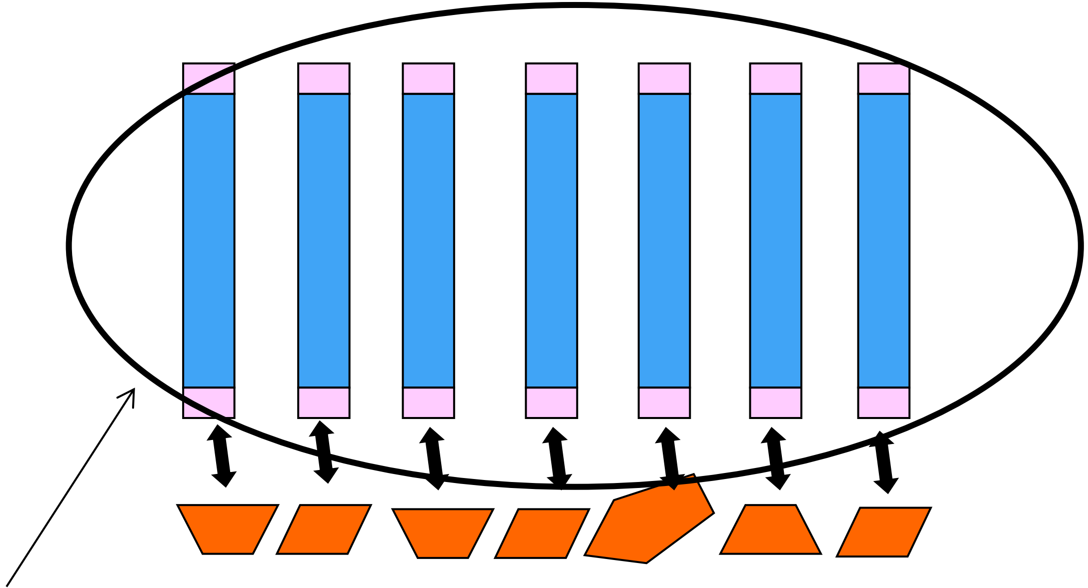
```
> mpirun –hostfile hostfile –n P ./a.out
```

Run the program as P processes on the nodes from hostfile. The hostfile has a node (a host) on each line followed by how many processes (slots) to allocated to each node. Here is an example for our cluster:

```
hpc-200-06-06 slots=2
hpc-200-06-17 slots=2
hpc-200-06-18 slots=2
```

An MPI program at runtime

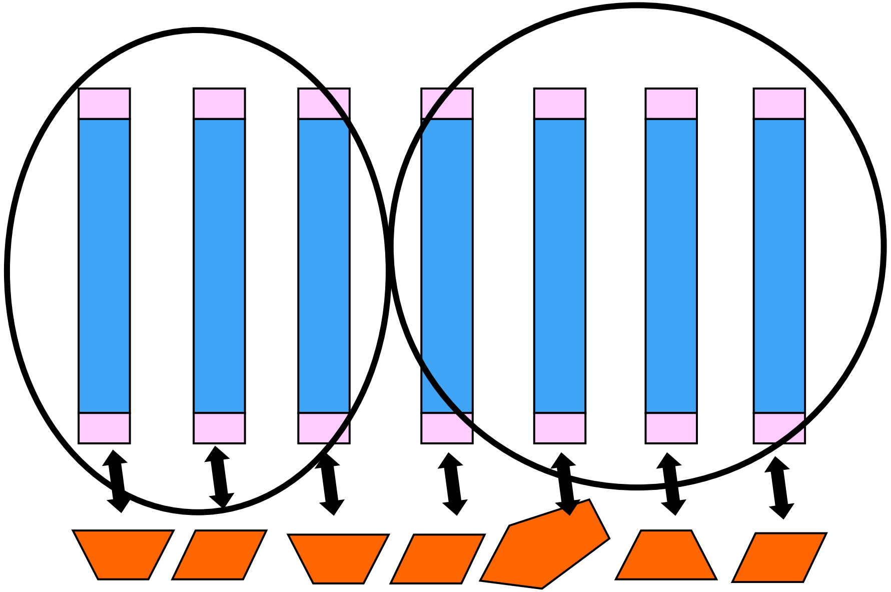
- Typically, when you run an MPI program, multiple processes all running the same program are launched … working on their own block of data.



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 **<u>process group</u>** into multiple subgroups to manage how processes are mapped onto different tasks

MPI Hello World 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;
}
```

MPI Hello World Program

`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

MPI Hello World Program

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

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

```
#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;
}
```

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

MPI Hello World Program

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”

```
#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;
}
```

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.

Compiling a program

- MPI provides a wrapper around the local compiler to create MPI programs.
- It is called mpicc or mpic++ or mpicxx …
- The wrapper provides the libraries and anything else required to support MPI compilation and linking. Additional arguments are passed directly to the compiler.
- It is important that the compiler on the local system matches the one used by mpicc/mpic++/mpicxx

```
> mpicc –o complexProg –O3 –fopenmp comp.c  mathyStuff.c  andMore.c
```

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

```
#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();
```

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
```

```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;
}
```

Without a hostfile, the processes launched by mpirun usually execute on the single node from which the command was issued.

To run on multiple nodes you need the host file

Running the program

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
   int rank, size;
   char name[MPI_MAX_PROCESSOR_NAME];
   int namLen;
   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &rank);
   MPI_Comm_size (MPI_COMM_WORLD, &size);
  MPI_Get_processor_name(name,&namLen);
   printf(" hello from %s process %d of nprocs = %d\n",name,ID, Nprocs);
   MPI_Finalize();
   return 0;
}
```

- On our 3 node cluster, I’d run this program (hello) as:

```
> mpirun –n 3 –hostfile hosts hello
hello from hpc-200-06-06.cr.cnaf.infn.it rank=0 of nprocs = 3
 hello from hpc-200-06-18.cr.cnaf.infn.it rank=2 of nprocs = 3
 hello from hpc-200-06-17.cr.cnaf.infn.it rank=1 of nprocs = 3
```

- The following is the hostfile used above

```
> Cat hosts
  hpc-200-06-06 slots=1
  hpc-200-06-17 slots=1
  hpc-200-06-18 slots=2
```

Exercise: Hello world part 3

- Goal
  - To explore how the hostfile interacts with the nodes we are using for MPI exercises.
- Program
  - Write a program that prints “hello world” to the screen.
  - Modify it to run as an MPI program … with each node printing “hello world”, its rank, and the name of the node.
  - Experiment with hostfile changing the order of nodes in the file and the number of slots per node.

```
#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_Get_processor_name(name,&namLen);
MPI_Finalize();
```

The number of slots is the number of processes to create on a node. You can run multiple processes on a CPU … it actually makes sense to do so sometimes … up to the number of cores on the system.

```
> cat hosts
  hpc-200-06-06 slots=1
  hpc-200-06-17 slots=1
  hpc-200-06-18 slots=2
```

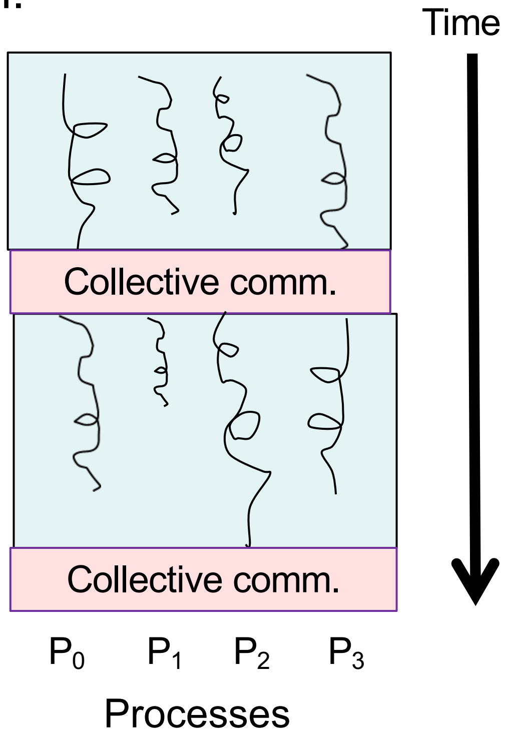
For ESC24, our cluster nodes have two 8 core CPUs with hyperthreading enabled (hence the OS things there are 16 cores)
Use the Linux command lscpu to learn about the CPUs on a node

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

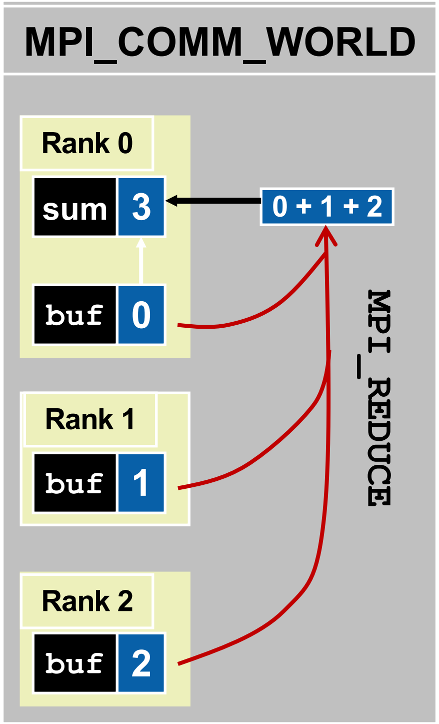
*This is a subset of available MPI types

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

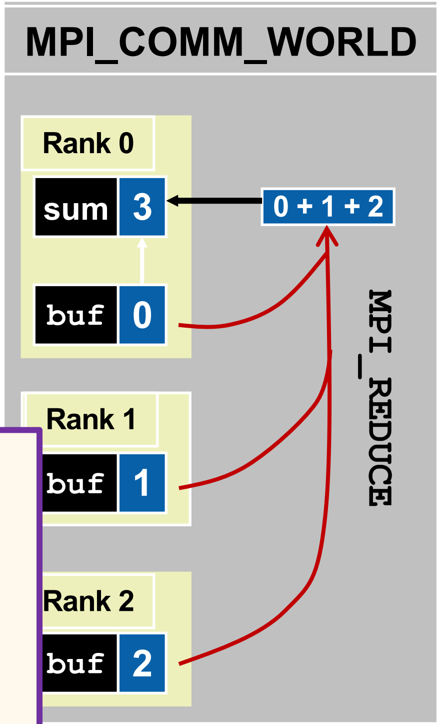
MPI_Reduce() Example

```
#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();
}
```



MPI_Reduce() Example

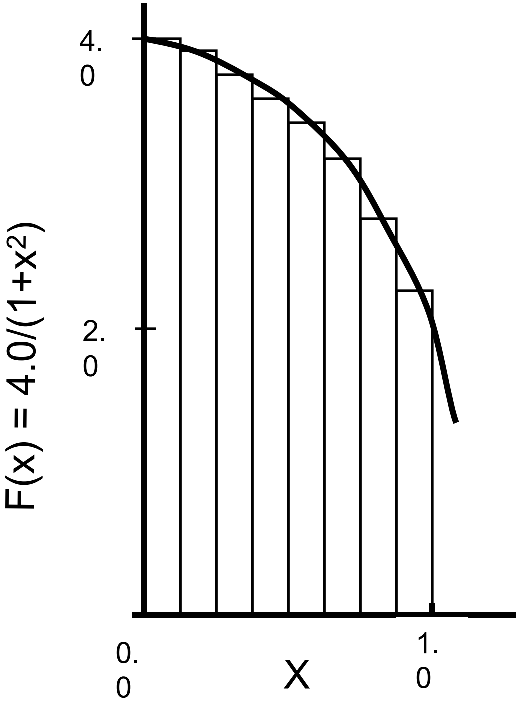
```
#include <mpi.h>

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;
```



C language comments:

- **char*** is a pointer to a collection of characters (a string).
- **char* argv[]** is the same as **char **argv**. They point to a collection of strings
- If you have a variable and you want its address, use the **&** character. C is a *call-by-value* language. If you want to pass updated values through a function argument, you need to pass in the address for that argument, for example **&myrank**

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

```
module load compilers/openmpi-4-1-5_gcc12.3
module load compilers/gcc-12.3_sl7
```

- 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);
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

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.

Exercise: Pi Program with MPI_Wtime()

- Goal
  - Time your Bulk Synchronous, SPMD program
- Program
  - Start with your parallel “pi program” and use MPI_Wtime() to 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);
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=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);
        if(my_id == 0) printf(“ runtime = %lf\n”,MPI_Wtime()-init_time);
}
```

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=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);
        if(my_id == 0) printf(“ runtime = %lf\n”,MPI_Wtime()-init_time);
}
```

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.

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=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);
        if(my_id == 0) printf(" runtime = %lf\n",MPI_Wtime()-init_time);
}
```

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

Is this a dependable way to get an elapsed time?

What if instead of a laptop, we are starting processes across a large cluster?   Is this time reliable?

*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.

Synchronization in MPI

- Synchronization … establishing ordering constraints among concurrent processes so we can establish happens-before relations.

- As we will see later … the semantics of how messages are passed includes synchronization properties.

- For a stand-alone synchronization construct, we can use a barrier (all processes in the group associated with comm arrive before any proceed):
  - **int MPI_Barrier(MPI_Comm comm)**

Synchronization in MPI

- Synchronization … establishing ordering constraints among concurrent processes so we can establish happens-before relations.

- As we will see later … the semantics of how messages are passed includes synchronization properties.

- For a stand-alone synchronization construct, we can use a barrier (all processes in the group associated with comm arrive before any proceed):
  - **int MPI_Barrier(MPI_Comm comm)**

What is this int for? All MPI routines other than the timing routines return an int error code. Equals MPI_SUCCESS when everything is OK, other values specific to routines when errors occur. It’s common to just ignore this output (which is bad practice, but ”we all” do it.

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) ;
        MPI_Barrier(MPI_COMM_WORLD);
        if(my_id ==0) double init_time = MPI_Wtime();
        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);
        if(my_id == 0) printf(“ runtime = %lf\n”,MPI_Wtime()-init_time);
}
```

Use a barrier to make sure all processes have started-up before we start timing the computation

We don’t need a barrier here since collective communication implies a barrier

Timing without a barrier

- Another option … forget the barrier. Collect times for all processes and report min, max and average. This is easy to do using the operations available for use in MPI_Reduce.

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

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

- Plus, knowing min, max and average gives you information about how well balanced the load it. It’s much more informative than a single number with barrier.

Exercise: Explore timing MPI programs with the Pi program

- Goal
  - To work with a number of reduction operators and use results to access load balancing.
- Program
  - Use MPI_Wtime(), MPI_Barrier() and other methods explore timing for the pi program.

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

```
#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();
int MPI_Barrier(MPI_COMM_WORLD);
MPI_Finalize();
```

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

Programming GPUs with OpenMP

- Get the repository:
  - https://github.com/tgmattso/ParProgForPhys.git
- This includes lecture-slides and exercises for my course on GPU programming with OpenMP

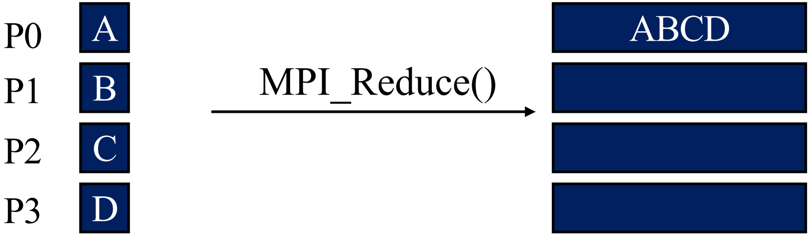
Pi program … return max time

```
#include <mpi.h>
void main (int argc, char *argv[])
{       int i, my_id, numprocs;  double x, pi, step, sum = 0.0, mxtime=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) ;
         MPI_Barrier(MPI_COMM_WORLD);
         double init_time = MPI_Wtime();
         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);
        double wtime = MPI_Wtime()-init_time
         MPI_Reduce(&wtime, &mxtime, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
         if(my_id == 0) printf(“ maximum time = %lf”,mxtime);
}
```

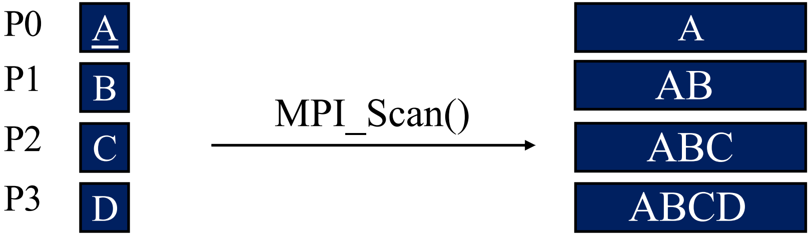
MPI defines a rich set of Collective operations

Collective Computations

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



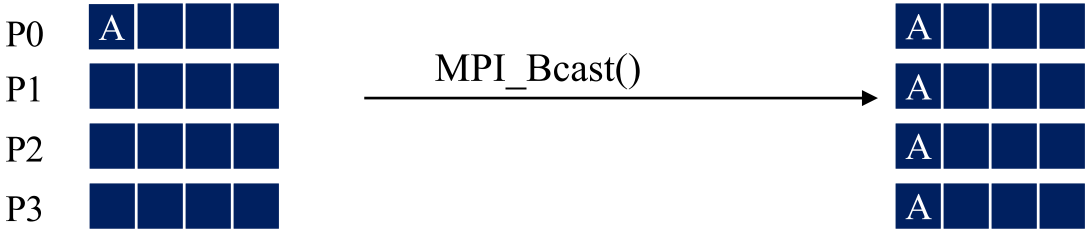
Scan: Take values on each P and combine them with a scan operation and spread the scan array out among all P.



```
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)
```

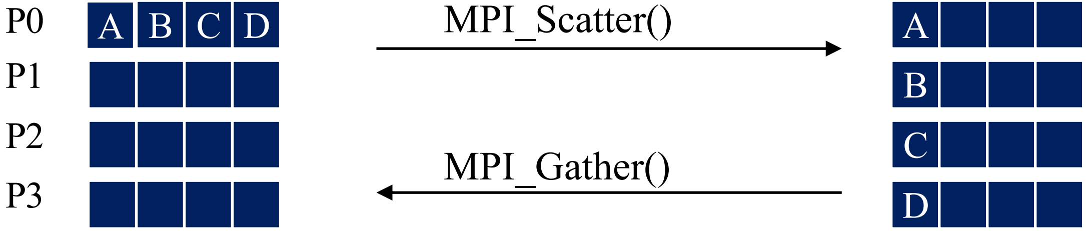
Collective Data Movement

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

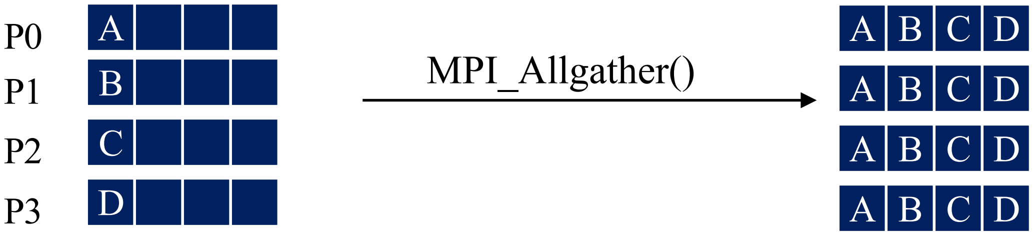
Gather values from P1, P2, and P3 into an array on P0 (the root)



```
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

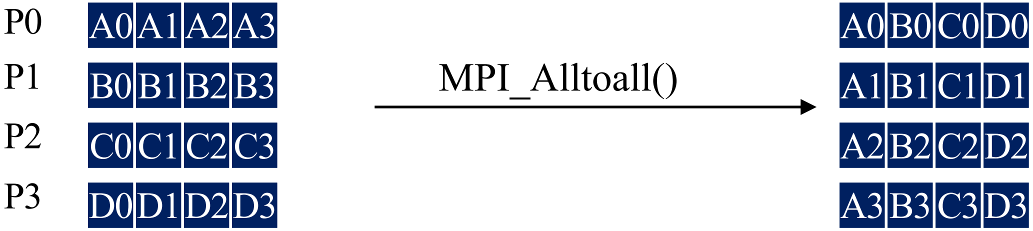
Gather a chunk from each P and put it into a single array. Each P gets a copy of the resulting array.



P0	A			
P1	B			
P2	C			
P3	D			

A	B	C	D
A	B	C	D
A	B	C	D
A	B	C	D

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



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

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

```
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.

Collective Communication: Theory, Practice, and Experience FLAME Working Note #22
Ernie Chan, Marcel Heimlich, Avi Purkayastha, Robert van de Geijn,
September 11, 2006, https://www.cs.utexas.edu/~flame/pubs/InterCol_TR.pdf

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:

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

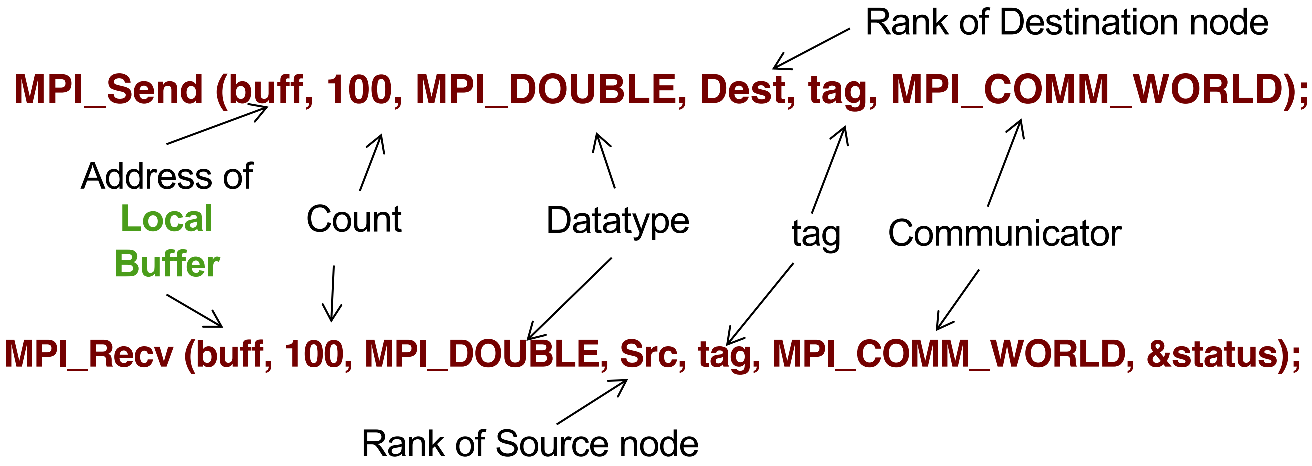
Network fixed costs plus overheads

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



Rank of Destination node

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

Address of **Local Buffer**

Count

Datatype

tag

Communicator

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 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. V
the received message. The most common usage is to find out how many items were in t

```
MPI_Status MyStat;        int count;      float buff[4];
int ierr = MPI_Recv(buf, 4, MPI_FLOAT, 2, 0, MPI_COMM_WORLD, &MyStat);   // receive 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.

What about C++?

- MPI used to have a C++ interface.
- The MPI forum, however, deprecated that interface.
  - It did not add much value compared to using the C interface in C++.
  - Supporting another language in the MPI specification adds a huge amount of work.

- The major challenge in moving between C++ and C is how to handle buffers when your arrays use std::vector or std::array.

- The following should work* (I haven’t fully tested these options):

```
vector<float> a(25);
MPI_Send(a.data(), 25, MPI_FLOAT, …)
MPI_Send( &a[0],  25, MPI_FLOAT, …)
MPI_Send( &a.front(), 25, MPI_FLOAT, …)
```

- You cannot send from an iterator …. Let recv determine size/capacity.

*Source: Victor Eijkhout of TACC. https://theartofhpc.com/

Exercise: Ping-Pong Program

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

Network fixed costs plus overheads

Network asymptotic bytes per second

- Goal
  - Measure the time to communicate a small message between nodes. Compare on-node vs between-node latencies.
- Program
  - Write a program to bounce a messages (a single value) between a pair of processes. Bounce the message back and forth multiple times and report the average one-way communication time. Then modify it to handle larger messages and explore communication time as a function of message size.

```
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_IGNORE
```

```
#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

```c
#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();
}
```

Ping Pong for different message sizes … but first a bit of C

- Input parameters from the command line (so you don’t need to recompile for each case):

```
int main(int argc, char **argv)
{
     if (argc == 3){
        int msg_size = atoi(*++argv);
        int num_pings = atoi(*++argv);
     }
     else{
        int msg_size = 1;
        int num_pings = 10;
     }
```

Argc → number of command line arguments
**argv →Pointer to a set of strings

Argc == 3 → the executable Plus two args

*++argv → increment to point to next string

atoi() →converts a string to an int

Define a default case for when skipped command line are omitted

- Allocate memory and initialize buffer (i.e., a dynamic array of doubles)

```
double *msg = (double*)malloc(msg_size*sizeof(double));
for(int i; i<msg_size; i++) msg[i] = (double) i;
```

Malloc allocates memory as a void*. Cast to the desired type

Msg is a pointer but we treat it like an array

Working with command line arguments

- You typically need to do some processing of command line arguments before proceeding with a computation.

- The common pattern is to pick a node to do that work and then broadcast the results to the other nodes before proceeding.

```
#include <mpi.h>
int main(int argc, char **argv)  {
   int rank, size, param;
   double t0;
   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   if(my_ID == 0){
     if (argc == 2){
           param = atoi(*++argv);
           if(param%2 == 0) param += 1; // if odd, make param even
     else {
          param = 5;
     }
    MPI_Bcast (&param, 1, MPI_INT, 0, MPI_COMM_WORLD);
          // now do the computation (not shown).
   MPI_Finalize();
}
```

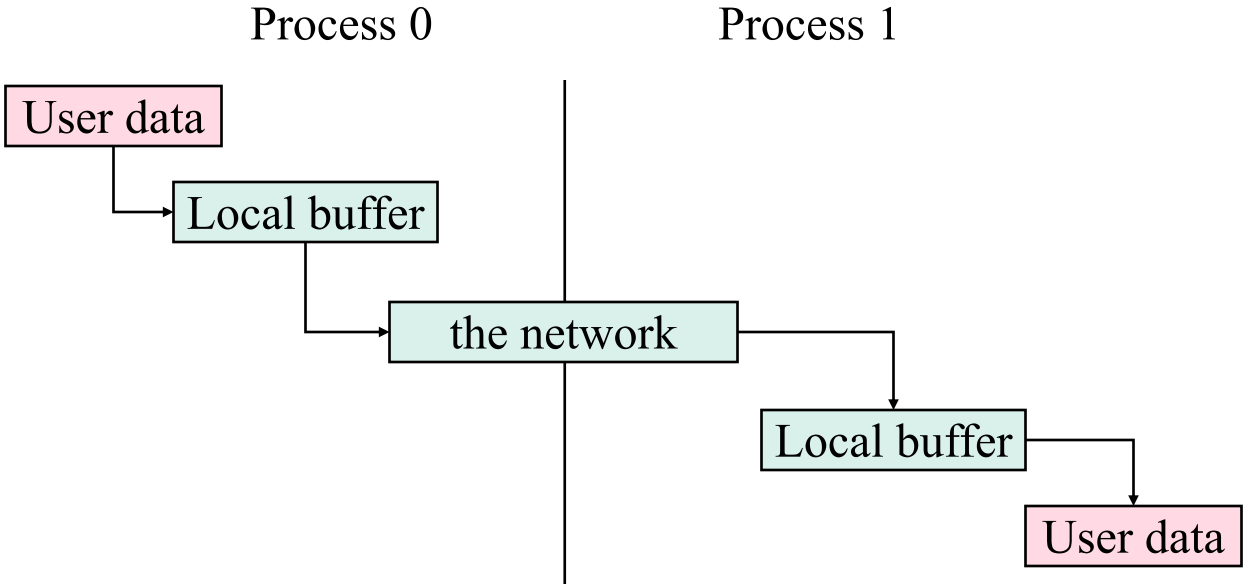
Broadcast one value of MPI_INT from node 0 to all other nodes

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

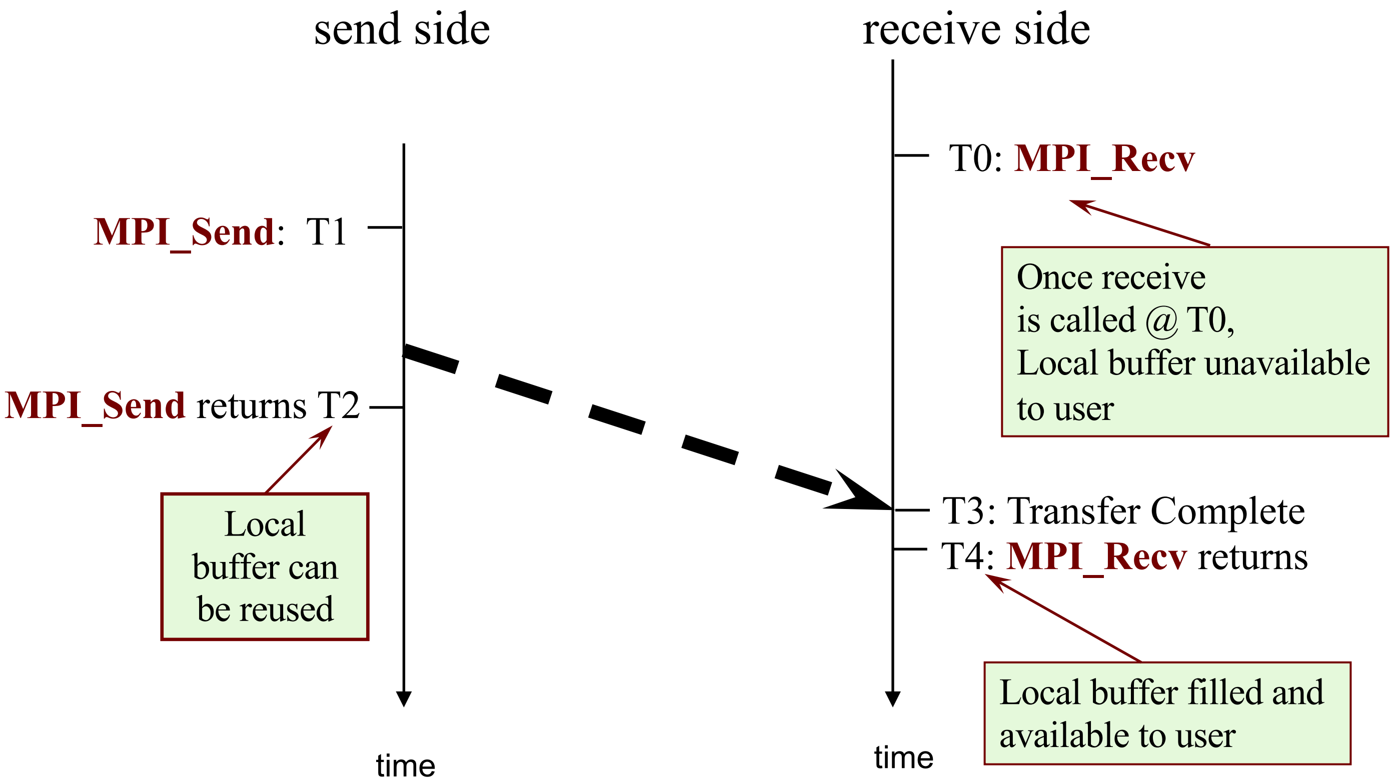
- Message passing is straightforward, but there are subtleties
  - Buffering and deadlock
  - Deterministic execution
  - Performance
- When you send data, where does it go? The following is the typical flow:



Derived from slides provided by Bill Gropp of UIUC

Blocking Send-Receive Timing Diagram

(Receive before Send)



send side

receive side

T0: **MPI_Recv**

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

MPI_Send: T1

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.

Exercise: Ring program

- Start with the basic ring program we provide. Compile as:

```
> mpicc ring.c ring_naive.c
```

- Study the code (ring.c and ring_naive.c) and note how I manage the computation of where the message goes to and where it comes from for each node.

- Run it for a range of message sizes and notes what happens for large messages.

```
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);
```

Sources of Deadlocks

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

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

- Use a collective “swap” so buffers created when the communication operation is posted:

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

Slide source: based on slides from Bill Gropp, UIUC

Non-Blocking Communication

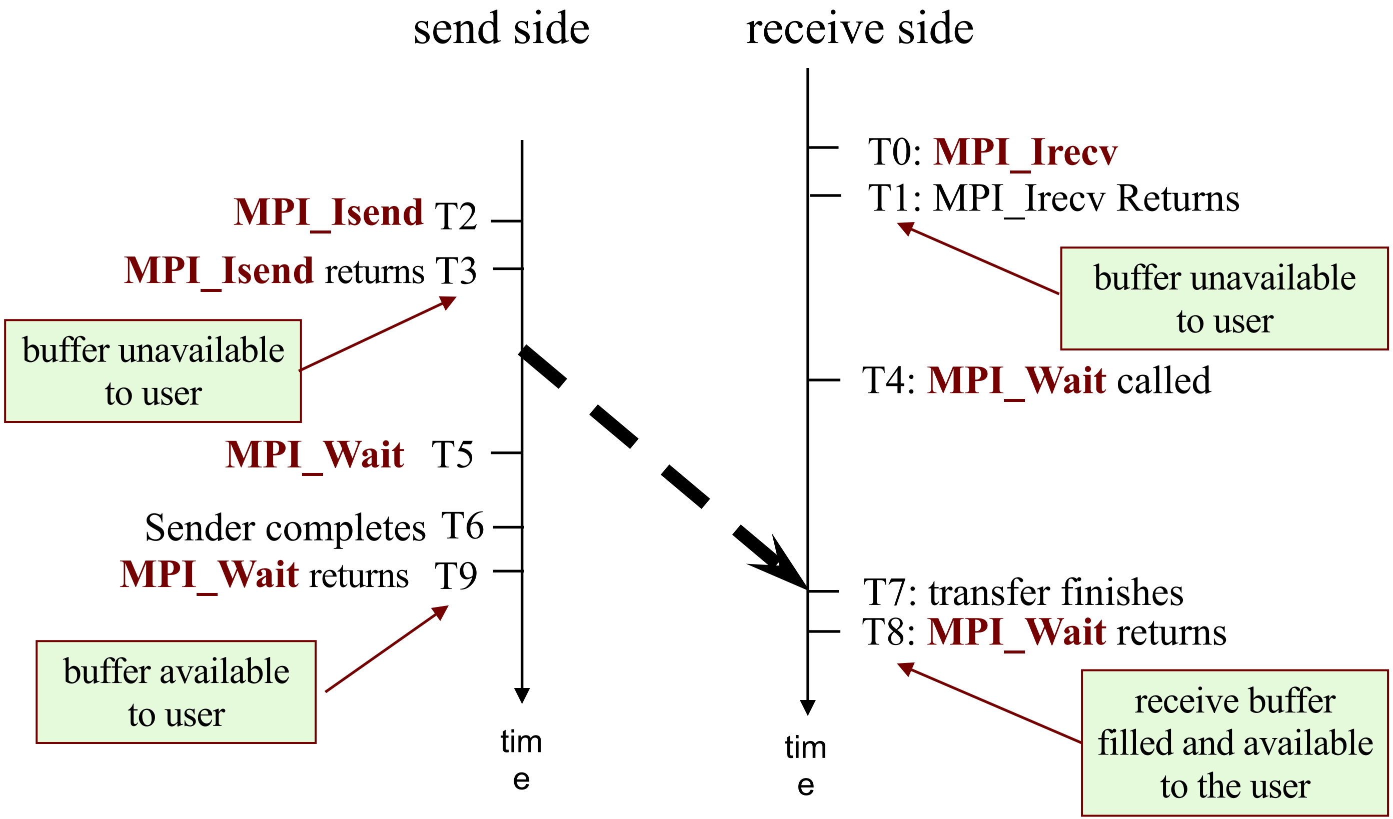
- Non-blocking operations return immediately and pass "request handles" that can be waited on and queried
  - **MPI_Isend(start, count, datatype, dest, tag, comm, request)**
  - **MPI_Irecv(start, count, datatype, src, tag, comm, request)**
  - **MPI_Wait(request, status)**
- One can also test without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



Exercise: Ring program

- Start with the basic ring program you wrote. .
  - Using blocking Send/Recv, 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).
- Make it more stable for large messages by:
  - Split-phase … half 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_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);
```

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

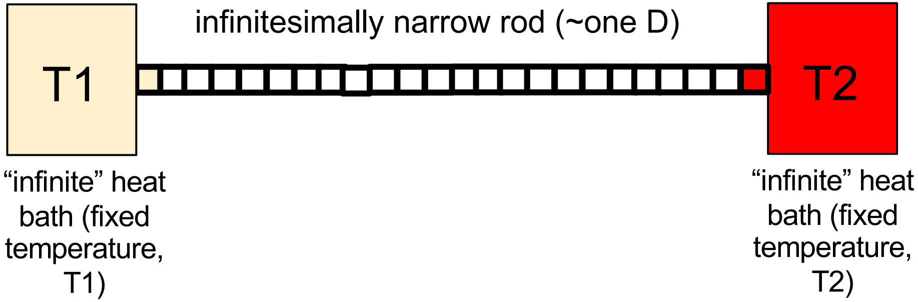
$$u_j^{n+1} = (1-2r)u_j^n + ru_{j-1}^n + ru_{j+1}^n \qquad r = k/h^2$$

- 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

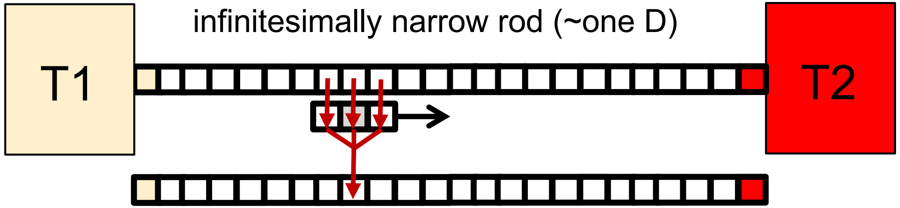


infinitesimally narrow rod (~one D)

"infinite" heat bath (fixed temperature, T1)

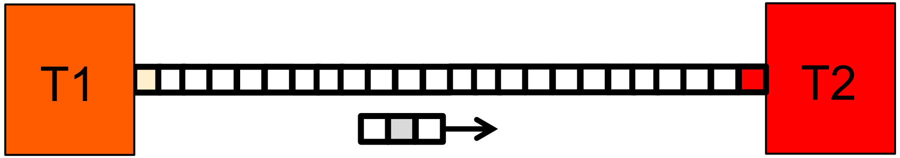
"infinite" heat bath (fixed temperature, T2)

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

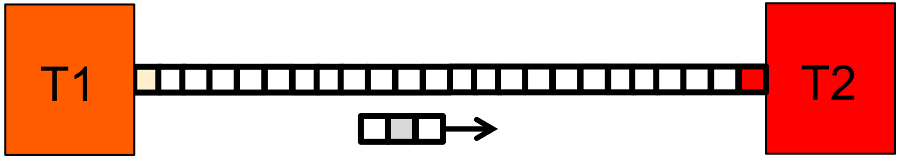


```
int main()
{
   double *u   = malloc (sizeof(double) * (N));
   double *up1 = malloc (sizeof(double) * (N));

   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?

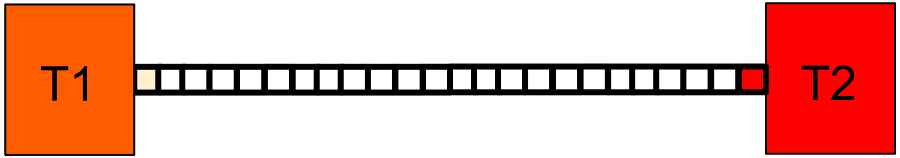
Exercise: Parallel heat diffusion

- Goal
  - Parallelize the heat diffusion code (MPI_Exercises/heat-eqn-seq.c) with OpenMP … should be a quick and easy way to familiarize yourself with the code.
  - As you do this, think about how you might parallelize this with MPI

```
#pragma omp parallel
#pragma omp for
#pragma omp critical
#pragma omp single
#pragma omp barrier
int omp_get_num_threads();
int omp_get_thread_num();
```

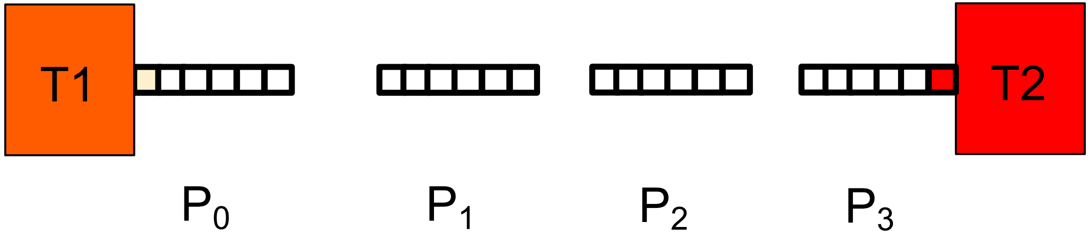
Heat Diffusion equation

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



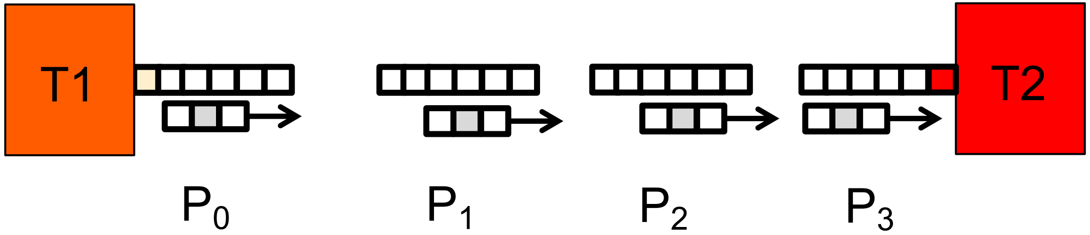
Heat Diffusion equation

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



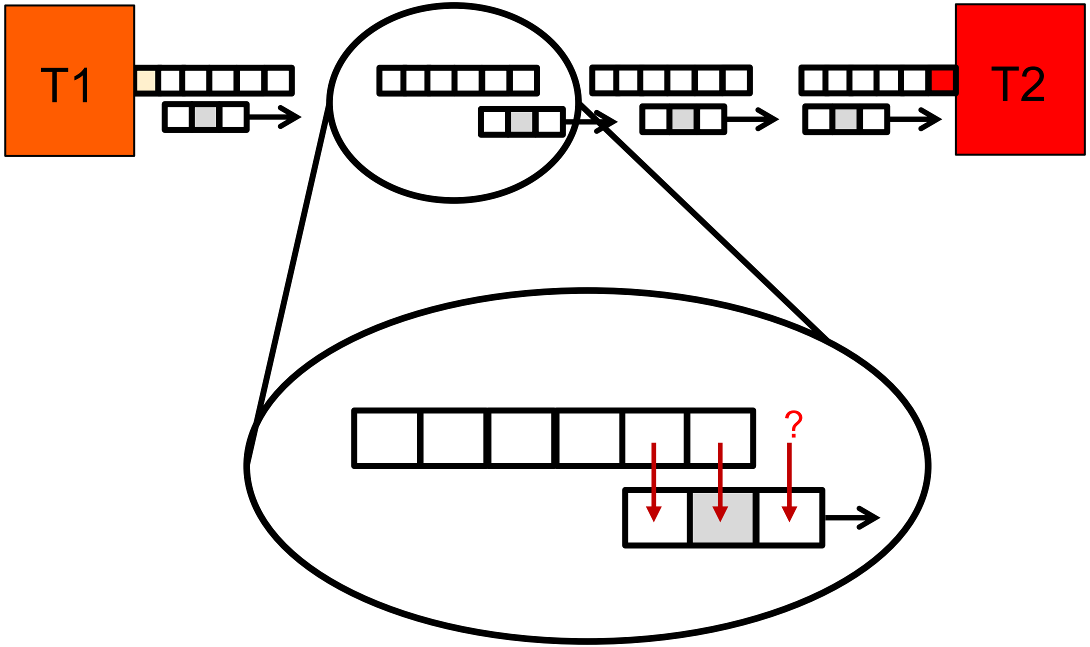
Heat Diffusion equation

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



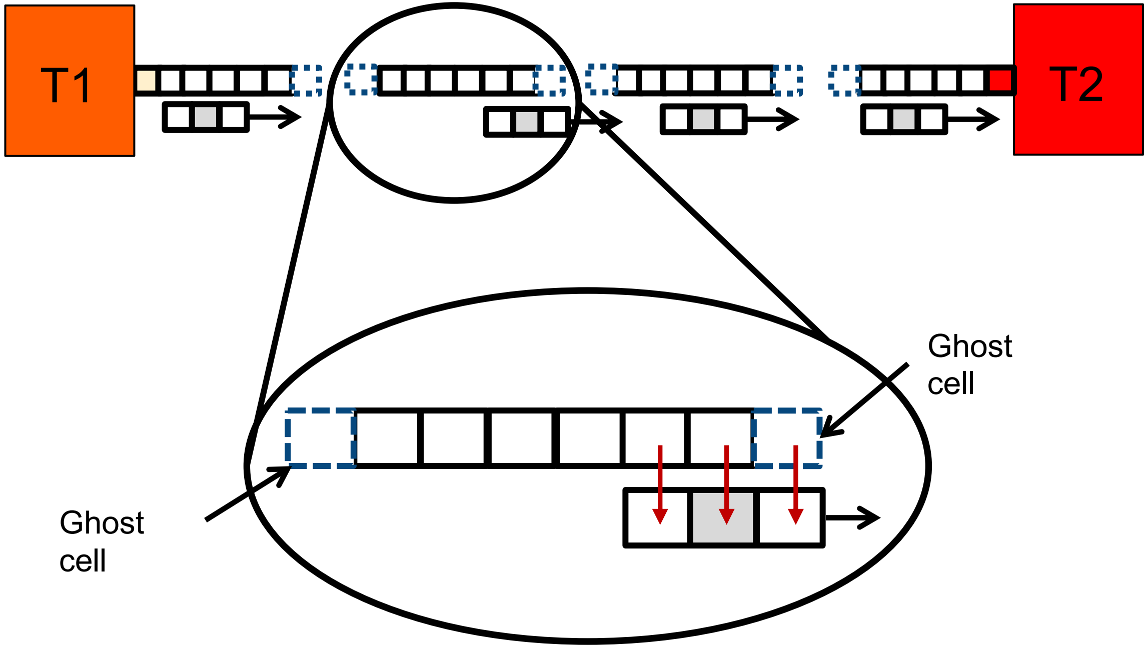
Heat Diffusion equation

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



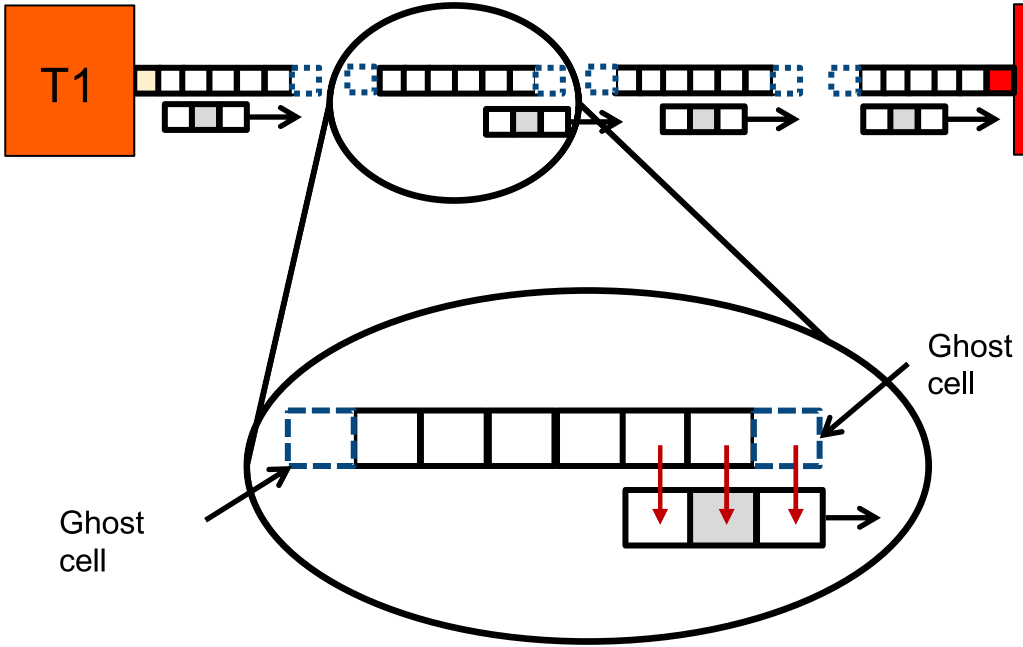
Heat Diffusion equation

- 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

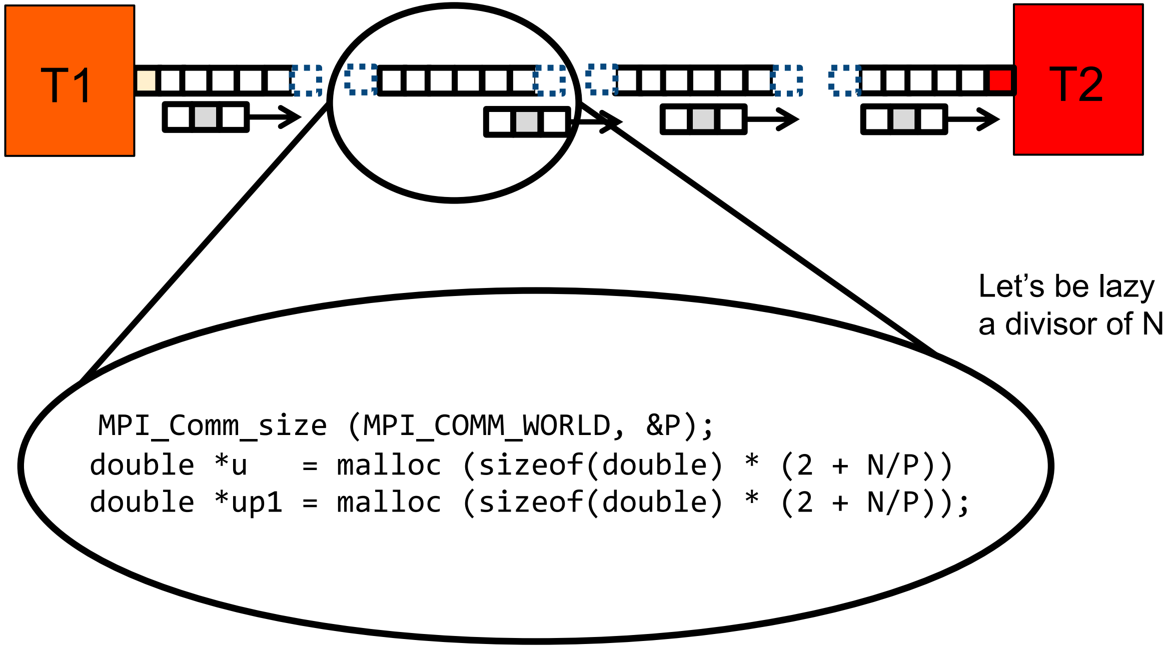
- We add ghost cells to the ends of each chunk, update them with the required values from neighbor chunks at each time step … hence giving the stencil everything it needs on any given chunk to update all of its values.



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.

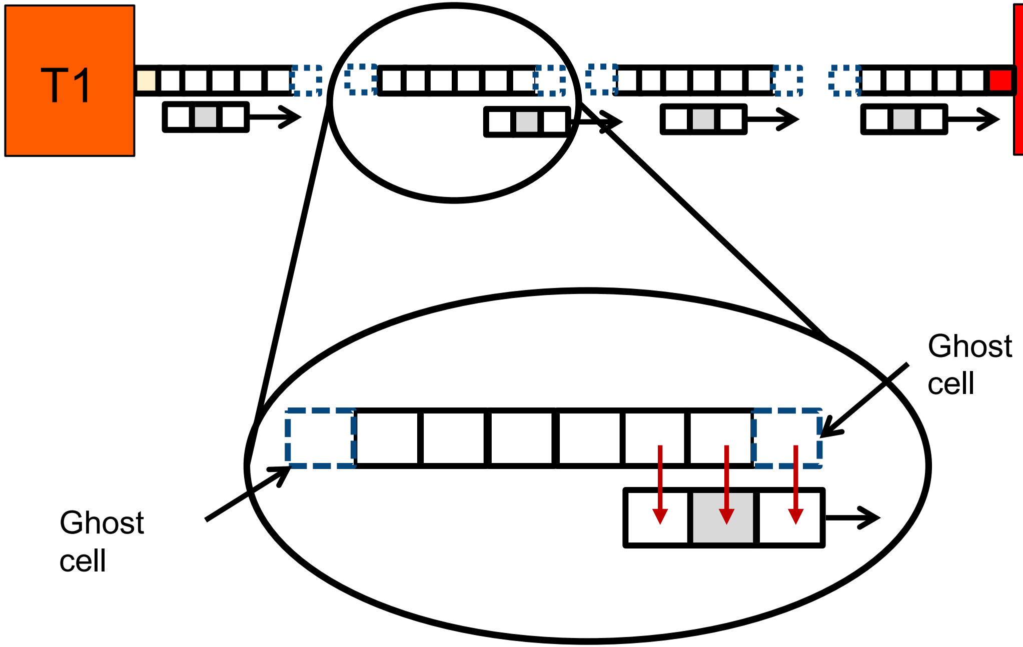


```
 MPI_Comm_size (MPI_COMM_WORLD, &P);
double *u   = malloc (sizeof(double) * (2 + N/P))
double *up1 = malloc (sizeof(double) * (2 + N/P));
```

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;
```

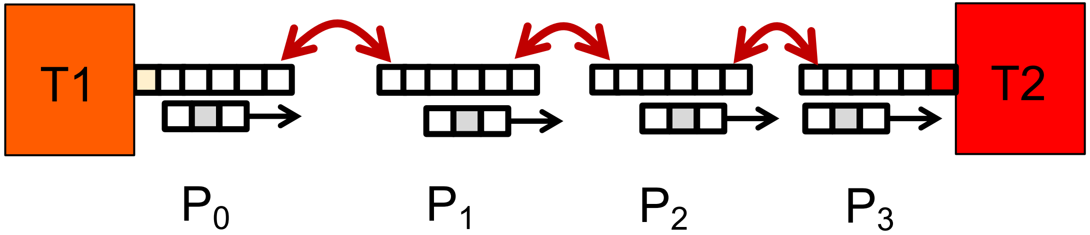
Update array values using local data and values from ghost cells.

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’

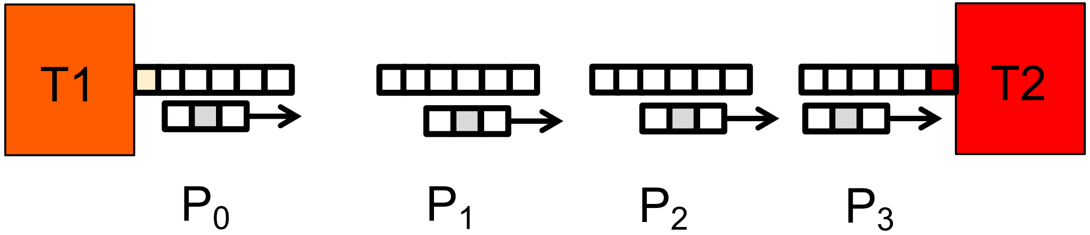
```
  if (myID != P-1) MPI_Send (&u[N/P], 1, MPI_DOUBLE, myID+1, 0, MPI_COMM_WORLD);
```

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.



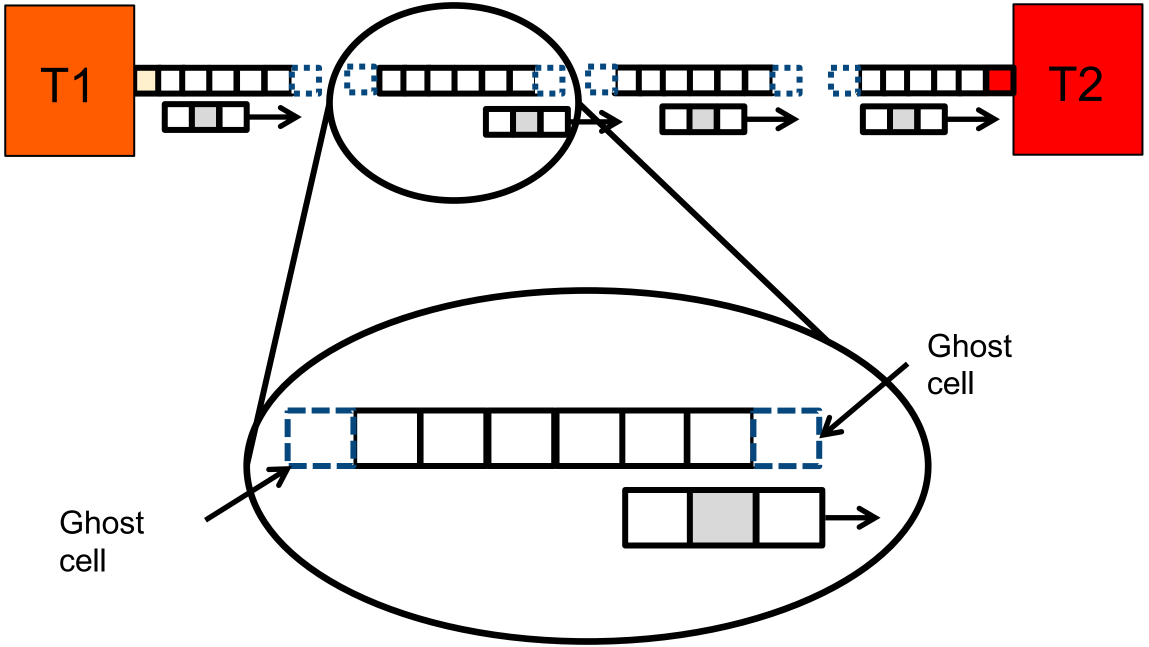
We now put all the pieces together for the full program

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.



Communicating boundary data

- Communicating boundary data was ugly and error prone:

```
 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);
```

- The constant MPI_PROC_NULL when used as a to/from parameter in a message passing function causes the function to return with MPI_SUCCESS as soon as it can.

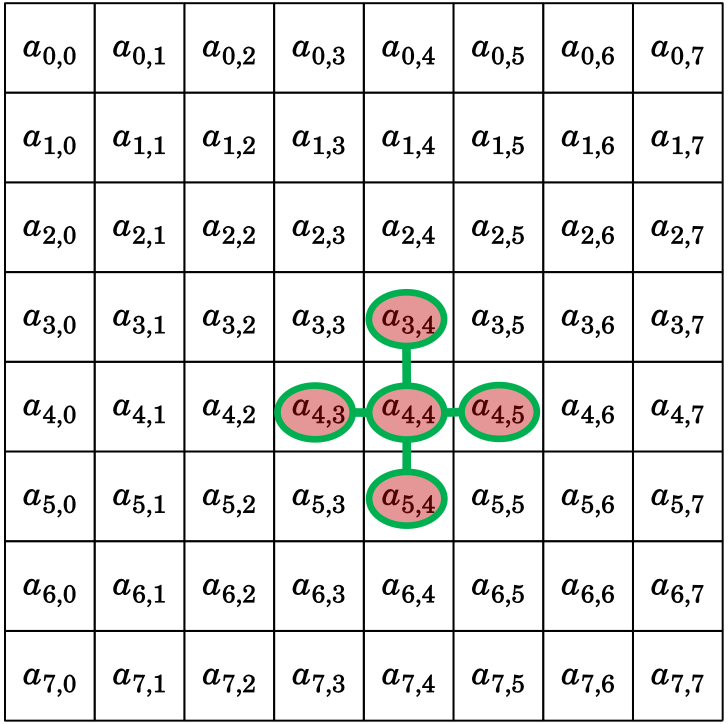
```
MPI_Send (&u[1], 1, MPI_DOUBLE, MPI_PROC_NULL, 0, MPI_COMM_WORLD);
```

Exercise: MPI heat diffusion

- Goal
  - Make the provided code, heat-eqn-mpi.c, simpler and less error prone

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 →



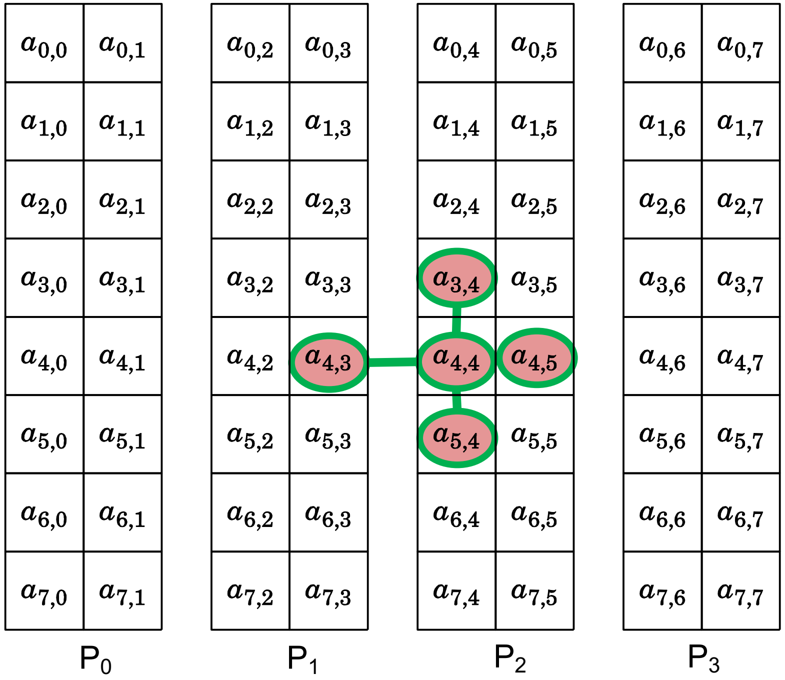
$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$	$a_{0,4}$	$a_{0,5}$	$a_{0,6}$	$a_{0,7}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$	$a_{1,4}$	$a_{1,5}$	$a_{1,6}$	$a_{1,7}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	$a_{2,4}$	$a_{2,5}$	$a_{2,6}$	$a_{2,7}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$	$a_{3,4}$	$a_{3,5}$	$a_{3,6}$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

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

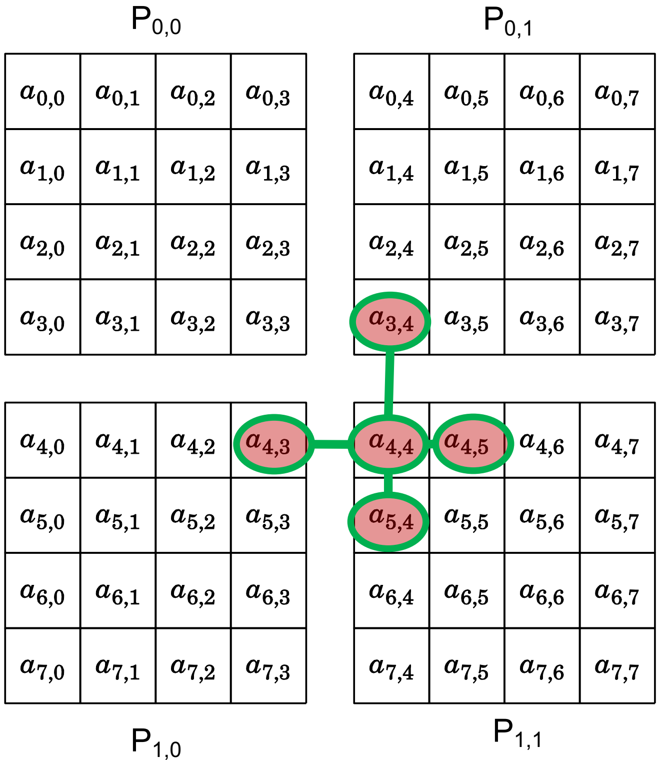
- If we parallelize in both dimensions, then we have $(N/P^{1/2})^2$ elements per processor, and we need to send **$O(N/P^{1/2})$ values** from each processor. Asymptotically better than O(N).

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 $N/P^{1/2}$



$P_{0,0}$

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

$P_{0,1}$

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

$P_{1,0}$

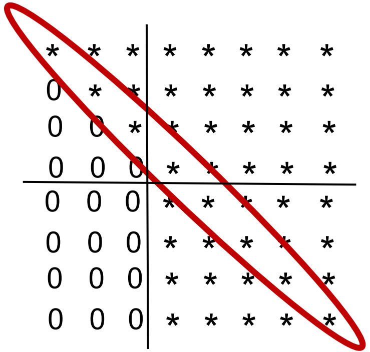
$a_{4,0}$	$a_{4,1}$	$a_{4,2}$	$a_{4,3}$
$a_{5,0}$	$a_{5,1}$	$a_{5,2}$	$a_{5,3}$
$a_{6,0}$	$a_{6,1}$	$a_{6,2}$	$a_{6,3}$
$a_{7,0}$	$a_{7,1}$	$a_{7,2}$	$a_{7,3}$

$P_{1,1}$

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

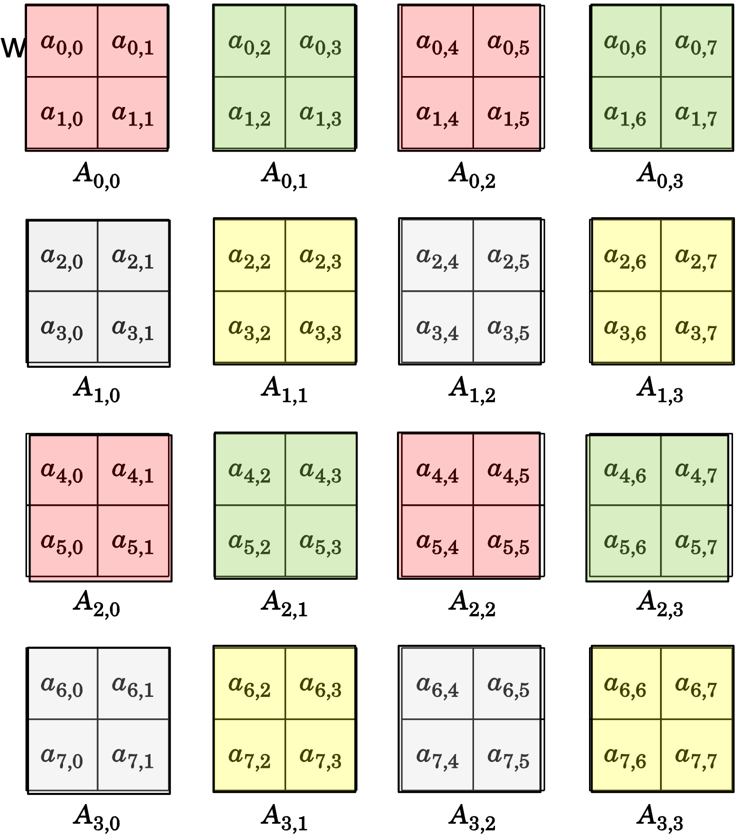
Partitioned Arrays: block cyclic distribution

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



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

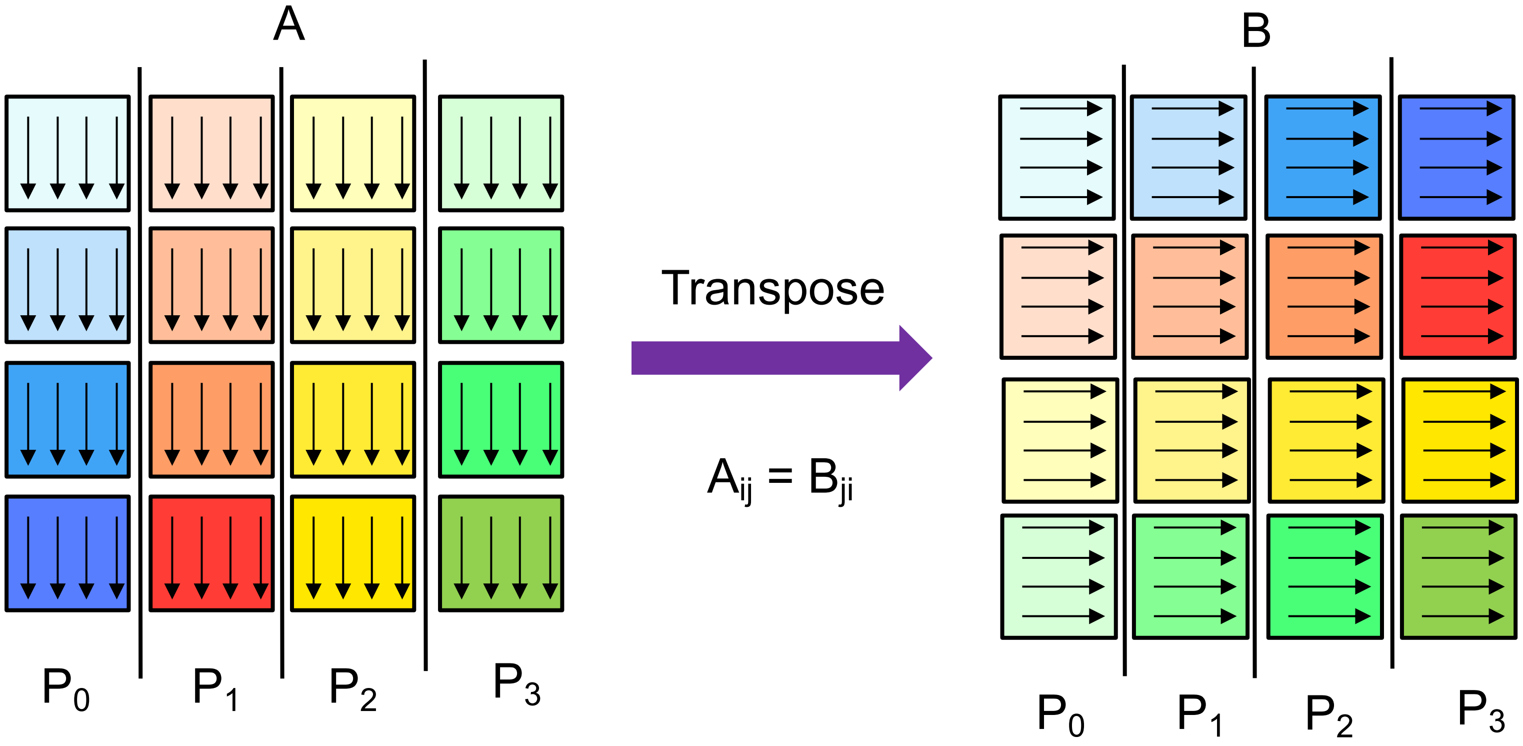
  - Zeros fill in the right lower triangle of the matrix … less work to do.
  - Balance load with cyclic distribution of blocks of A mapped onto a grid of nodes (2x2 in this case … colors show the mapping to nodes).



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

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

Matrix Transposition

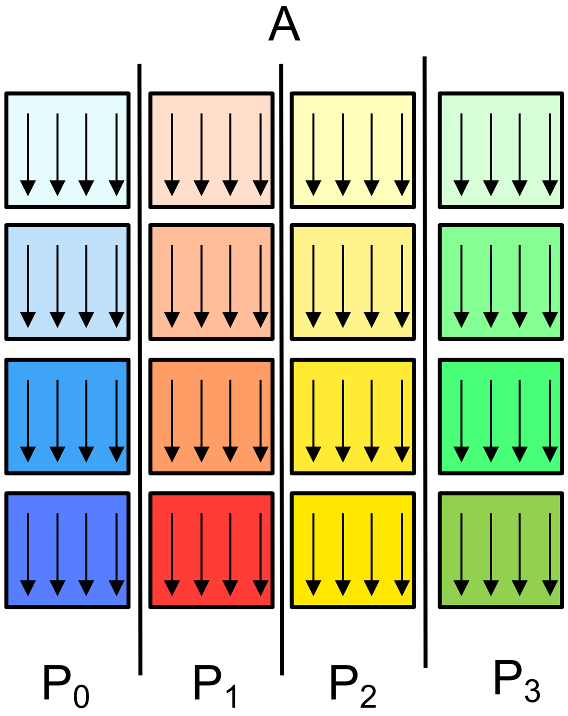
We are going to create a transpose program that uses the SPMD pattern.

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

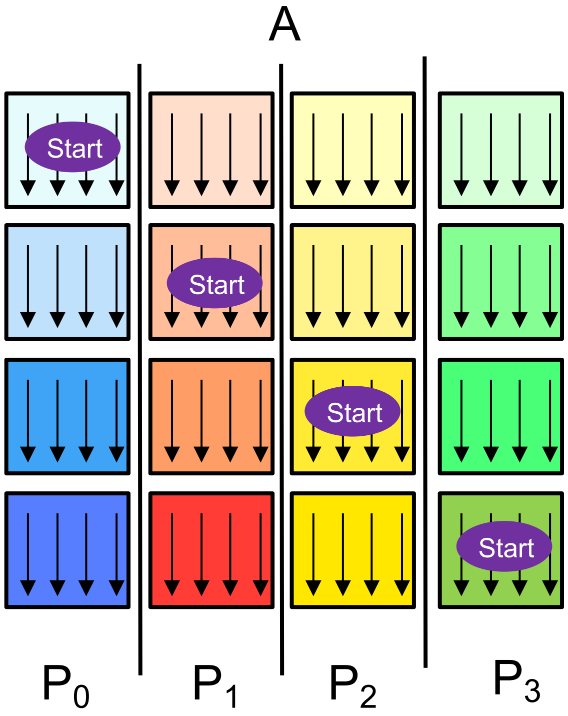
We are going to create a transpose program that uses the SPMD pattern.

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 more than 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

Phase 0 … transpose your diagonal

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

Matrix Transposition

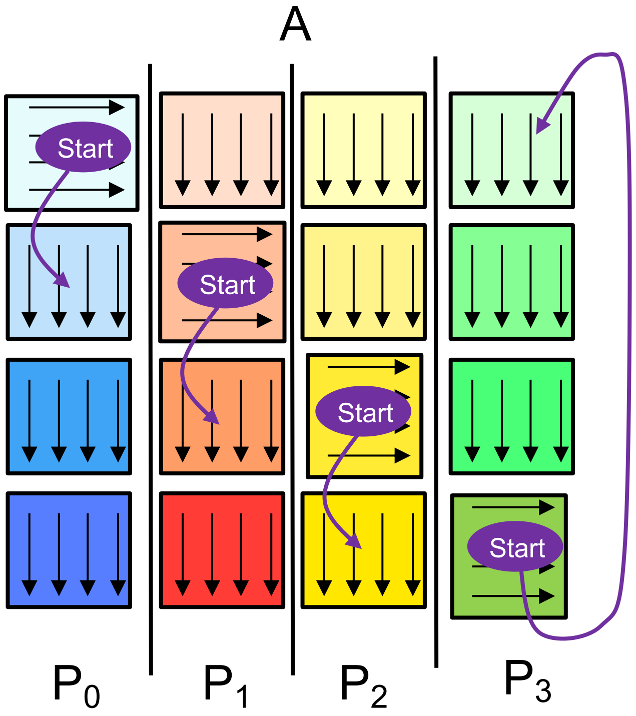
We are going to create a transpose program that uses the SPMD pattern.

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

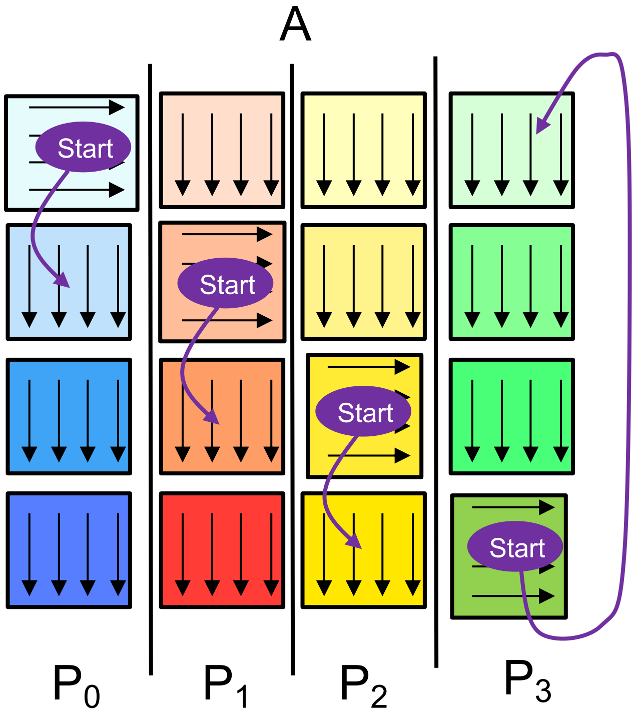
We are going to create a transpose program that uses the SPMD pattern.

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

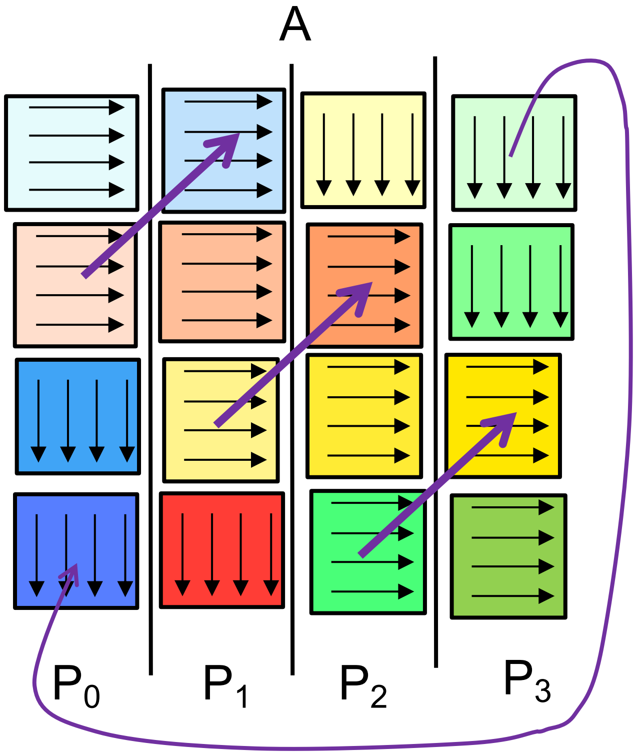
We are going to create a transpose program that uses the SPMD pattern.

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 several trans_*.c functions).

```
> mpicc transpose.c trans_utility.c trans_sendrcv.c
```

- 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 (**what is there now is wrong … 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

The ~~12~~ 10 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, use MPI_Isend().

The ~~12~~ 10 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

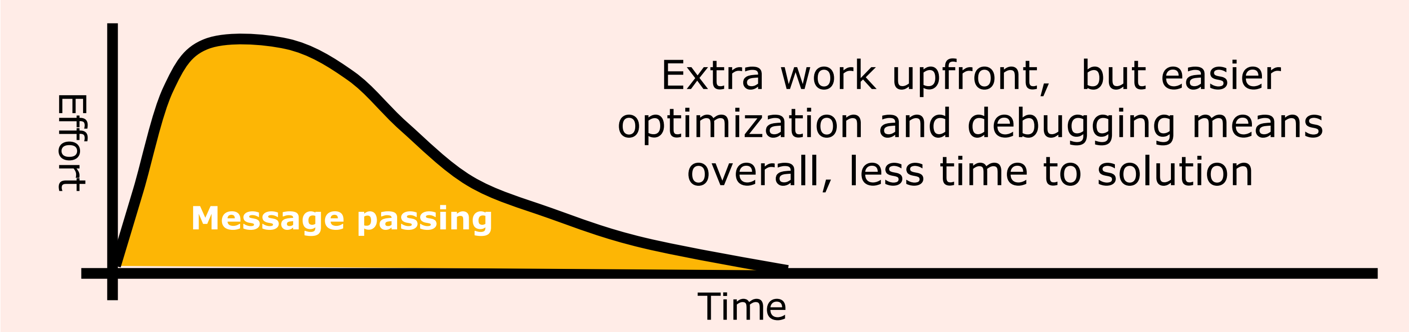
My friends on the MPI forum hate this slide.
These are indeed the functions most people use, but these date back to MPI 1.5 ... The spec is currently at version 5.0

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

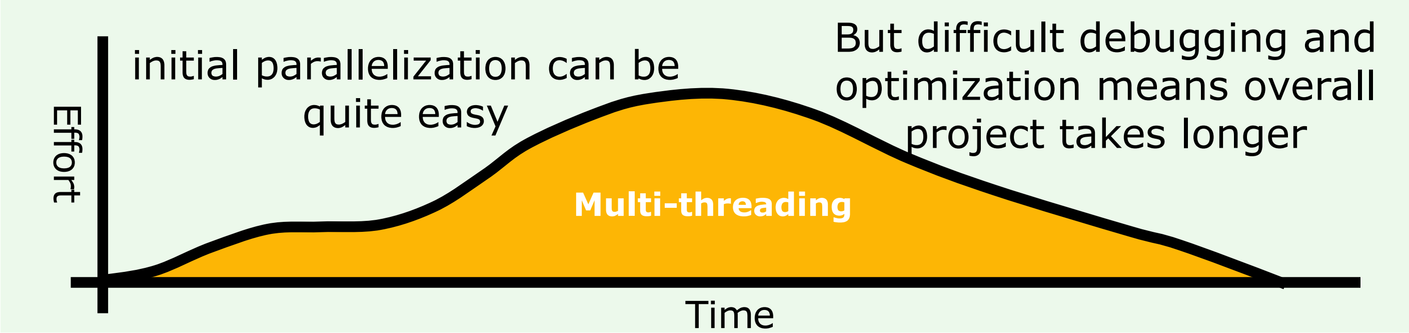
Master these 12 constructs before exploring newer features in MPI.
Then learn about:

- Support for mixing MPI and OpenMP
- Topologies
- One-sided communication
- User defined types
- Shared memory programming within MPI (no need for OpenMP)

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

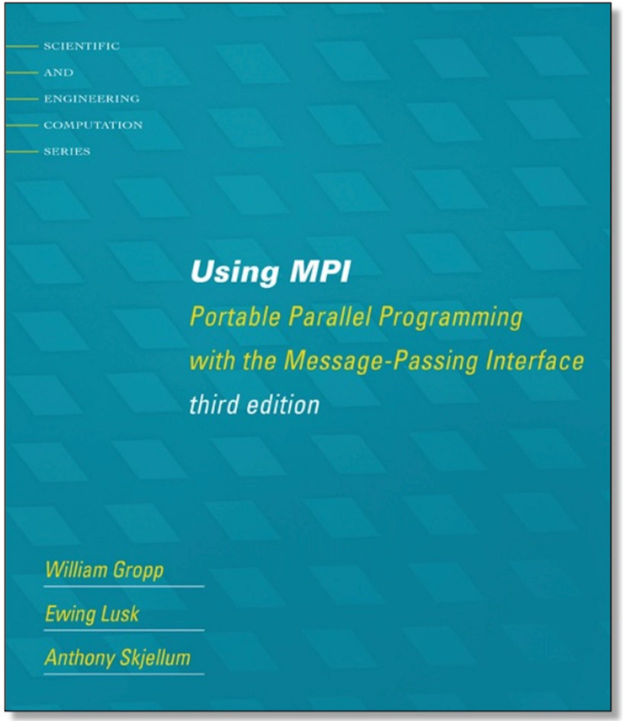
But difficult debugging and optimization means overall project takes longer

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:



SCIENTIFIC AND ENGINEERING COMPUTATION SERIES

Using MPI

Portable Parallel Programming with the Message-Passing Interface

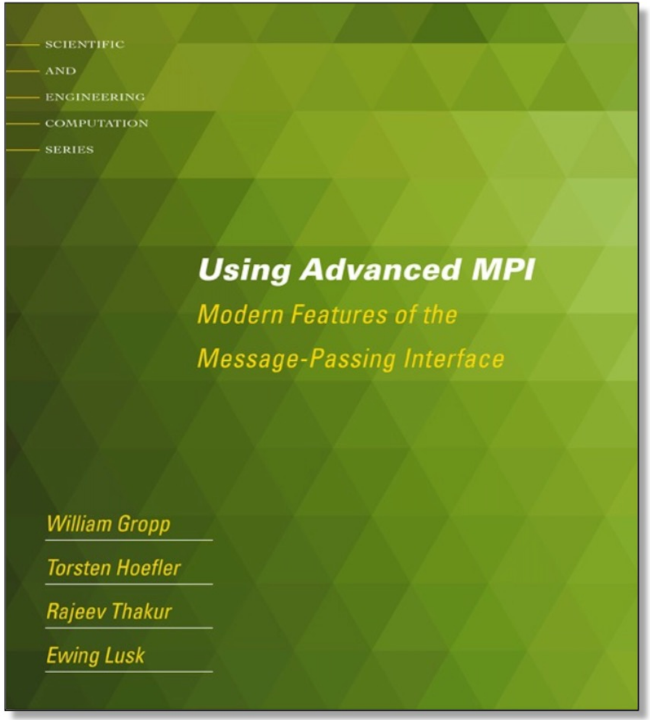
third edition

William Gropp

Ewing Lusk

Anthony Skjellum

Basic MPI



SCIENTIFIC AND ENGINEERING COMPUTATION SERIES

Using Advanced MPI

Modern Features of the Message-Passing Interface

William Gropp

Torsten Hoefler

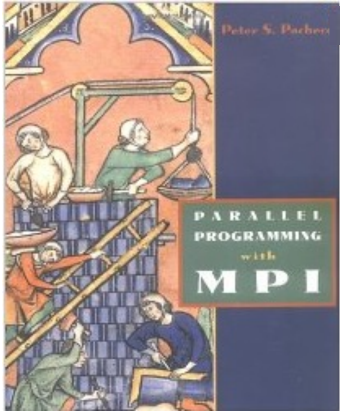
Rajeev Thakur

Ewing Lusk

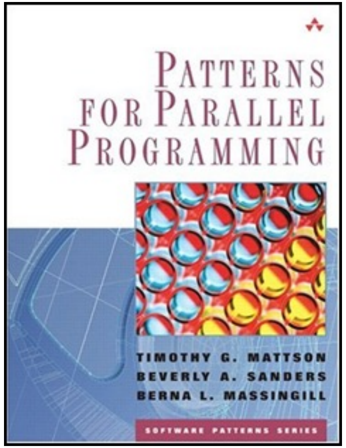
Advanced MPI, including MPI-3

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

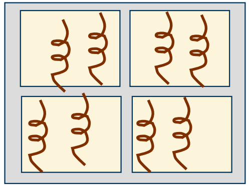


Outline

- 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
- → Wait … there is one more case for us to consider … HPC and the cloud

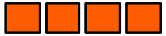
Hardware is diverse … and its only getting worse!!!

Write code with TBB or OpenMP

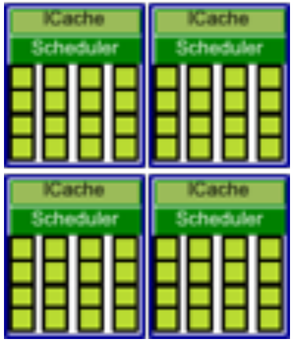


CPU

Work with the compiler to vectorize code



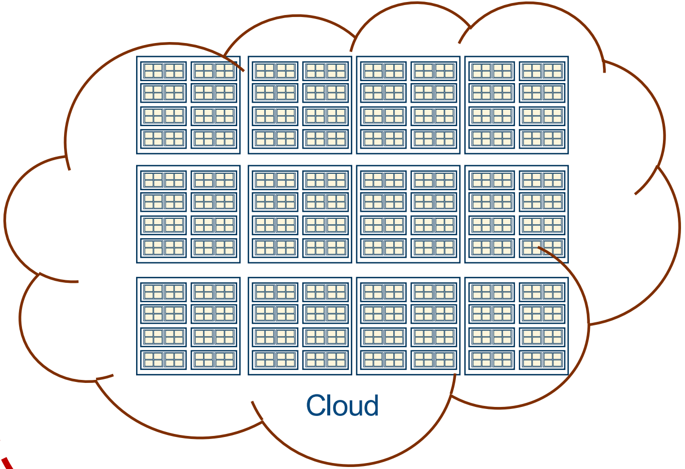
SIMD/Vector



GPU

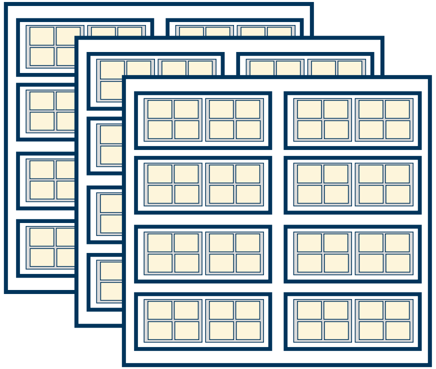
Use a portable API but if you must, use CUDA. It’s all the same model

MPI works in the cloud, but its not really “cloud-like”



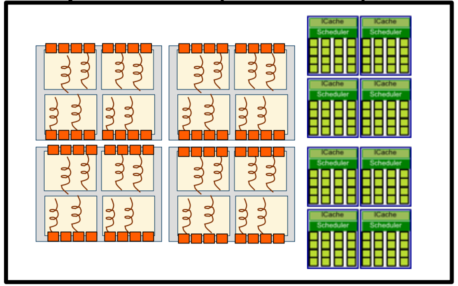
Cloud

All you need is MPI



Cluster

OpenMP lets you “do it all”. Or combine CUDA and OpenMP (or TBB).



Heterogeneous node

The Eight Fallacies of Distributed Computing

(Peter Deutsch of Sun Microsystems, 1994 … item 8 added in 1997 by James Gosling)

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences.

1. The network is reliable
2. Latency is zero
3. Bandwidth is infinite
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is zero
8. The network is homogeneous

https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

The Eight Fallacies of Distributed Computing

(Peter Deutsch of Sun Microsystems, 1994 … item 8 added in 1997 by James Gosling)

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences.

1. The network is reliable
2. Latency is low and fixed
3. Bandwidth is high and fixed
4. The network is secure
5. Topology doesn't change
6. There is one administrator
7. Transport cost is negligible
8. The network is homogeneous

https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

The Eight Fallacies of Distributed Computing

(Peter Deutsch of Sun Microsystems, 1994 … item 8 added in 1997 by James Gosling)

Essentially everyone, when they first build a distributed application, makes the following eight assumptions. All prove to be false in the long run and all cause *big* trouble and *painful* learning experiences.

Cloud

X 1. ~~The network is reliable~~
X 2. ~~Latency is low and fixed~~
X 3. ~~Bandwidth is high and fixed~~
X 4. ~~The network is secure~~
X 5. ~~Topology doesn't change~~
X 6. ~~There is one administrator~~
X 7. ~~Transport cost is negligible~~
X 8. ~~The network is homogeneous~~

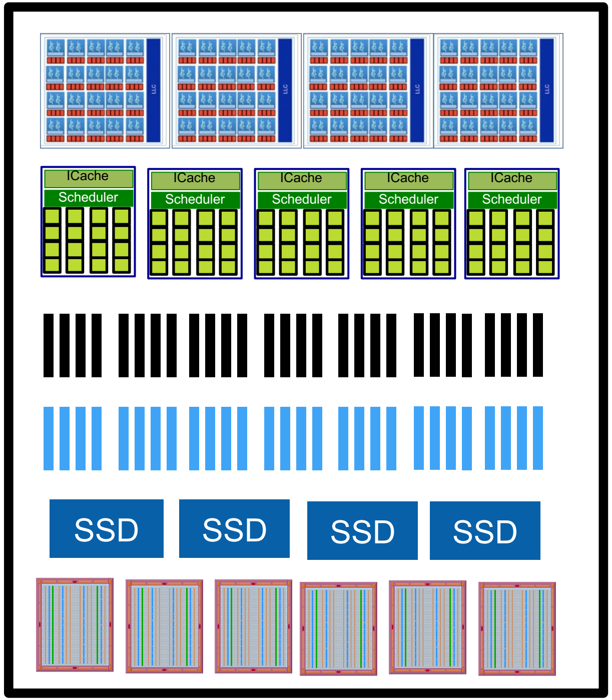
HPC Cluster

✓ 1. The network is reliable
✓ 2. Latency is low and fixed
✓ 3. Bandwidth is high and fixed
✓ 4. The network is secure
✓ 5. Topology doesn't change
✓ 6. There is one administrator
X 7. ~~Transport cost is negligible~~
✓ 8. The network is homogeneous

https://en.wikipedia.org/wiki/Fallacies_of_distributed_computing

Disaggregated Computing for SW Defined Servers (SDS)

Consider a Rack composed of multiple pools



CPU pool

GPU pool

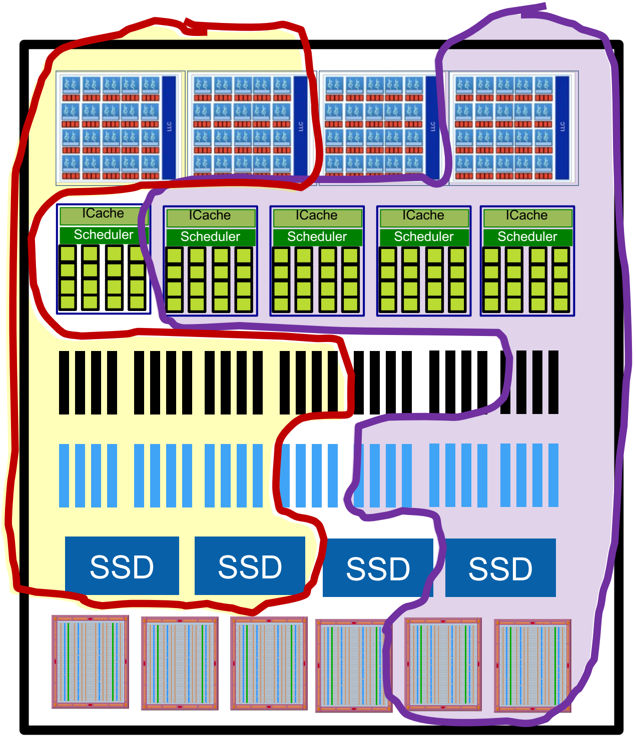
DRAM pool

NVRAM pool

SSD pool

FPGA pool

Dynamically compose across pools to match a software defined server to the workload

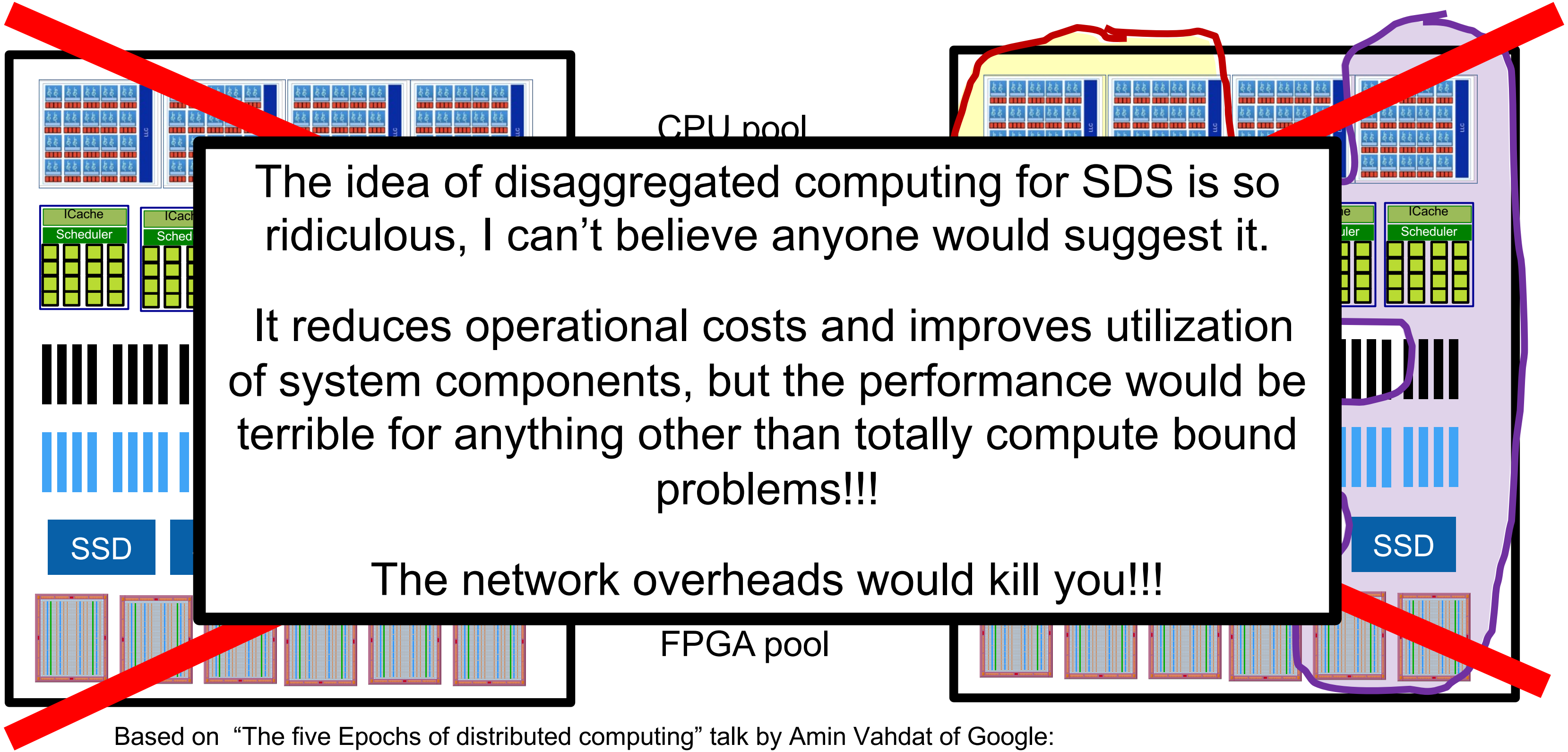


Based on "The five Epochs of distributed computing" talk by Amin Vahdat of Google:

Disaggregated Computing for SW Defined Servers (SDS)

Consider a Rack composed of multiple pools

Dynamically compose across pools to match a software defined server to the workload



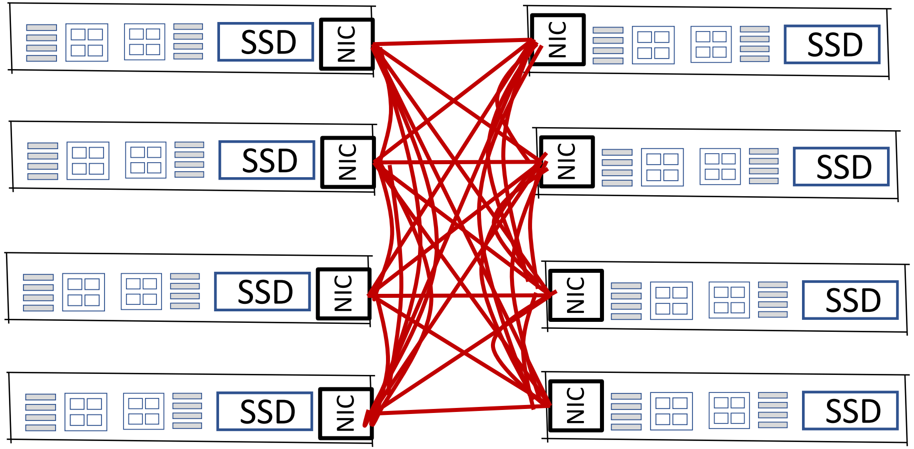
The idea of disaggregated computing for SDS is so ridiculous, I can't believe anyone would suggest it.

It reduces operational costs and improves utilization of system components, but the performance would be terrible for anything other than totally compute bound problems!!!

The network overheads would kill you!!!

Based on "The five Epochs of distributed computing" talk by Amin Vahdat of Google:

Networking technology… replace generic data center network with a cluster of cliques



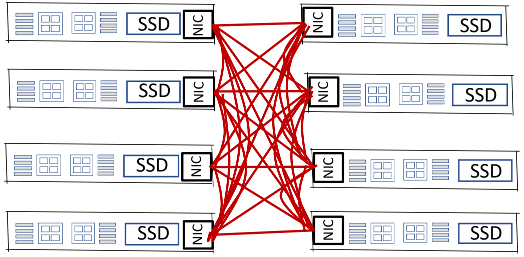
A clique: A graph where every vertex is connected to every other vertex

A Clique: a network of diameter one with O(¼N^2) bisection bandwidth

Combine with next generation optical networks to hit latencies close to DRAM latencies (100 ns)

Latencies every engineer should know …

L1 cache reference 1.5 ns
L2 cache reference 5 ns
Branch misprediction 6 ns
Uncontended mutex lock/unlock 20 ns
L3 cache reference 25 ns
Main memory reference 100 ns
“Far memory”/Fast NVM reference 1,000 ns (1us)
Read 1 MB sequentially from memory 12,000 ns (12 us)
SSD Random Read 100,000 ns (100 us)
Read 1 MB bytes sequentially from SSD 500,000 ns (500 us)
Read 1 MB sequentially from 10Gbps network 1,000,000 ns (1 ms)
Read 1 MB sequentially from disk 10,000,000 ns (10 ms)
Disk seek 10,000,000 ns (10 ms)
Send packet California→Netherlands→California (150 ms)



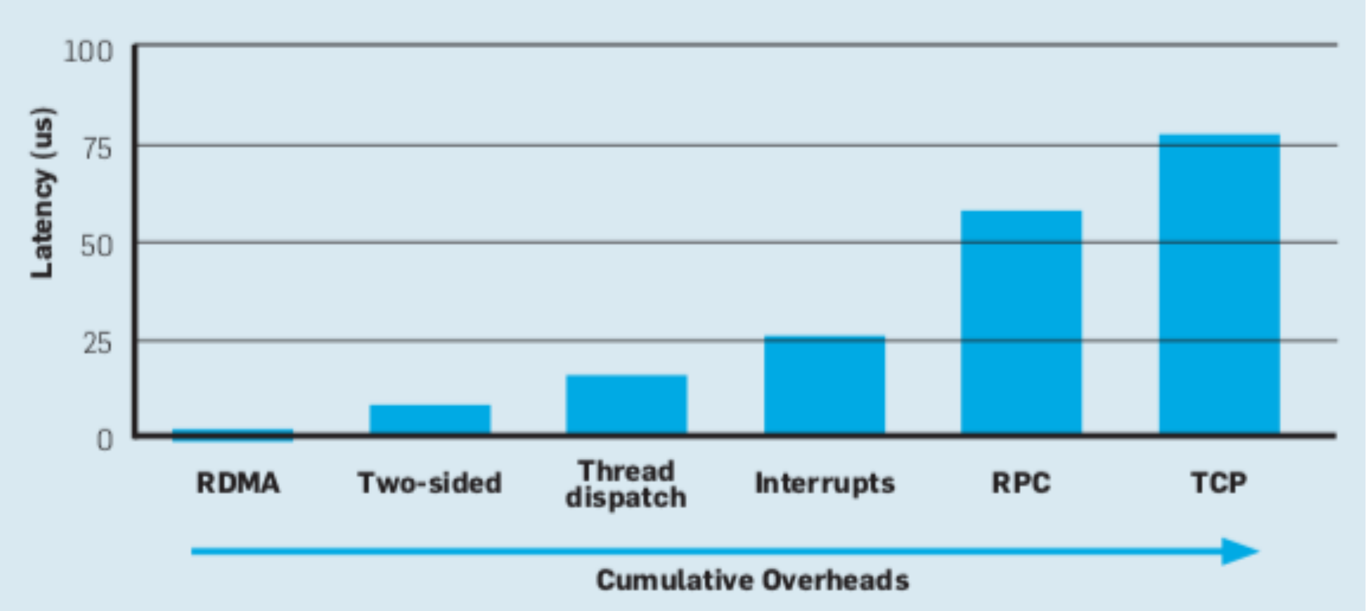
A cluster of nodes with a Clique network topology and low latency optical network…

Yields one hop network latencies on par with DRAM access latencies.

Source: **The Datacenter as a Computer: Designing Warehouse-Scale Machines**, Luiz Andre Barroso, Urs Holzle, Parthasarathy Ranganathan, 3rd edition, Morgan & Claypool, 2019.

Take out the big stuff & you're left with lots of μs overheads

All those SW overheads add up … like bricks that combine to build a networking-wall …
turning a 2 μs network into a 100 μs network…



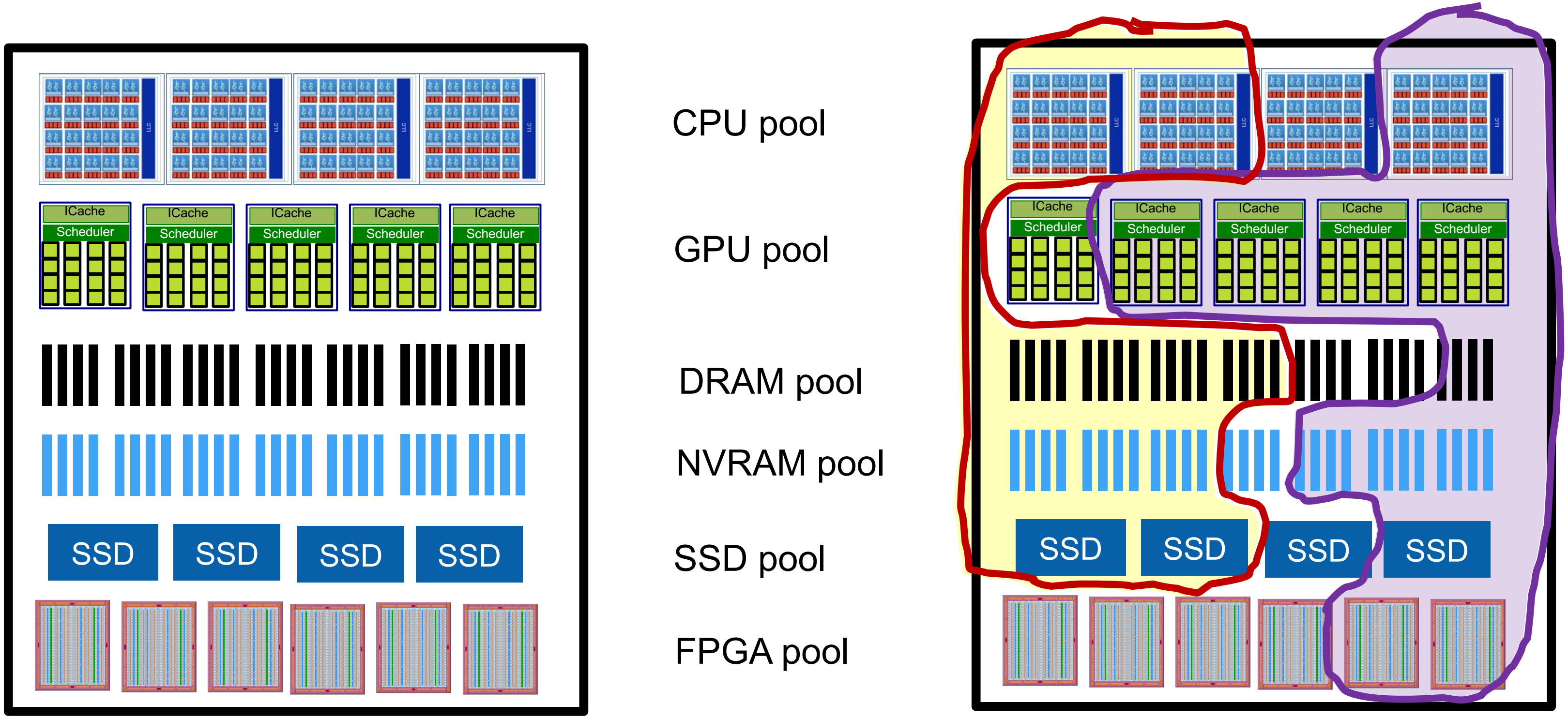
Computer Scientists need to rethink system SW stacks to minimize latencies …
fast RDMA, reduce sync contention, low latency interrupt handlers, and more ….
All to hit O(μs) latencies.

Source: Fig 1 from "Attack of the Killer Microseconds", Barroso, Marty, Patterson, and Ranganathan, Comm ACM vol 60, # 4, p. 48, 2017

Disaggregated Computing for SW Defined Servers (SDS):

Consider a Rack composed of multiple pools

Dynamically compose across pools to match a software defined server to the workload



CPU pool

GPU pool

DRAM pool

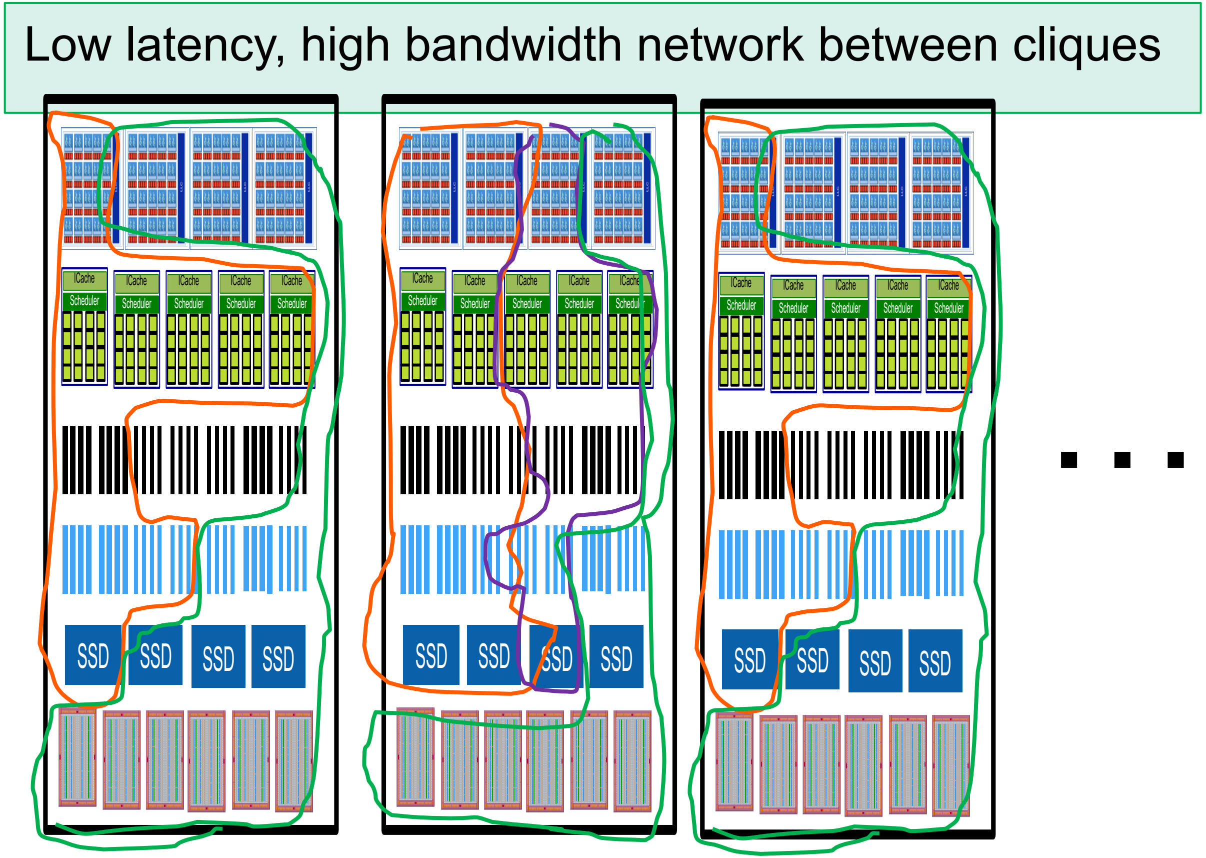
NVRAM pool

SSD pool

FPGA pool

Based on “The five Epochs of distributed computing” talk by Amin Vahdat of Google:

SW Defined clusters of SW defined Servers



■ ■ ■

- Dynamic … changing from one job to the next.
- SW defined severs composed of heterogeneous components
- Dynamically composed into a cluster
- Integrated over a 5G network to devices (and people) at the edge

Implications for Software development

A High-Level Taxonomy of parallel applications*

- Program: a sequence of operations (work) that modify data
- Task: a subset of the work defined by a program.
- Parallel application: a collection of tasks that run in parallel.
  - The tasks are usually concurrent (i.e. unordered) except for fixed points where they synchronize (often including an exchange of data).
  - Time is regular when synchronization events happen at ~common frequencies between tasks.
  - Data is regular when is it roughly the same size across a set of tasks.
- We define the following four application classes:

Synchronous: regular Data, regular time

Loosely synchronous: regular Data, irregular time

Asynchronous: irregular in Time and Data

Embarrassingly Parallel: independent tasks

The programmer must assure that the needs of the application can be met by the hardware

*Source: The Caltech Concurrent Computation Project, mid-1980s

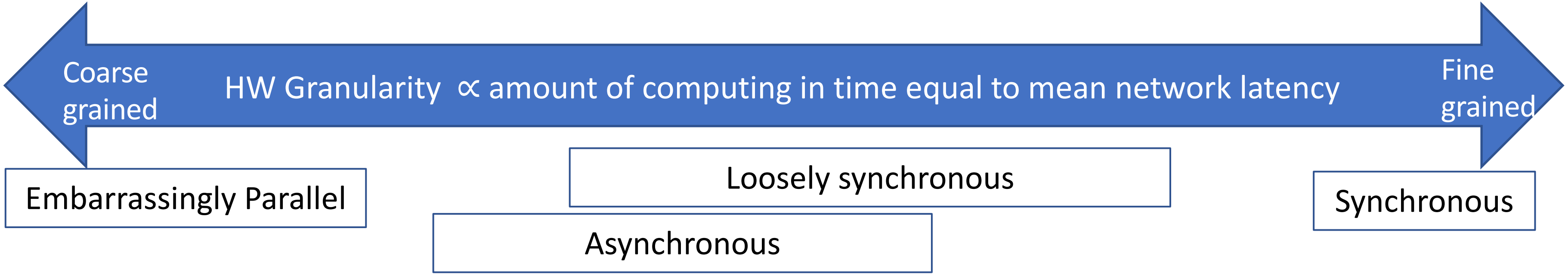
Distributed/Parallel Computing today

Cloud

- ✗ 1. ~~The network is reliable~~
- ✗ 2. ~~Latency is low and fixed~~
- ✗ 3. ~~Bandwidth is high and fixed~~
- ✗ 4. ~~The network is secure~~
- ✗ 5. ~~Topology doesn't change~~
- ✗ 6. ~~There is one administrator~~
- ✗ 7. ~~Transport cost is negligible~~
- ✗ 8. ~~The network is homogeneous~~

HPC Cluster

- ✓ 1. The network is reliable
- ✓ 2. Latency is low and fixed
- ✓ 3. Bandwidth is high and fixed
- ✓ 4. The network is secure
- ✓ 5. Topology doesn't change
- ✓ 6. There is one administrator
- ✗ 7. ~~Transport cost is negligible~~
- ✓ 8. The network is homogeneous



Coarse grained

HW Granularity ∝ amount of computing in time equal to mean network latency

Fine grained

Embarrassingly Parallel

Loosely synchronous

Asynchronous

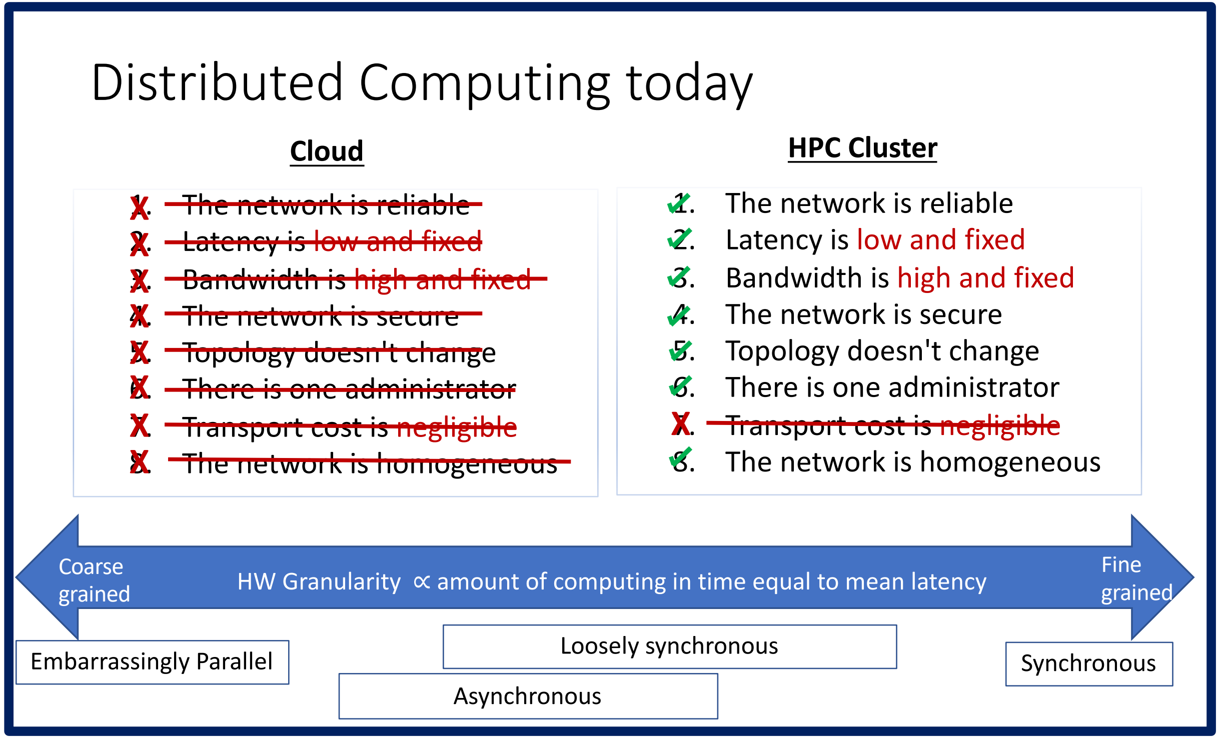
Synchronous

Programming Distributed computers

There is a clean split between applications that run in the cloud and those that need a dedicated HPC cluster.

This is reflected in the programming models used:

- Cloud: Remote Procedure Call (RPC), distributed object store distinct from tasks, execution flows as task graphs for Function as a service. Heavy use of microservices.
- HPC Cluster: SPMD design pattern with MPI … also PGAS with SHMEM.



Distributed Computing today

Cloud

✗1. ~~The network is reliable~~
✗2. ~~Latency is low and fixed~~
✗3. ~~Bandwidth is high and fixed~~
✗4. ~~The network is secure~~
✗5. ~~Topology doesn't change~~
✗6. ~~There is one administrator~~
✗7. ~~Transport cost is negligible~~
✗8. ~~The network is homogeneous~~

HPC Cluster

✓1. The network is reliable
✓2. Latency is low and fixed
✓3. Bandwidth is high and fixed
✓4. The network is secure
✓5. Topology doesn't change
✓6. There is one administrator
✗7. ~~Transport cost is negligible~~
✓8. The network is homogeneous

Coarse grained — HW Granularity ∝ amount of computing in time equal to mean latency — Fine grained

Embarrassingly Parallel

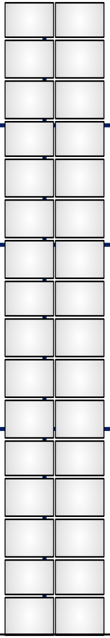
Loosely synchronous

Asynchronous

Synchronous

The three domains of parallel programming

Platform*	Laptop or server	HPC Cluster	Cloud
Execution Agent	Threads	Processes	Microservices
Memory	Single Address Space	Distributed memory, local memory owned by individual processes	Distributed object store (in memory) backed by a persistent storage system
Typical Execution Pattern	Fork-join	SPMD	Event driven tasks, FaaS, and Actors

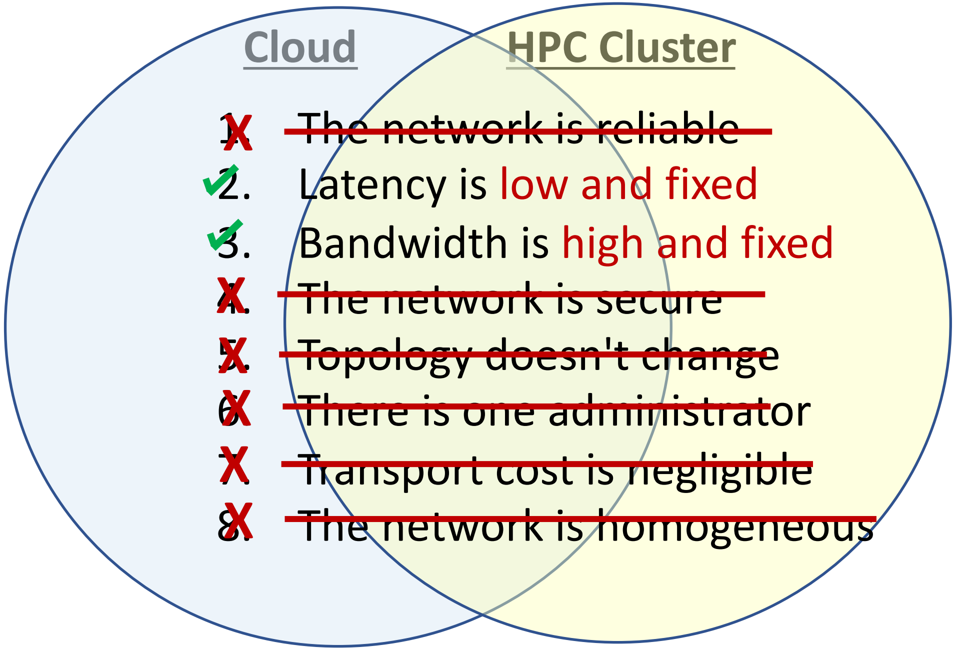


Laptop/server and cluster models work well together.

An impenetrable wall separates them from the cloud-native world

Optically-networked disaggregated cloud systems ... cloud and cluster overlap … or even merge!

Chip-to-chip optical networks push latency down and bandwidth up



✗ 1. ~~The network is reliable~~
✓ 2. Latency is low and fixed
✓ 3. Bandwidth is high and fixed
✗ 4. ~~The network is secure~~
✗ 5. ~~Topology doesn't change~~
✗ 6. ~~There is one administrator~~
✗ 7. ~~Transport cost is negligible~~
✗ 8. ~~The network is homogeneous~~

Data Streaming Accelerator reduces tail latency.

P4/P5/P6 + Infrastructure Processing Units drive down latency and reduces jitter

With Low Latencies, high bandwidths and stable performance, we can do loosely synchronous and synchronous applications in the cloud.   The economics of the cloud vs dedicated HPC clusters means the cloud will dominate HPC

HPC applications will need to change to deal with reliability and network inhomogeneities.

The three domains of parallel programming

Platform*	Laptop or server	HPC Cluster	Cloud
Execution Agent	Threads	Processes	Microservices
Memory	Single Address Space	Distributed memory, local memory owned by individual processes	Distributed object store (in memory) backed by a persistent storage system
Typical Execution Pattern	Fork-join	SPMD	Event driven tasks, FaaS, and Actors

There will always be a need for top-end scalable systems in supercomputer centers, but economics will push the bulk of scientific computing into the cloud.

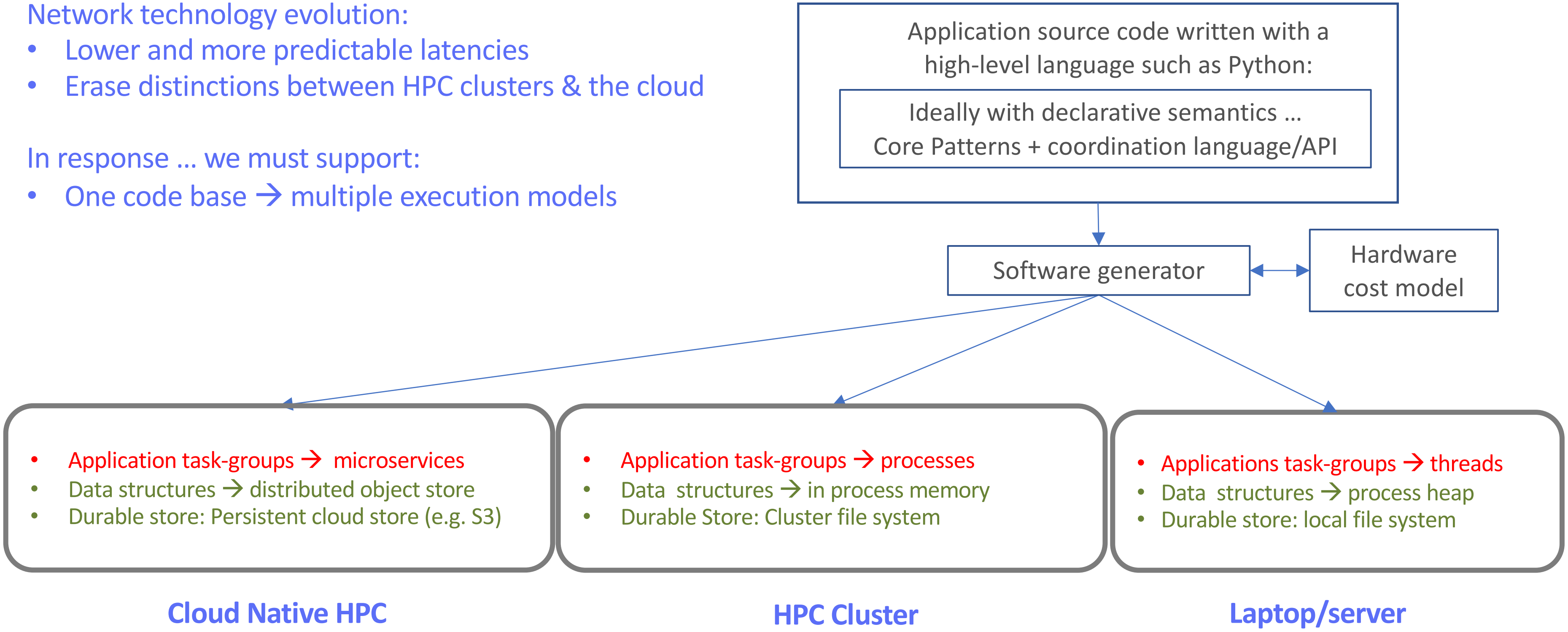
Writing Parallel Distributed Applications

Network technology evolution:

- Lower and more predictable latencies
- Erase distinctions between HPC clusters & the cloud

In response … we must support:

- One code base → multiple execution models



Application source code written with a high-level language such as Python:

Ideally with declarative semantics …
Core Patterns + coordination language/API

Software generator

Hardware cost model

- Application task-groups → microservices
- Data structures → distributed object store
- Durable store: Persistent cloud store (e.g. S3)

Cloud Native HPC

- Application task-groups → processes
- Data structures → in process memory
- Durable Store: Cluster file system

HPC Cluster

- Applications task-groups → threads
- Data structures → process heap
- Durable store: local file system

Laptop/server

Writing Parallel Distributed Applications

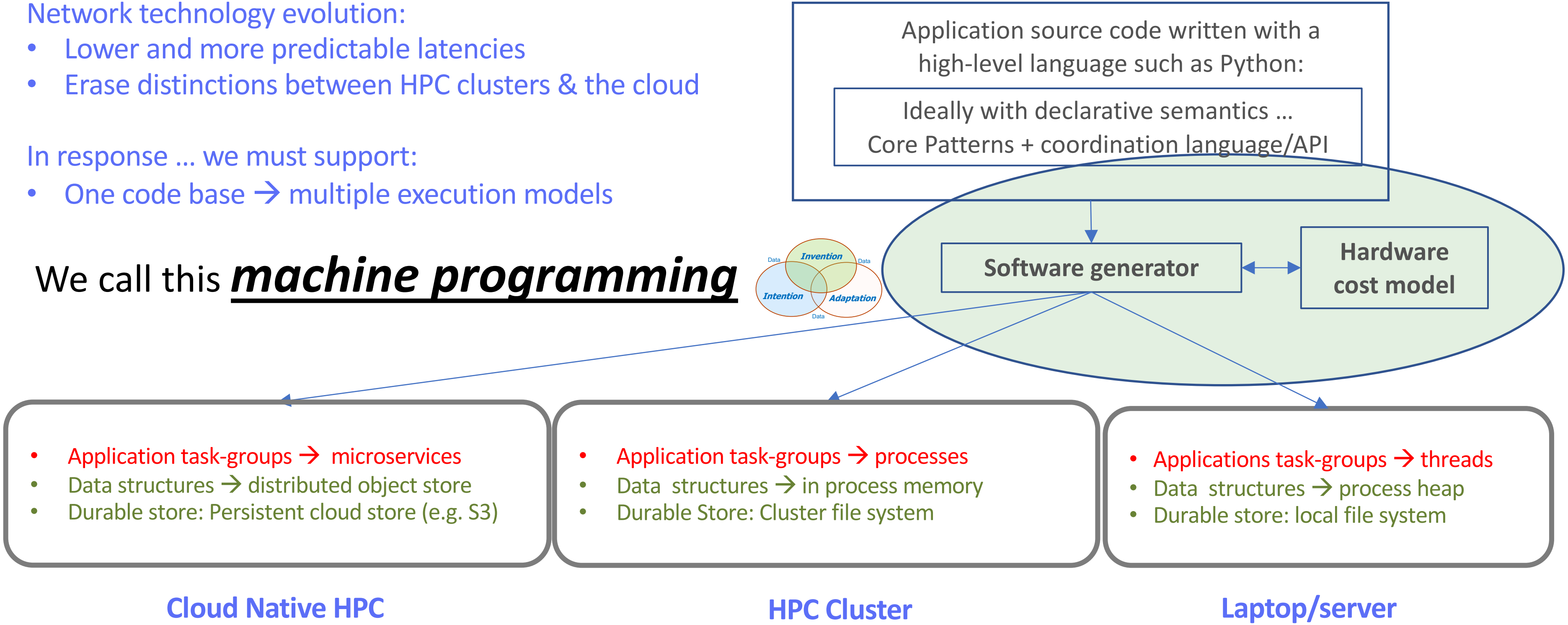
Network technology evolution:

- Lower and more predictable latencies
- Erase distinctions between HPC clusters & the cloud

In response … we must support:

- One code base → multiple execution models

We call this ***machine programming***



Application source code written with a high-level language such as Python:

Ideally with declarative semantics …
Core Patterns + coordination language/API

Software generator

Hardware cost model

- Application task-groups → microservices
- Data structures → distributed object store
- Durable store: Persistent cloud store (e.g. S3)

Cloud Native HPC

- Application task-groups → processes
- Data structures → in process memory
- Durable Store: Cluster file system

HPC Cluster

- Applications task-groups → threads
- Data structures → process heap
- Durable store: local file system

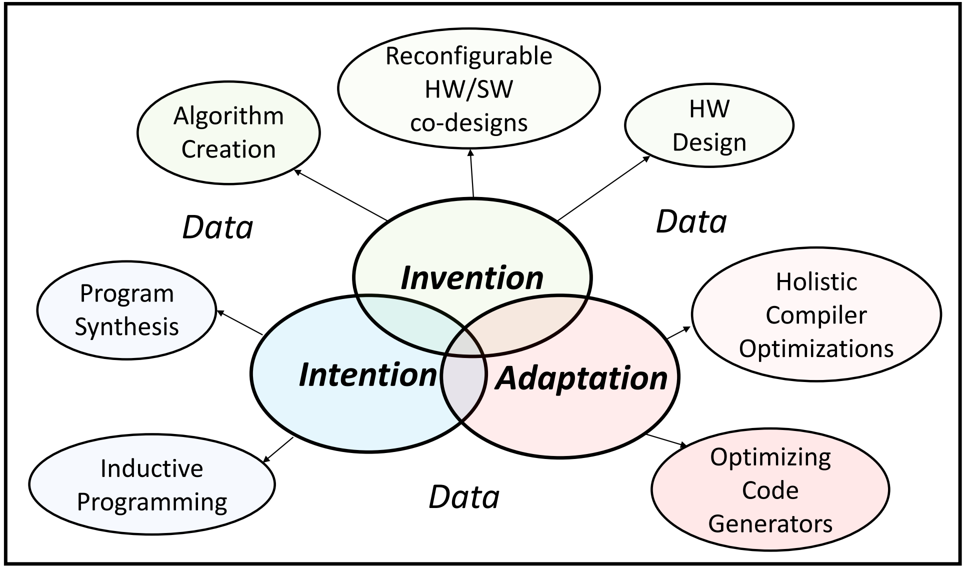
Laptop/server

The Three Pillars of Machine Programming

MAPL/PLDI'18

This is a long talk I won't go into now

Justin Gottschlich, Intel
Armando Solar-Lezama, MIT
Nesime Tatbul, Intel
Michael Carbin, MIT
Martin, Rinard, MIT
Regina Barzilay, MIT
Saman Amarasinghe, MIT
Joshua B Tenebaum, MIT
Tim Mattson, Intel



A position paper laying out our vision for how to solve the machine programming problem. The three Pillars:

- **Intention**: Discover the intent of a programmer
- **Invention**: Create new algorithms and data structures
- **Adaption**: Evolve in a changing hardware/software world

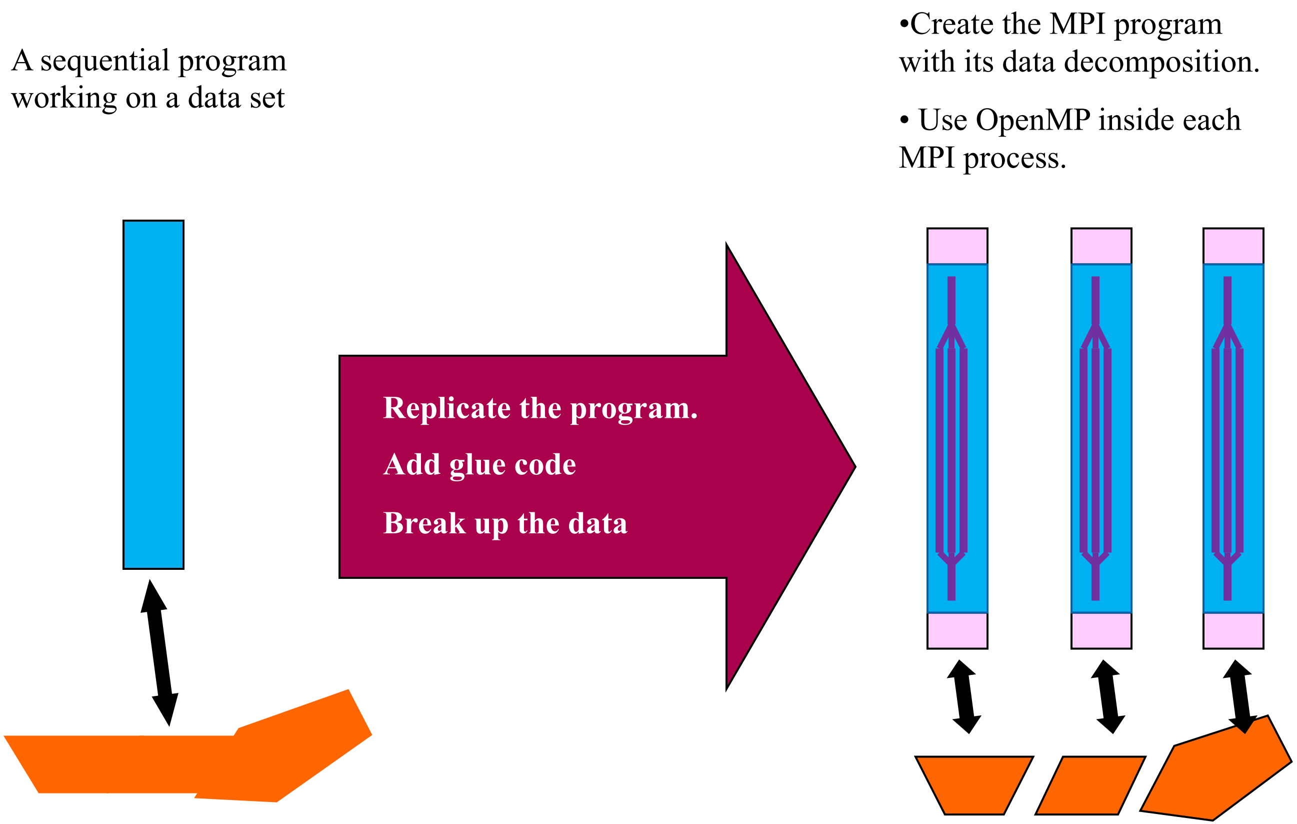
2nd ACM SIGPLAN Workshop on Machine Learning and Programming Languages (MAPL), PLDI'18, arxiv.org/pdf/1803.07244.pdf

OK, Now we are really done

Backup Content

- Mixing OpenMP and MPI
- Loading MPI on your system

How do people mix MPI and OpenMP?



A sequential program working on a data set

Replicate the program.

Add glue code

Break up the data

•Create the MPI program with its data decomposition.

• Use OpenMP inside each MPI process.

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) ;
        MPI_Finalize();
}
```

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 Content

- 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`	

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
```

Grab 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
```