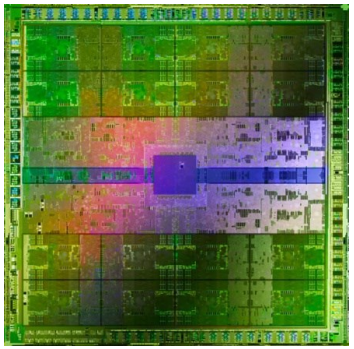
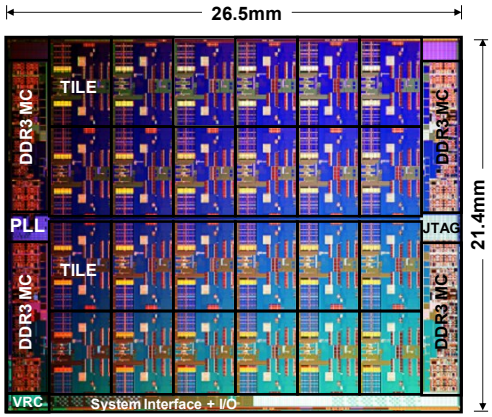
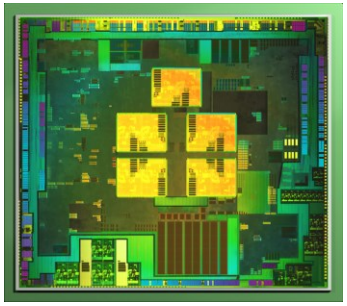


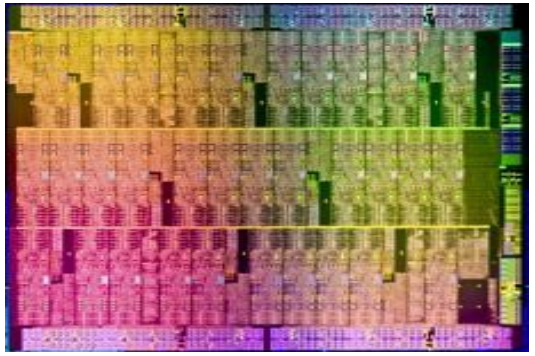
NVIDIA GTX 480 processor



Intel labs 48 core SCC processor



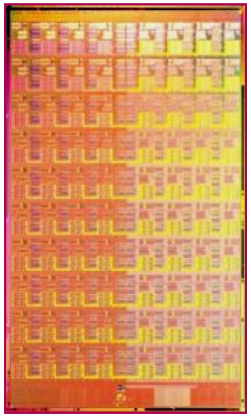
NVIDIA Tegra 3 (quad Arm Corex A9 cores + GPU)



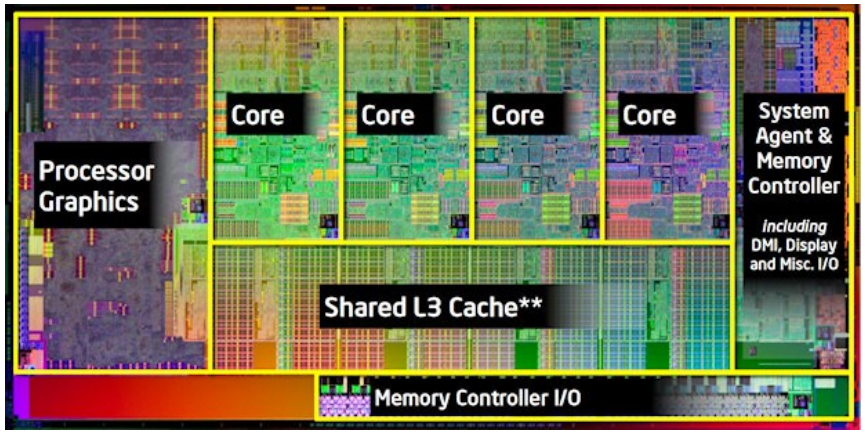
An Intel MIC processor

A hand's on introduction to Cluster Computing

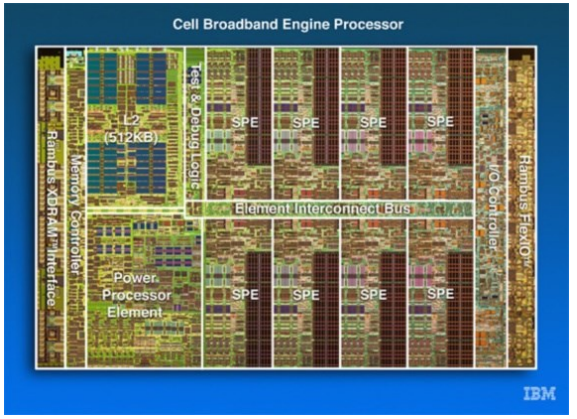
Tim Mattson (Intel Labs)



Intel Labs 80 core Research processor



Intel "Sandybridge" processor



IBM Cell Broadband engine processor

Other than the Intel lab's research processors. Die photos from UC Berkeley CS194 lecture notes

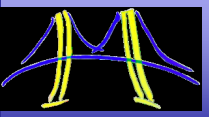
Third party names are the property of their owners

Disclaimer

READ THIS … its very important

- The views expressed in this talk are those of the speakers and not their employer.
- This is an academic style talk and does not address details of any particular Intel product. You will learn nothing about Intel products from this presentation.
- This was a team effort, but if we say anything really stupid, it's our fault … don't blame our collaborators.

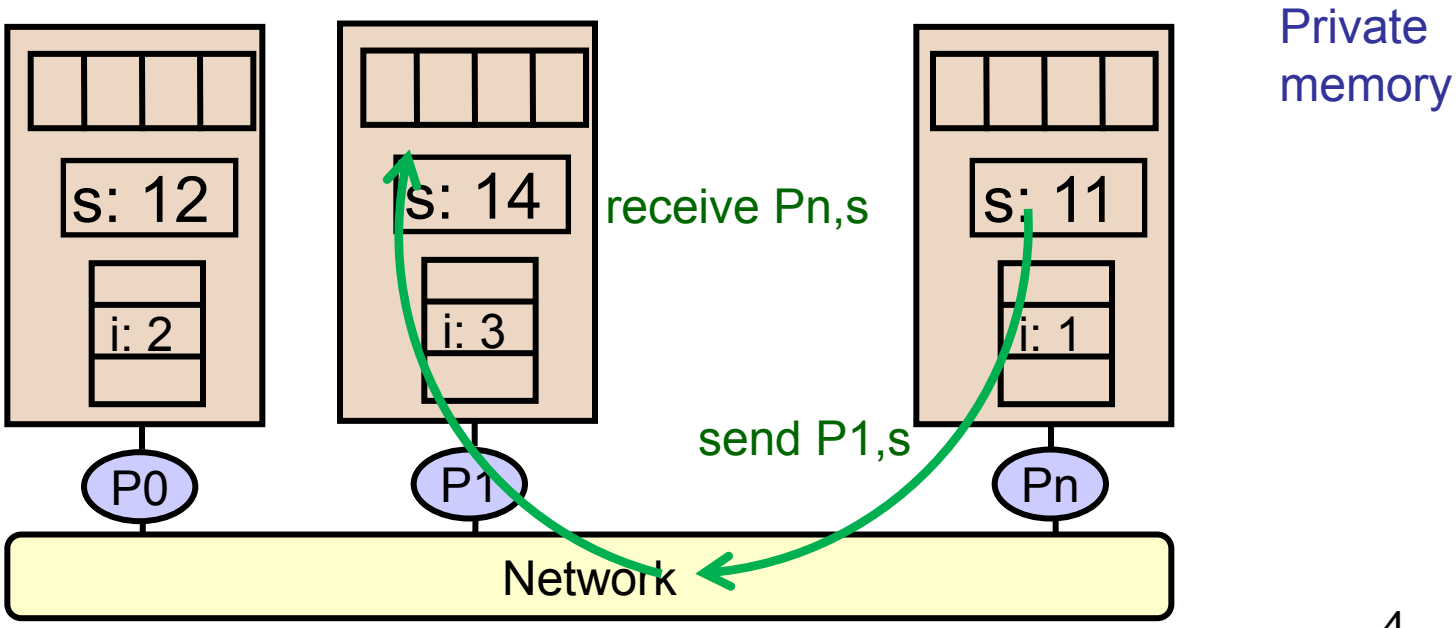
Slides marked with this symbol were produced-with Kurt Keutzer and his team for CS194 … A UC Berkeley course on Architecting parallel applications with Design Patterns.

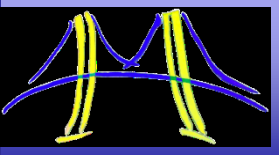


INTRODUCTION TO MPI

Programming Model: Message Passing

- Program consists of a collection of **named** processes.
  - **Number of processes almost always fixed at program startup time**
  - **Local address space per node -- NO physically shared memory.**
  - **Logically shared data is partitioned over local processes.**
- Processes communicate through explicit communication events.
  - **Coordination is implicit in every communication event.**
  - **MPI (Message Passing Interface) is the most commonly used SW**





MPI: Message Passing Interface

MPI_Type_contiguous

MPI_Bcast

MPI_Recv_init

MPI_Group_size

MPI_S

ORLD

MPI_

mpare

C$(

artall

MPI_

I_Pack

MPI_

ck)

MPI_Sendrecv_replace

MPI_Ssend

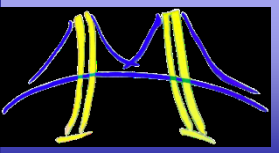
MPI_Waitall

MPI_Alltoallv

MPI_Send

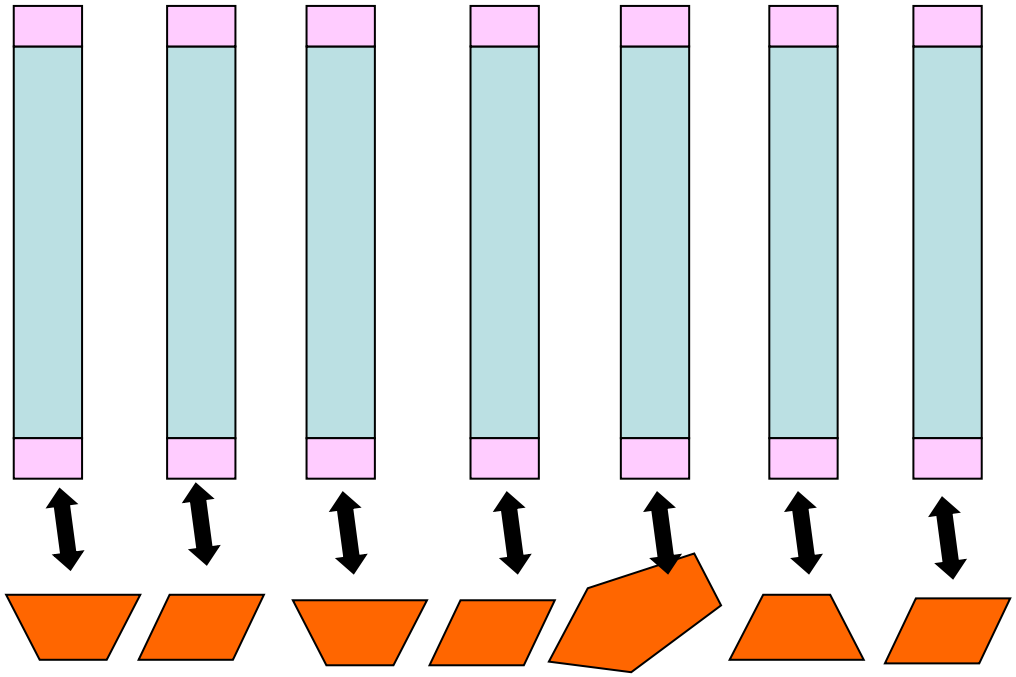
MPI: An API for Writing Clustered Applications

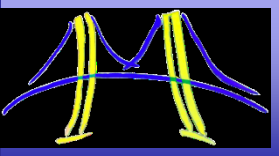
- **A library of routines to coordinate the execution of multiple processes.**
- **Provides point to point and collective communication in Fortran, C and C++**
- **Unifies last 30+ years of cluster computing and MPP practice**



An MPI program at runtime

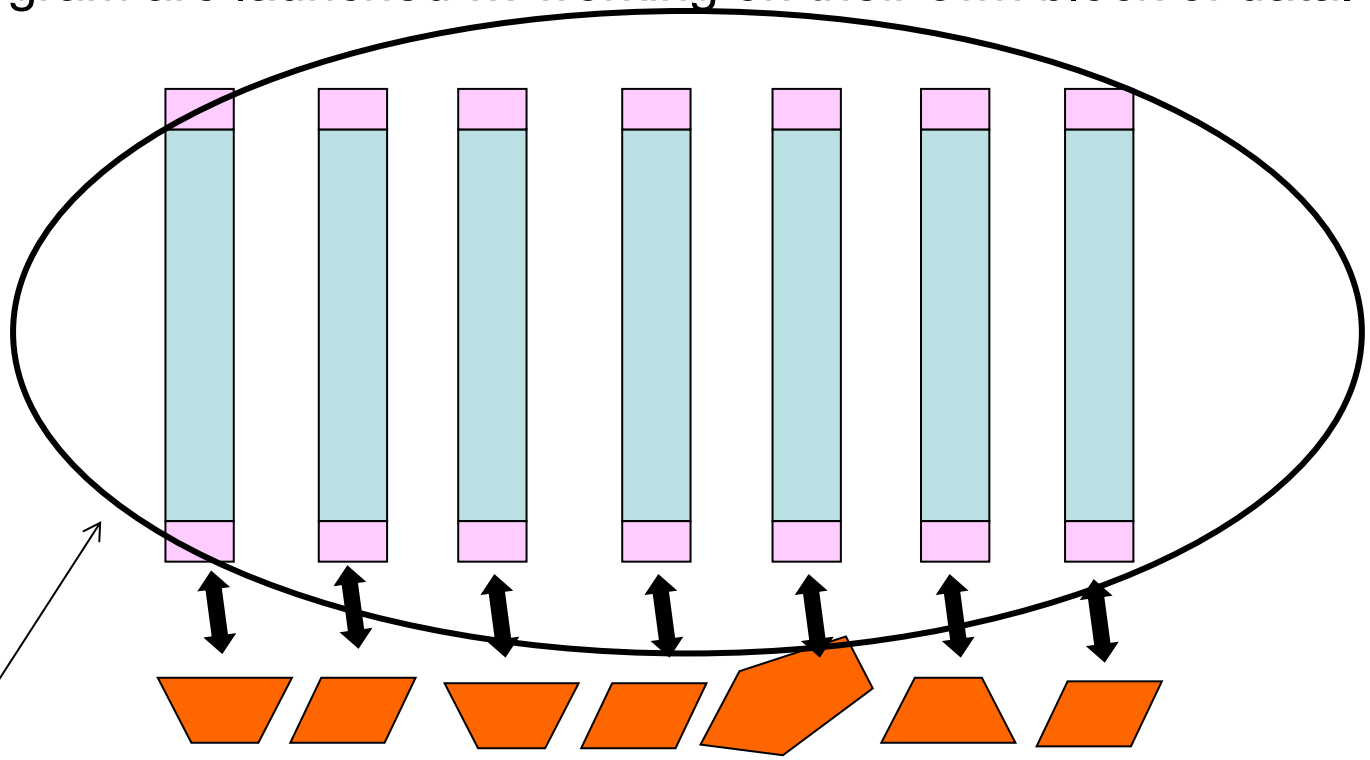
- Typically, when you run an MPI program, multiple processes all running the same program are launched … working on their own block of data.



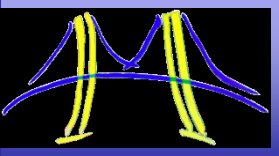


An MPI program at runtime

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

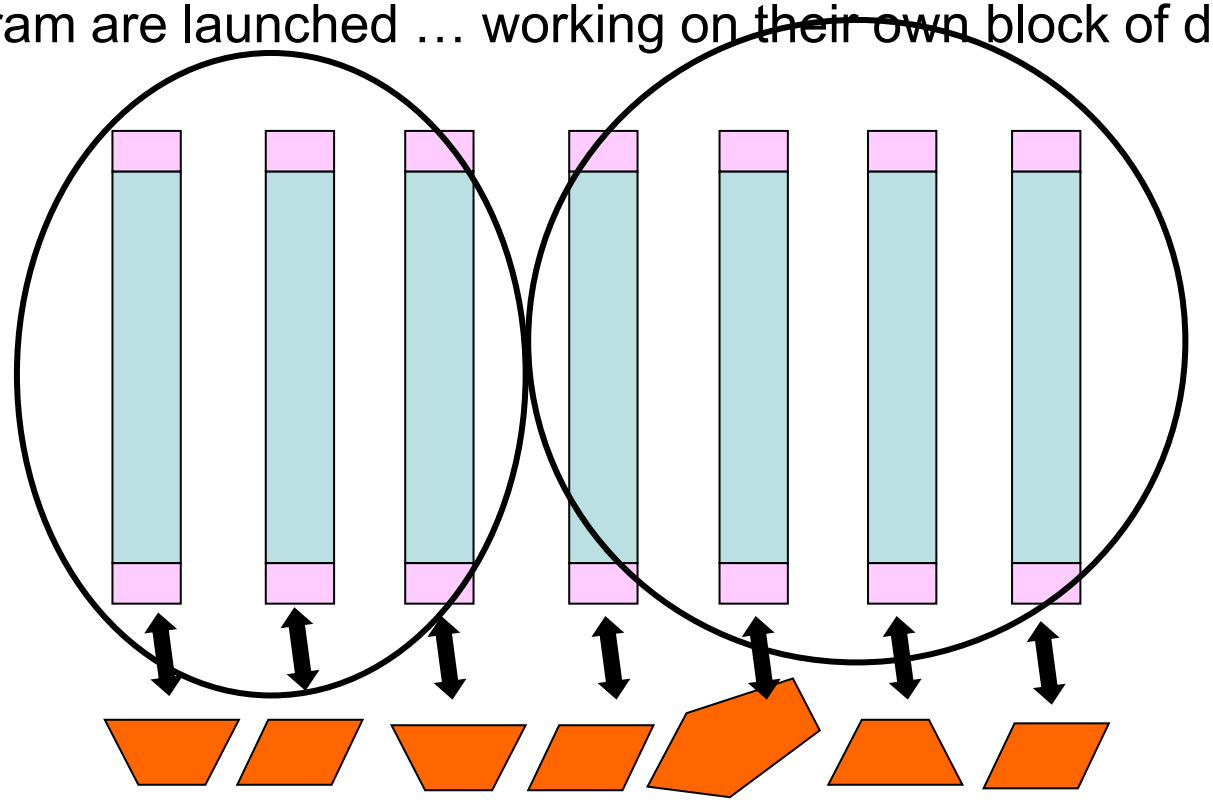


The collection of processes involved in a computation is called “a **process group**”



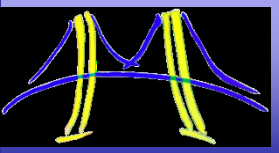
An MPI program at runtime

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



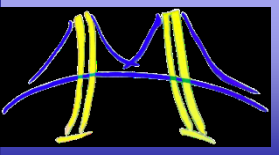
You can dynamically split a **process group** into multiple subgroups to manage how processes are mapped onto different tasks

MPI functions work within a "**context**" … events in different contexts … even if they share a process group … cannot interfere with each other.



MPI Hello World

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



Initializing and finalizing MPI

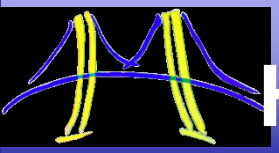
```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **argv){
    int rank, size;
    MPI_Init (&argc, &argv);
    MPI_Comm_rank (MPI_COMM_WORLD, &rank);
    MPI_Comm_size (MPI_COMM_WORLD, &size);
    printf( "Hello from process %d of %d\n",
                                rank, size );
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

`int MPI_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()

`int MPI_Finalize (void)`

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



How many processes are involved?

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

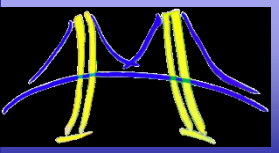
- `MPI_Comm`, an *opaque data type called a communicator. D*efault context: MPI_COMM_WORLD (all processes)
- `MPI_Comm_size` returns the number of processes in the process group associated with the communicator

```
#include <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 me control how groups of messages interact.

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



Which process "am I" (the rank)

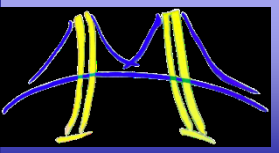
```
int MPI_Comm_rank (MPI_Comm comm, int* rank)
```

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

```
#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.

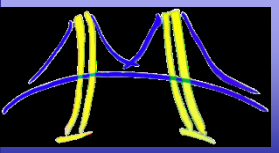
This makes sense .. You need a context and group of processes that the MPI functions impact … and those come from the communicator.



Running the program

- On a 4 node cluster, I'd run this program (hello) as:
  > mpiexec –n 4 hello
- What would this program would output?

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



Exercise 1: Hello world

- Goal
  - To confirm that you can run a 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 process in the process group printing “hello world” and its rank

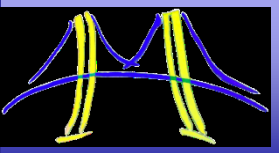
```
#include <mpi.h>
int size, rank, argc;   char **argv;
MPI_Init (&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Finalize();
```

To run the executable hello on 2 nodes of the cluster, type:

```
> mpiexec –n 4 a.out
```

If you need to start the MPI deamon (mpd) go to your home directory and type:

```
> touch .mpd.conf
> chmod 600 .mpd.conf
```

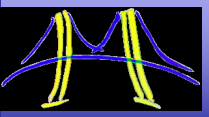


Running the program

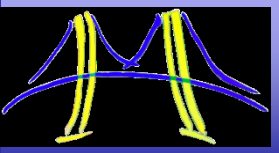
- On a 4 node cluster, I'd run this program (hello) as:

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

```
#include <stdio.h>
#include <mpi.h>
int main (int argc, char **a
    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 FOR BULK SYNCHRONOUS PROGRAMS



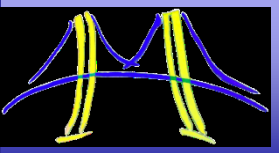
Sending and Receiving Data

```
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_Send** performs a blocking send of the specified data ("count" copies of type "datatype," stored in "buf") to the specified destination (rank "dest" within communicator "comm"), with message ID "tag"

- **MPI_Recv** performs a blocking receive of specified data from specified source whose parameters match the send; information about transfer is stored in "status"

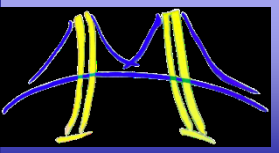
By "blocking" we mean the functions return as soon as the buffer, "buf", can be safely used.



MPI Data Types for C

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

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



The data in a message: datatypes

- The data in a message to send or receive is described by a triple:
  - **(address, count, datatype)**
- An MPI datatype is defined as:
  - Predefined, simple data type from the language (e.g., MPI_DOUBLE)
  - Complex data types (contiguous blocks or even custom types.
- E.g. … A particle's state is defined by its 3 coordinates and 3 velocities

```
MPI_Datatype PART;
MPI_Type_contiguous( 6, MPI_DOUBLE, &PART );
MPI_Type_commit( &PART );
```

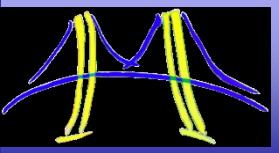
- You can use this data type in MPI functions, for example, to send data for a single particle:

```
MPI_Send (buff, 1, PART, Dest, tag, MPI_COMM_WORLD);
```

address

count

Datatype



Receiving the right message

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

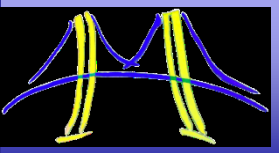
MPI_Recv (buff, 1, PART, Src, tag, MPI_COMM_WORLD, &status);

Source tag

- Can relax tag checking by specifying MPI_ANY_TAG as the tag in a receive.
- Can relax source checking by specifying MPI_ANY_SOURCE

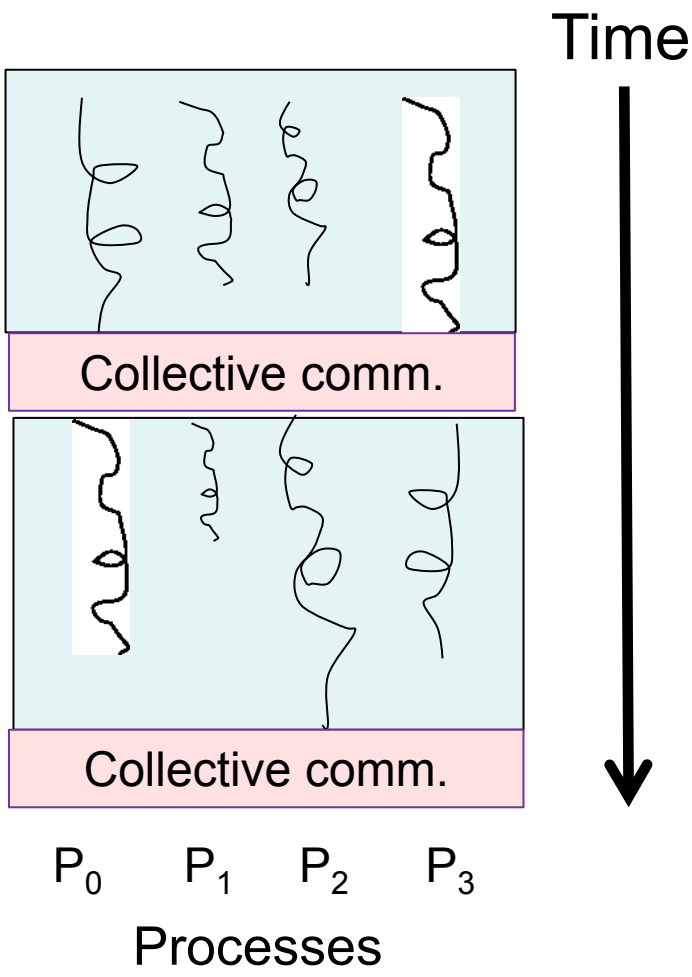
MPI_Recv (buff, 1, PART, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &status);

- This is a useful way to insert race conditions into an MPI program

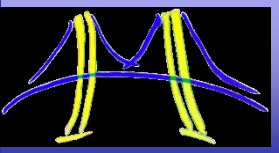


A typical pattern with MPI Programs

- Many MPI applications have few (if any) sends and receives. 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

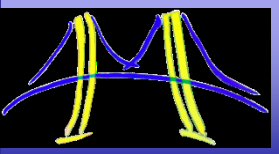


This is a subset or the SPMD pattern sometimes referred to as the Bulk Synchronous pattern.



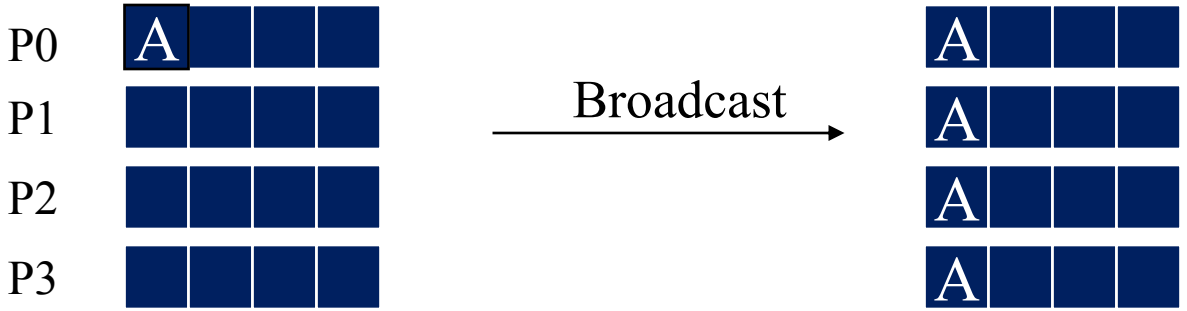
MPI Collective Routines

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



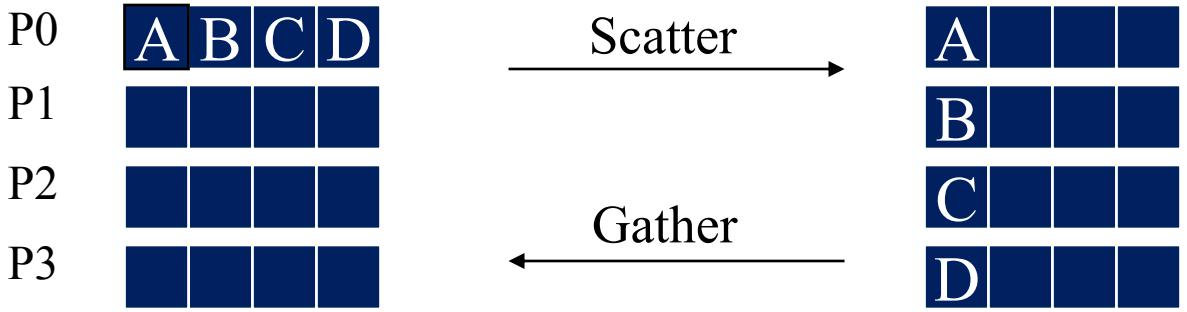
Collective Data Movement

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



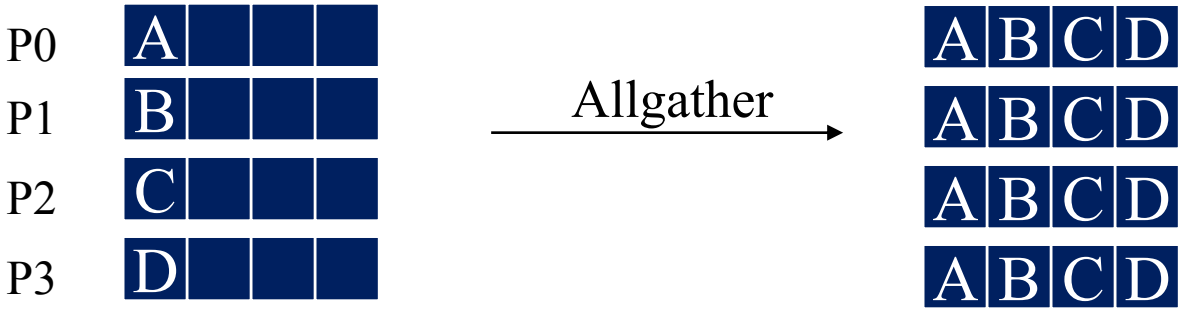
Scatter an array on P0 to P1, P2, and P3

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

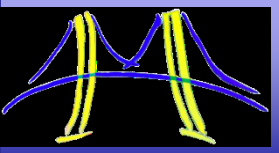


More Collective Data Movement

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

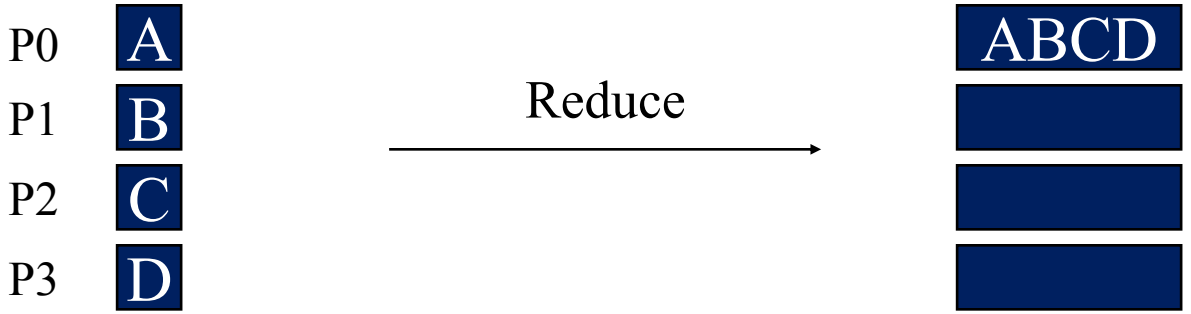


Take arrays on each P and spread them out to arrays on each P

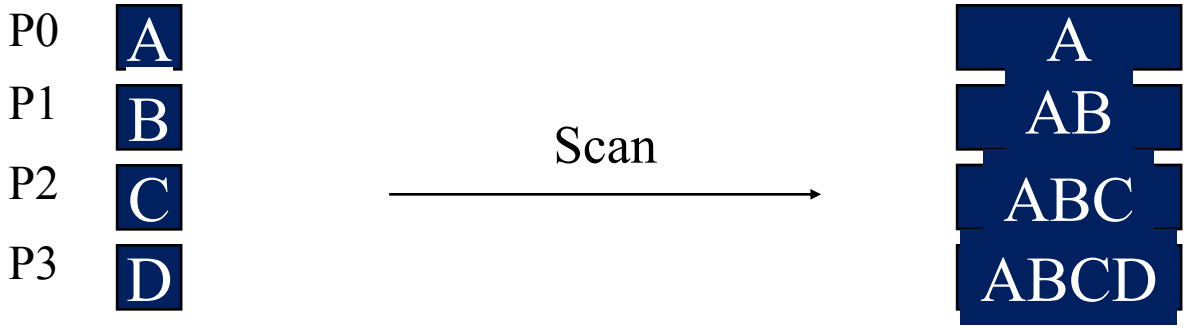


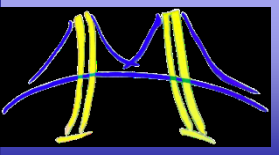
Collective Computation

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



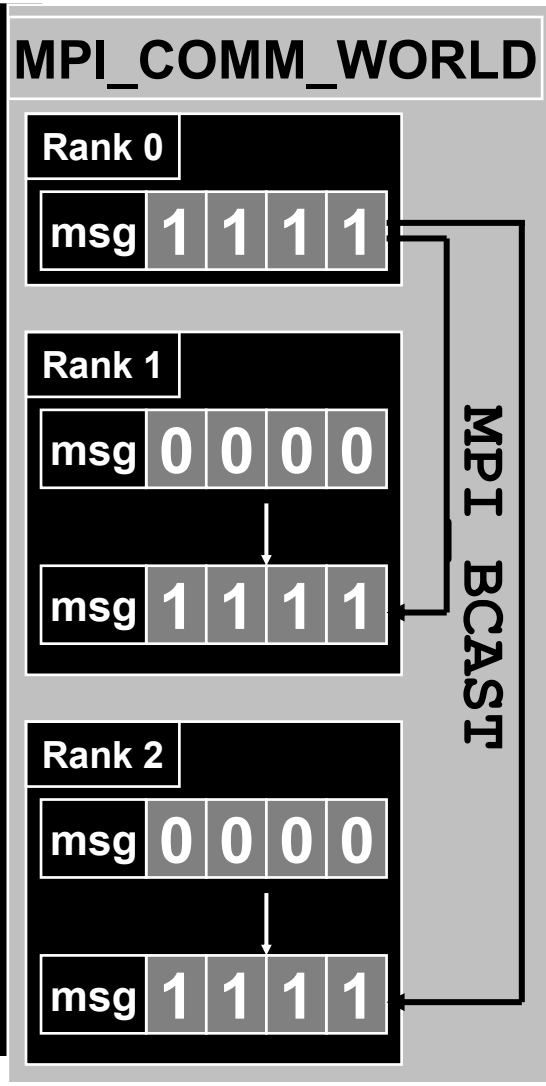
Take values on each P and combine them with a scan operation and spread the scan array out among all P.

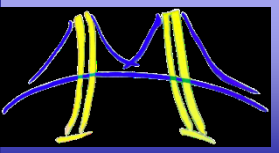




MPI_BCAST Example

```
#include <mpi.h>

int main(int argc, char *argv[]) {
 int nprocs, myrank, msg[4] = {0,0,0,0};

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

 if (myrank == 0) msg[0] = 1;
 else             msg[0] = 0;

 MPI_Bcast(msg, 4, MPI_INT, 0, MPI_COMM_WORLD);

 MPI_Finalize();
}
```





Reduction

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

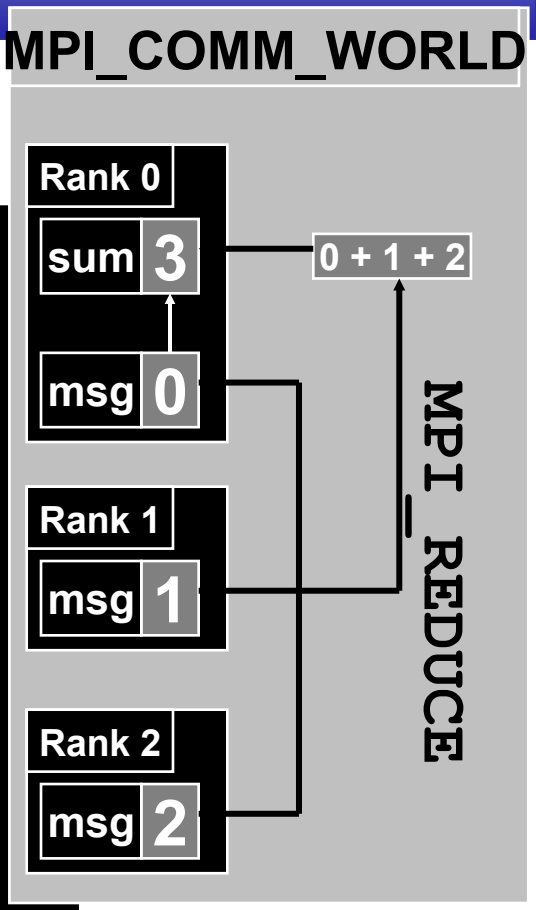
- **`MPI_Reduce`** performs specified reduction operation on specified data from all processes in communicator, places result in process “root” only.
- **`MPI_Allreduce`** places result in all processes (avoid unless necessary)

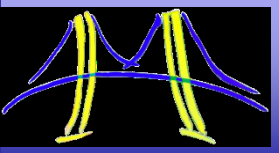
Operation	Function
MPI_SUM	Summation
MPI_PROD	Product
MPI_MIN	Minimum value
MPI_MINLOC	Minimum value and location
MPI_MAX	Maximum value
MPI_MAXLOC	Maximum value and location
MPI_LAND	Logical AND

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 msg, sum, nprocs, myrank;

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

  sum = 0;
  msg = myrank;

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

  MPI_Finalize();
}
```





Exercise 2: Pi Program

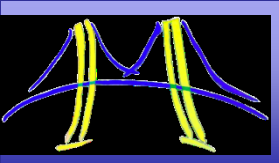
- Goal
  - To write a simple Bulk Synchronous, SPMD program
- Program
  - Start with the provided “pi program” and using an MPI reduction, write a parallel version of the program.

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

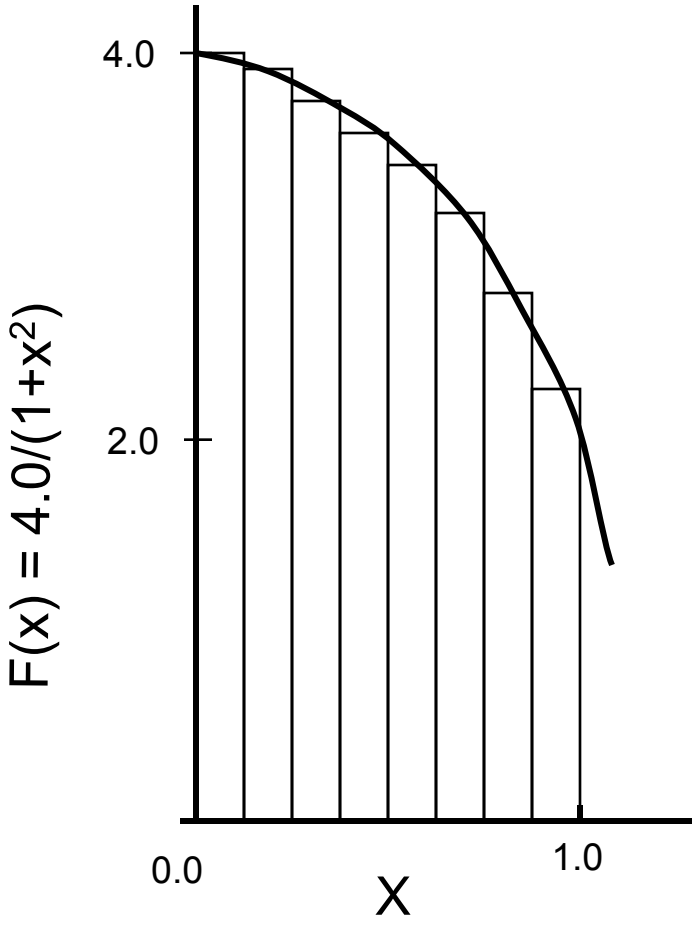
Operation	Function
MPI_SUM	Summation
MPI_PROD	Product

```
#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



Example Problem: Numerical Integration



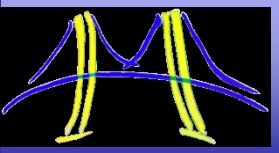
Mathematically, we know that:

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

We can approximate the integral as a sum of rectangles:

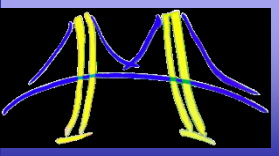
$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

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;
}
```



Pi program in MPI

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

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

MPI Pi program performance

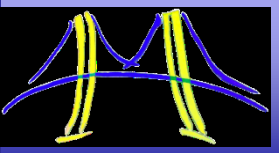
Pi program in MPI

```
#include <mpi.h>
void main (int argc, char *argv[])
{
        int i, my_id, numprocs; double x, pi, step, sum
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_Rank(MPI_COMM_WORLD,
        MPI_Comm_Size(MPI_COMM_WORLD, &

        for (i=my_id; i<num_steps; ; i=i+numprocs)
        {
                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)
}
```

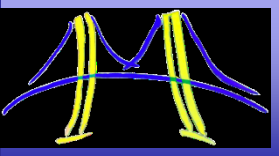
Thread or procs	OpenMP SPMD critical	OpenMP PI Loop	MPI
1	0.85	0.43	0.84
2	0.48	0.23	0.48
3	0.47	0.23	0.46
4	0.46	0.23	0.46

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

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

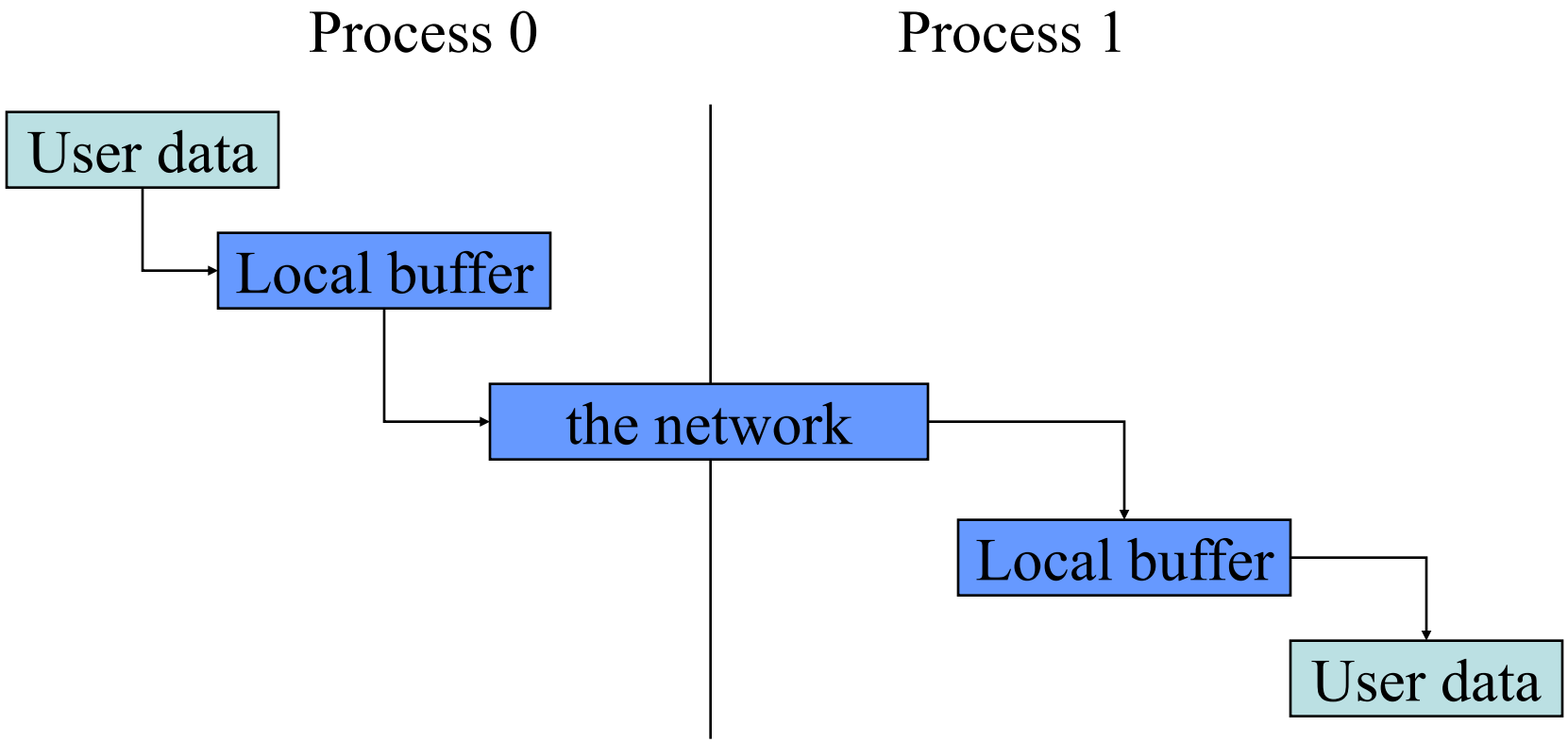


UNDERSTANDING MESSAGE PASSING WITH MPI

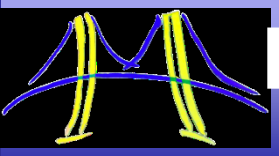


Buffers

- Message passing has a small set of primitives, but there are subtleties
  - Buffering and deadlock
  - Deterministic execution
  - Performance
- When you send data, where does it go? One possibility is:

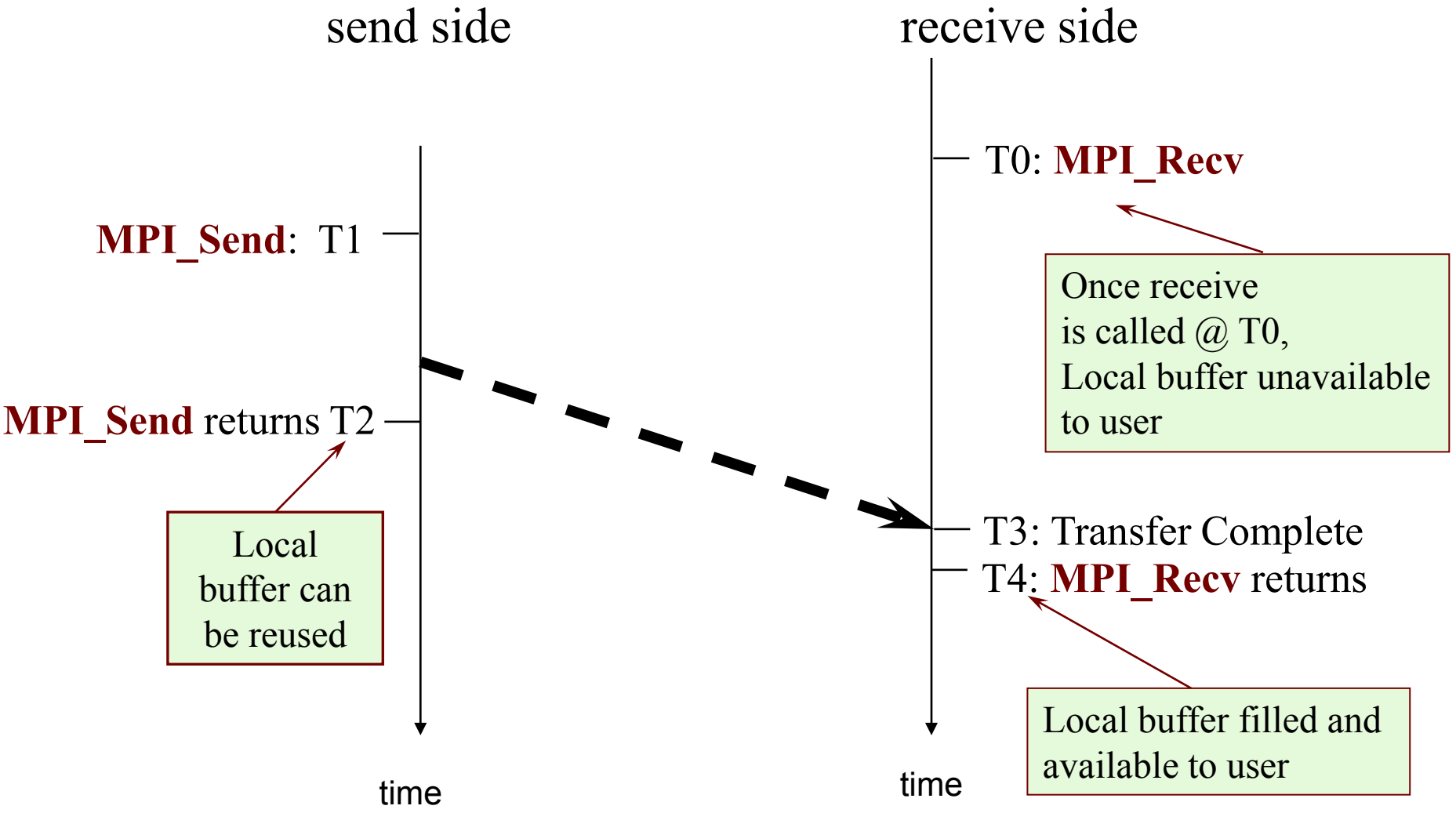


Derived from: Bill Gropp, UIUC

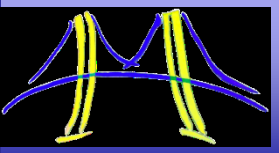


Blocking Send-Receive Timing Diagram

(Receive before Send)



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



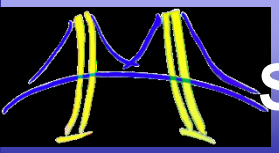
Sources of Deadlocks

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

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

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

Slide source: based on slides from Bill Gropp, UIUC



Some Solutions to the "deadlock" Problem

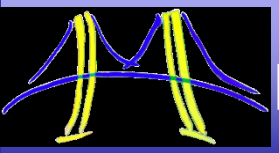
- Order the operations more carefully:

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

- Supply receive buffer at same time as send:

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

Slide source: Bill Gropp, UIUC



More Solutions to the "unsafe" Problem

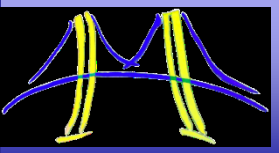
- Supply a sufficiently large buffer in the send function

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

- Use non-blocking operations:

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

Slide source: Bill Gropp, UIUC

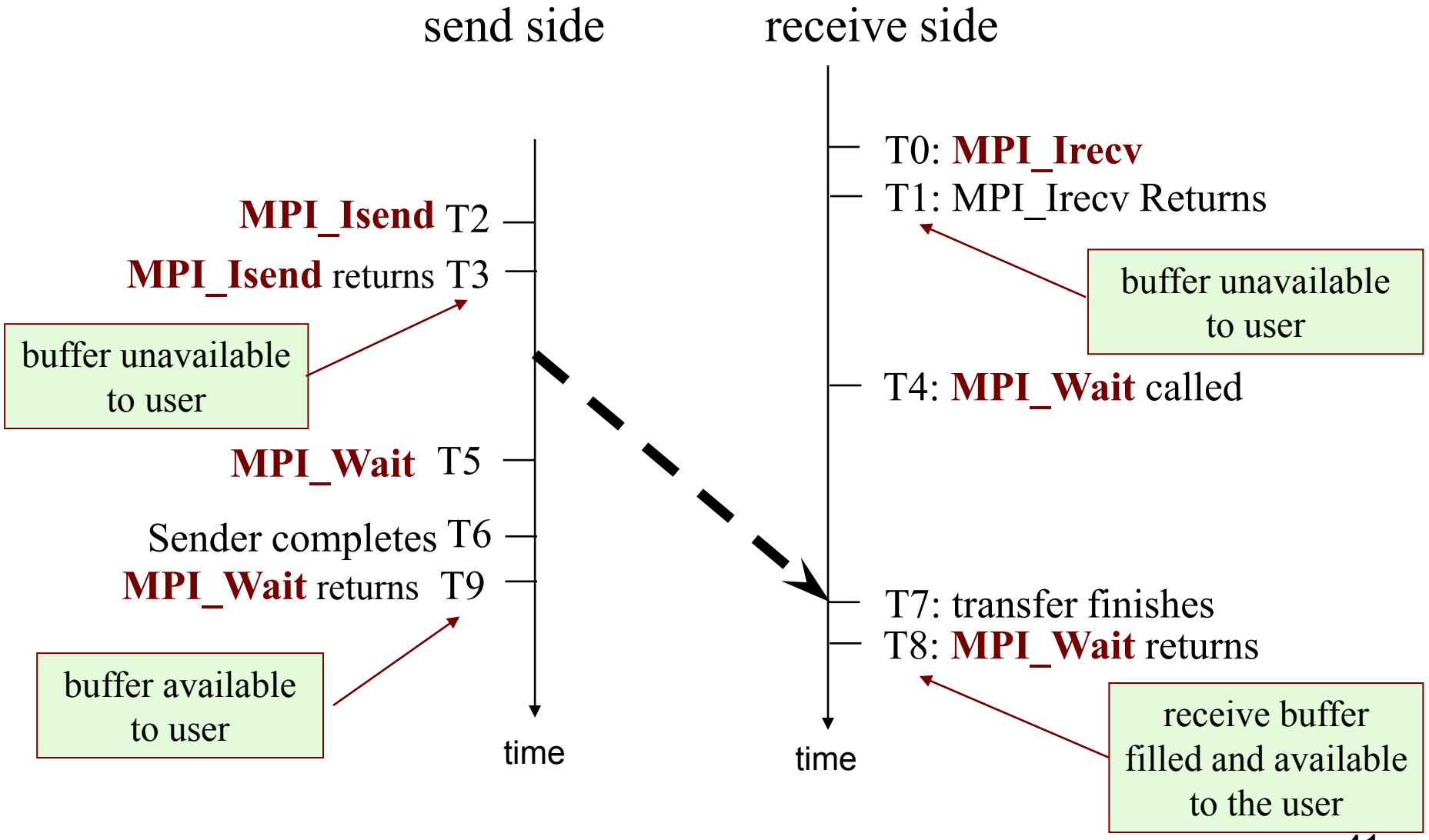


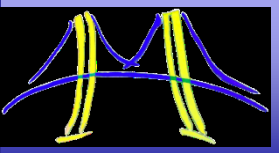
Non-Blocking Communication

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

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

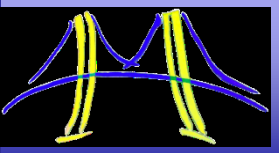
Non-Blocking Send-Receive Diagram





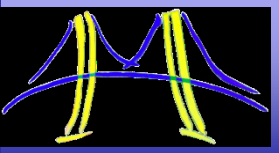
Example: shift messages around a ring (part 1 of 2)

```
#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);
  }
```



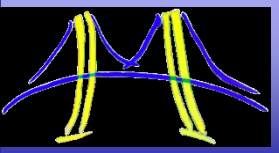
Example: shift messages around a ring (part 2 of 2)

```
do {
  MPI_Irecv(&num, 1, MPI_INT, from, tag, MPI_COMM_WORLD, &req2);
  MPI_Wait(&req2, &status2);
  printf("Process %d received %d from process %d\n", rank, num, from);

  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;
}
```



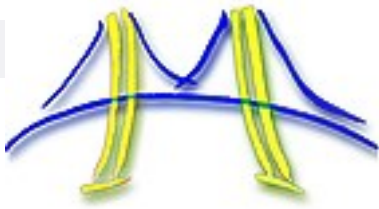
Exercise 3: Ring program

- Goal
  - Explore other modes of message passing in MPI
- Program
  - Start with the basic ring program we provide. Run it for a range of message sizes and notes what happens for large messages.
  - If the program deadlocks (and it should) figure out why and how to fix it.
  - Try a range of message passing functions to understand how they work.



MPI AND THE GEOMETRIC DECOMPOSITION PATTERN

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

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

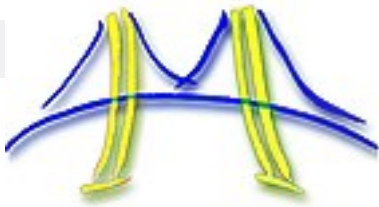
- Central difference approximation for spatial derivative (at fixed time)

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

- 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

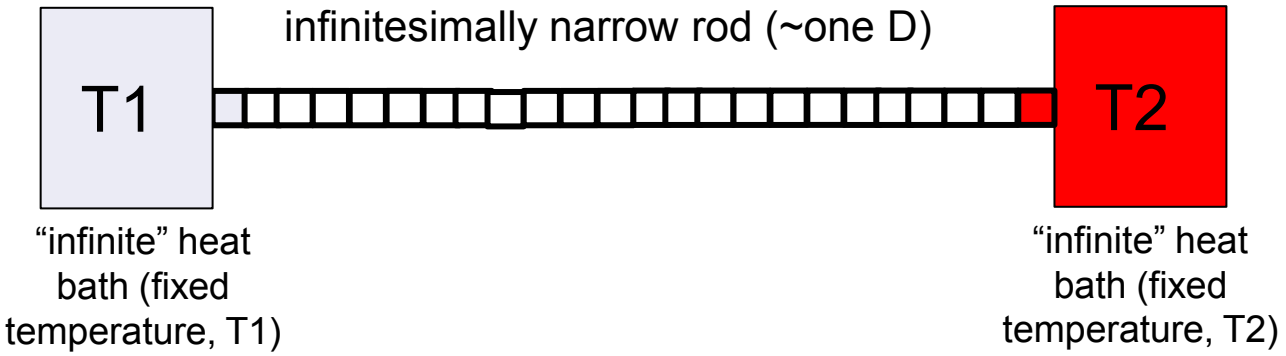
$$u_j^{n+1} = (1-2r)u_j^n + ru_{j-1}^n + ru_{j+1}^n \qquad r = k/h^2$$

- 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).

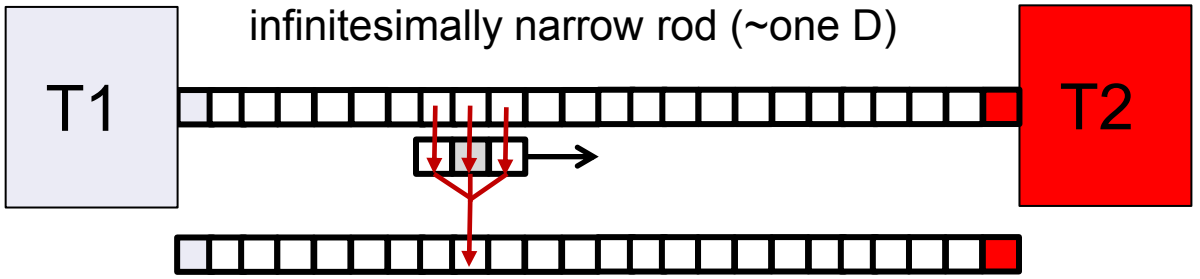
```
for (int t = 0; t < N_STEPS-1; ++t)
   for (int x = 1; x < N-1; ++x)
         u[t+1][x] = u[t][x] + r*(u[t][x+1] - 2*u[t][x] + u[t][x-1]);
```

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

Heat Diffusion equation

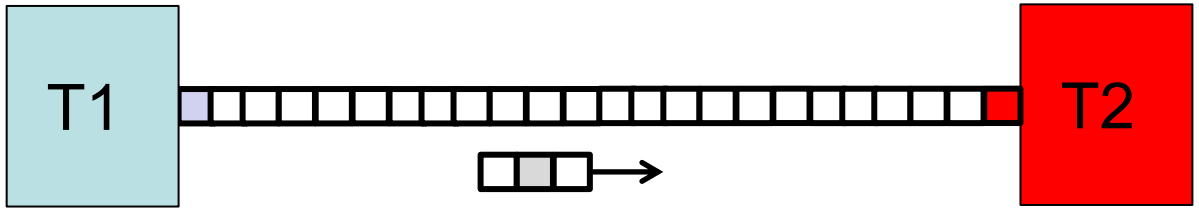


Heat Diffusion equation



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

Heat Diffusion equation

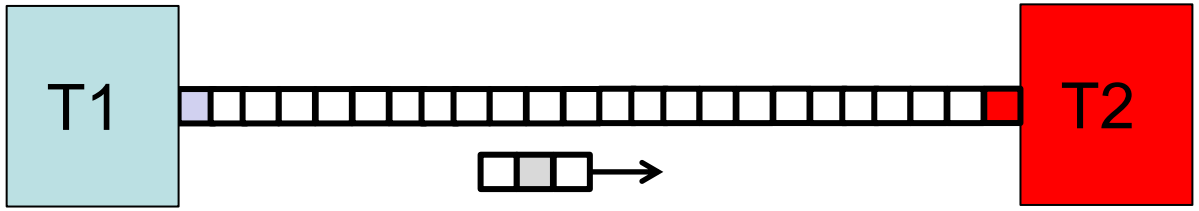



```
int main()
{
   double *u   = malloc (sizeof(double) * (N));
   double *up1 = malloc (sizeof(double) * (N));

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

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

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

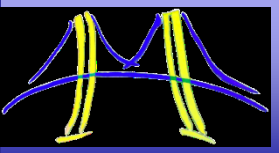
A well known trick with 2 arrays so I don't overwrite values from step k-1 as I fill in for step k

Heat Diffusion equation




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

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

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

How would you parallelize this program?



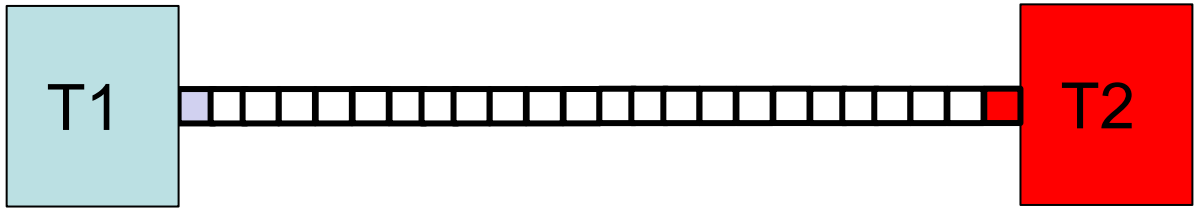
Seven strategies for parallelizing software

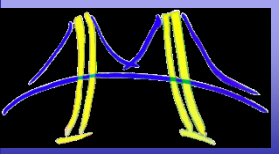
- These seven strategies for parallelizing software give us:
  - Names: so we can communicate better
  - Categories: so we can gather and share information
  - A palette (like an artist's palette) of approaches that is:
    - Necessary: we should consider them all and
    - Sufficient: once we have considered them all then we don't have to worry that we forgot something

Parallel Algorithm Strategy Patterns		
Task-Parallelism	Data-Parallelism	Discrete-Event
Divide and Conquer	Pipeline	Geometric-Decomposition
		Speculation

Heat Diffusion equation

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





Seven strategies for parallelizing software

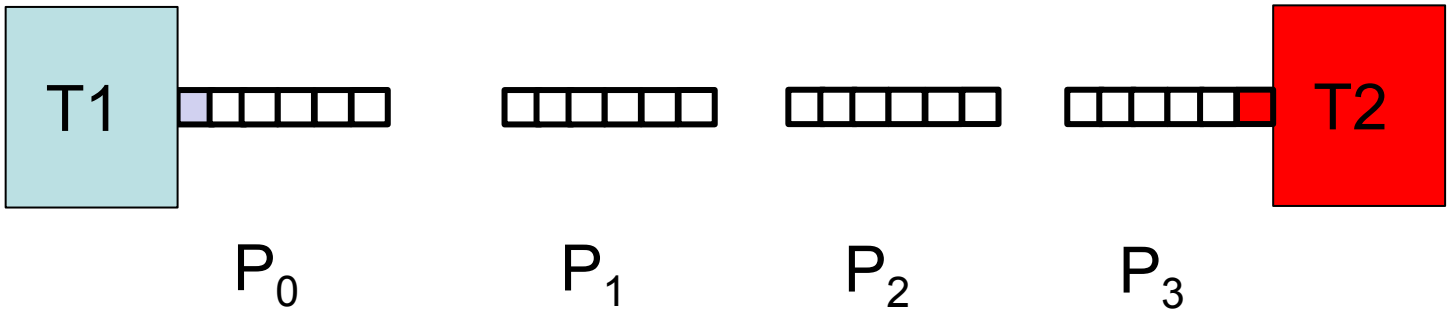
- These seven strategies for parallelizing software give us:
  - Names: so we can communicate better
  - Categories: so we can gather and share information
  - A palette (like an artist's palette) of approaches that is:
    - Necessary: we should consider them all and
    - Sufficient: once we have considered them all then we don't have to worry that we forgot something

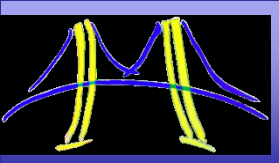
Parallel Algorithm Strategy Patterns

Task-Parallelism	Data-Parallelism	Discrete-Event
Divide and Conquer	Pipeline	Geometric-Decomposition
		Speculation

Heat Diffusion equation

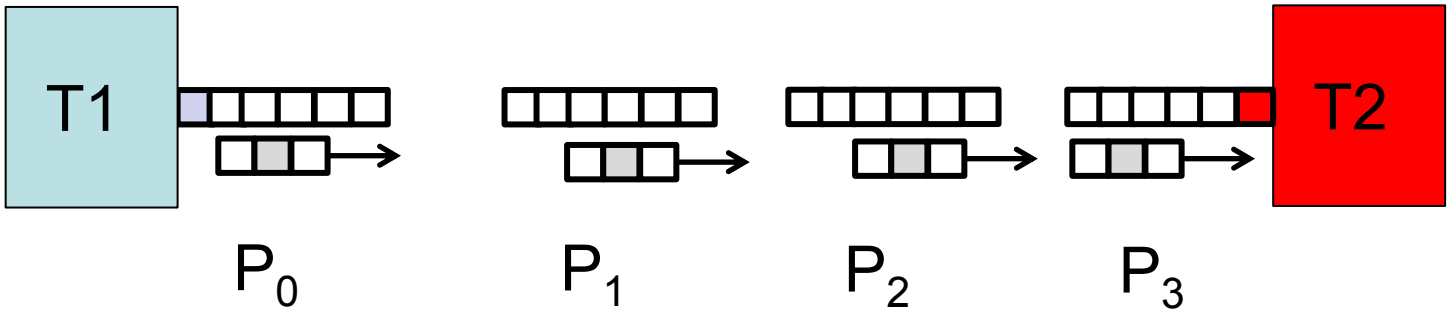
- Break it into chunks assigning one chunk to each process.

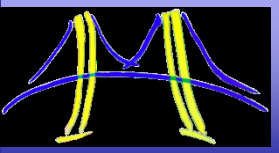




Heat Diffusion equation

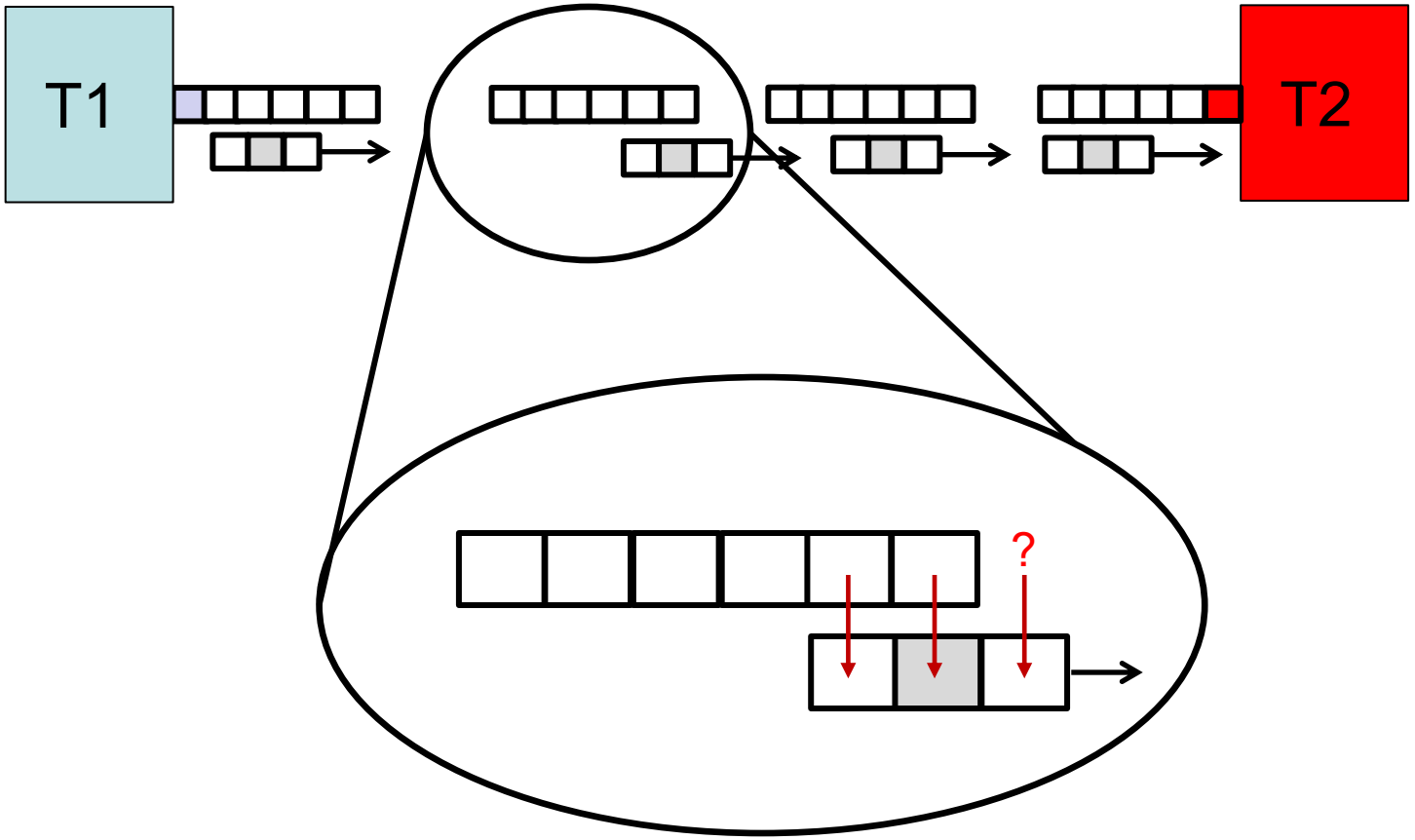
- Each process works on it's own chunk … sliding the stencil across the domain to updates its own data.

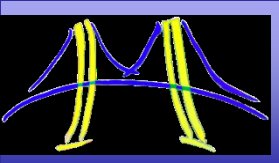




Heat Diffusion equation

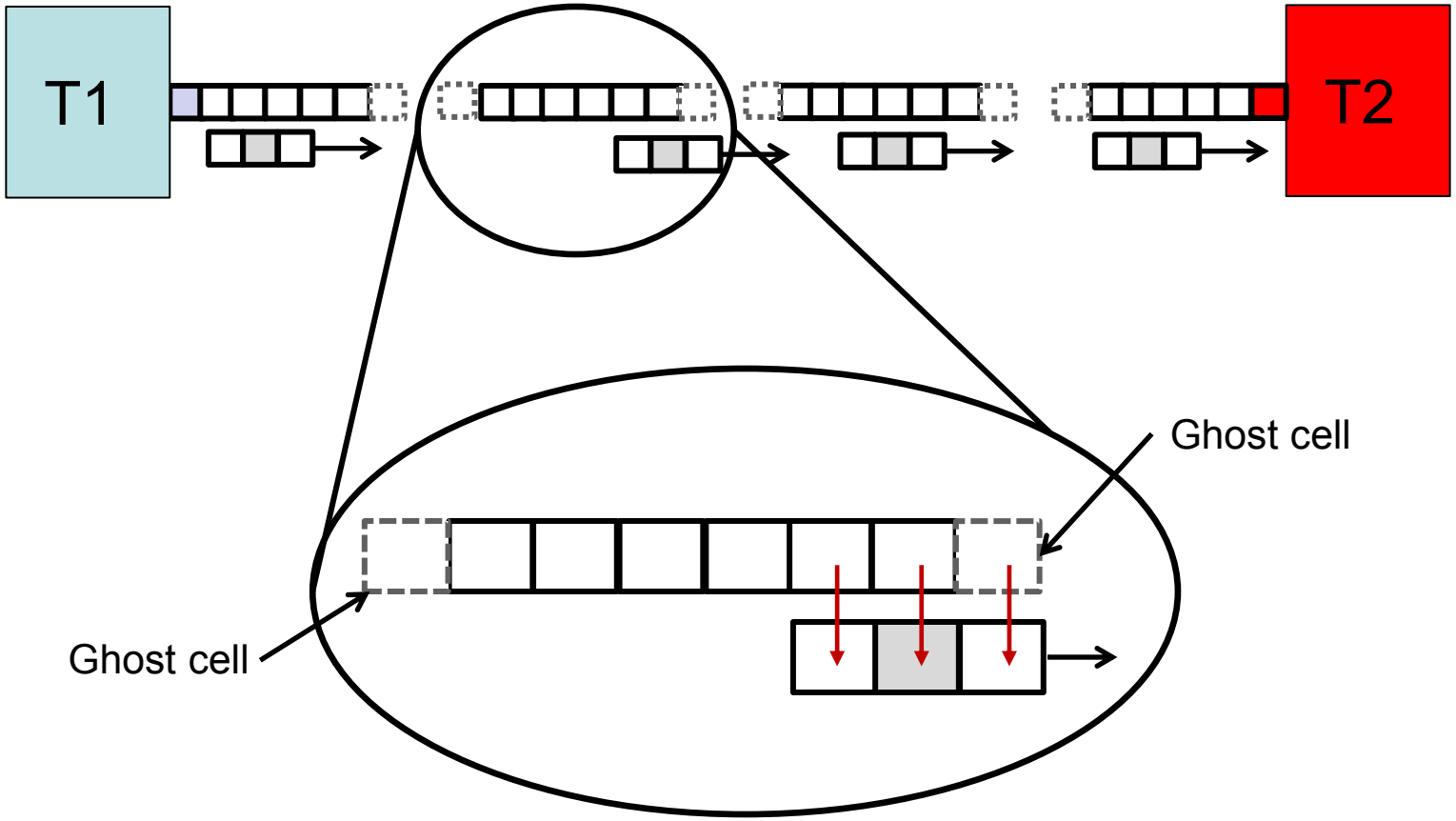
- What about the ends of each chunk … where the stencil will run off the end and hence have missing values for the computation?

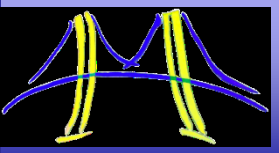




Heat Diffusion equation

- 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.



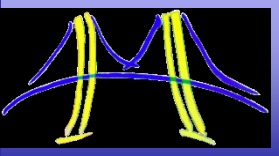


SPMD: Single Program Mulitple Data

- Run the same program on P processing elements where P can be arbitrarily large.
- Use the rank … an ID ranging from 0 to (P-1) … to select between a set of tasks and to manage any shared data structures.

This pattern is very general and has been used to support most (if not all) the algorithm strategy patterns.

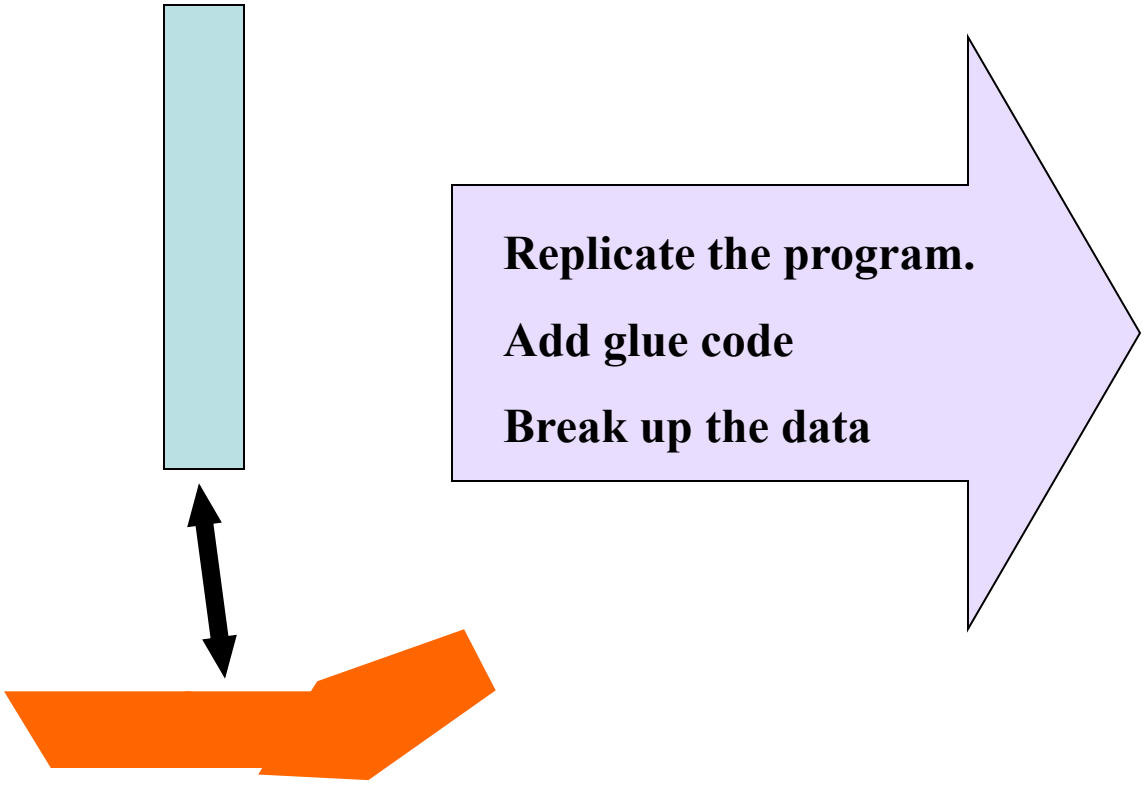
MPI programs almost always use this pattern … it is probably the most commonly used pattern in the history of parallel programming.

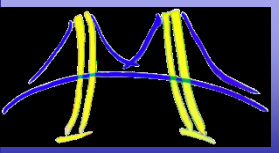


How do people use MPI? The SPMD Design Pattern

A sequential program working on a data set

- A single program working on a decomposed data set.
- Use Node ID and numb of nodes to split up work between processes
- Coordination by passing messages.

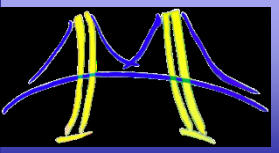




Heat Diffusion MPI Example

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

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

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

MPI_Finalize();
return 0;
```

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



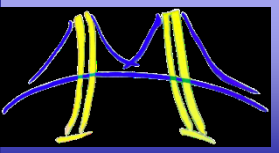
Heat Diffusion MPI Example

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

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

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

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

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

MPI_Finalize();
return 0;
```

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

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

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



Heat Diffusion MPI Example

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

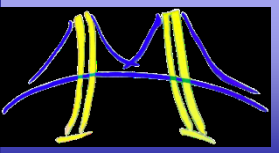
```
MPI_Init (&argc, &argv);
MPI_Comm_size (MPI_COMM_WORLD, &P);
MPI_Comm_rank (MPI_COMM_WORLD, &myID);
double *u   = malloc (sizeof(double) * (2 + N/P))  // include "Ghost Cells"
double *up1 = malloc (sizeof(double) * (2 + N/P)); // to hold values
                                                   // from my neighbors

initialize_data(uk, ukp1, N, P);
for (int t = 0; t < N_STEPS; ++t){
  if (myID != 0)
    MPI_Send (&u[1], 1, MPI_DOUBLE, myID-1, 0, MPI_COMM_WORLD);

  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);
/* continued on previous slide */
```

Send my "left" boundary value to the neighbor on my "left'

Receive my "right" ghost cell from the neighbor to my "right'

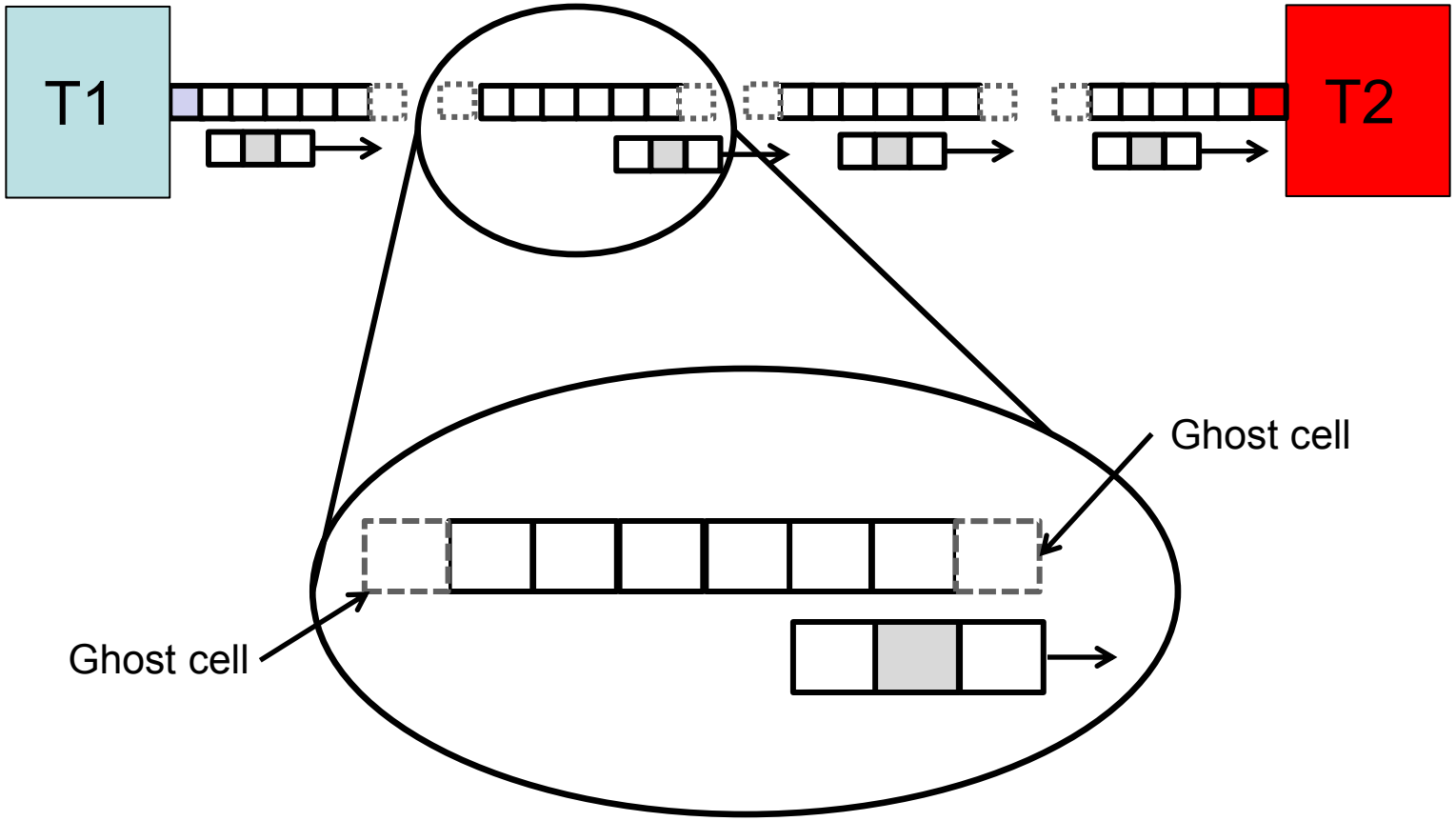
Send my "right" boundary value to the neighbor to my "right'

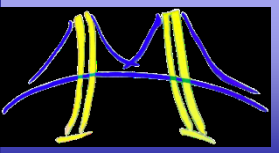
Receive my "left" ghost cell from the neighbor to my "left"



The Geometric Decomposition Pattern

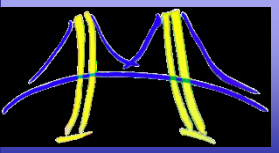
- This is an instance of a very important design pattern … the Geometric decomposition pattern.
- We will cover this pattern in more detail in a later lecture.





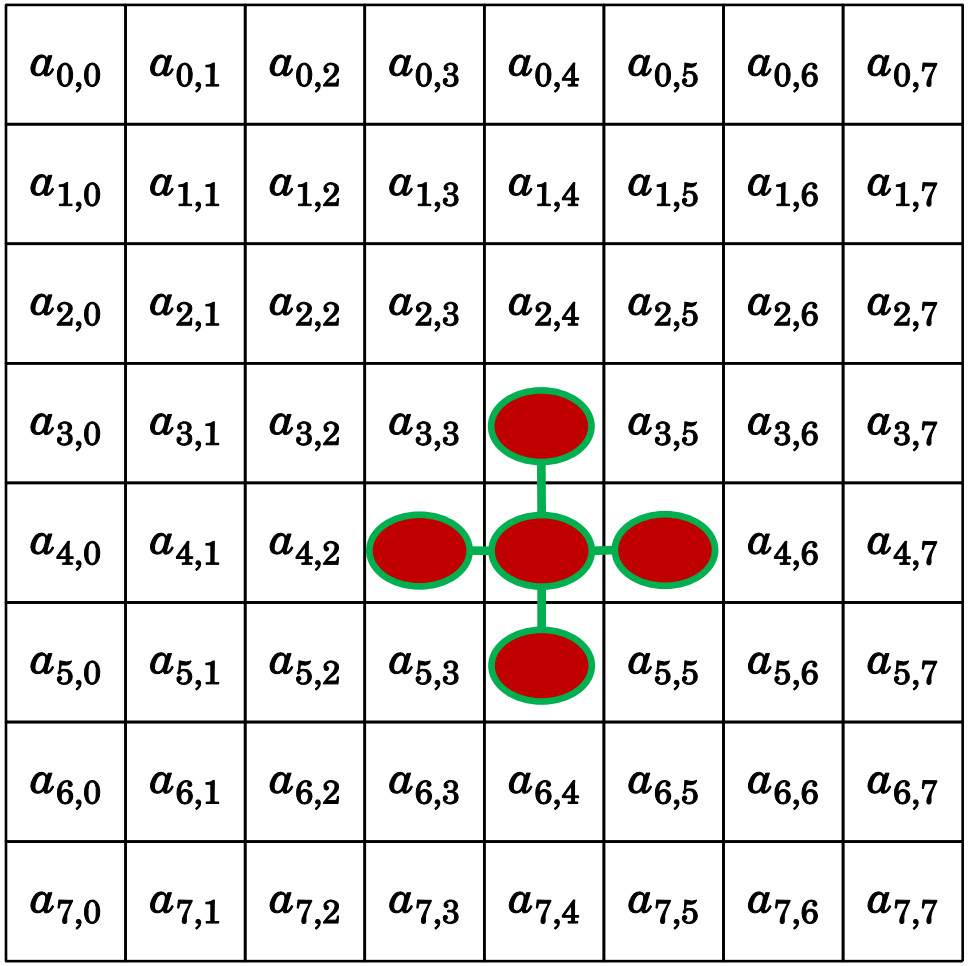
Partitioned Array Pattern

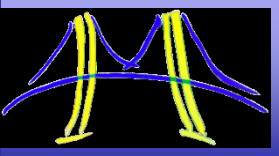
- Problem:
  - Arrays often need to be partitioned between multiple UEs. How can we do this so the resulting program is both readable and efficient?
- Forces
  - Large number of small blocks organized to balance load.
  - Able to specialize organization to different platforms/problems.
  - Understandable indexing to make programming easier.
- Solution:
  - Express algorithm in blocks
  - Abstract indexing inside mapping functions … programmer works in an index space natural to the domain, functions map into distribution needed for efficient execution.
  - The text of the pattern defines some of these common mapping functions (which can get quite confusing … and in the literature are usually left as “an exercise for the reader”).



Partitioned Arrays

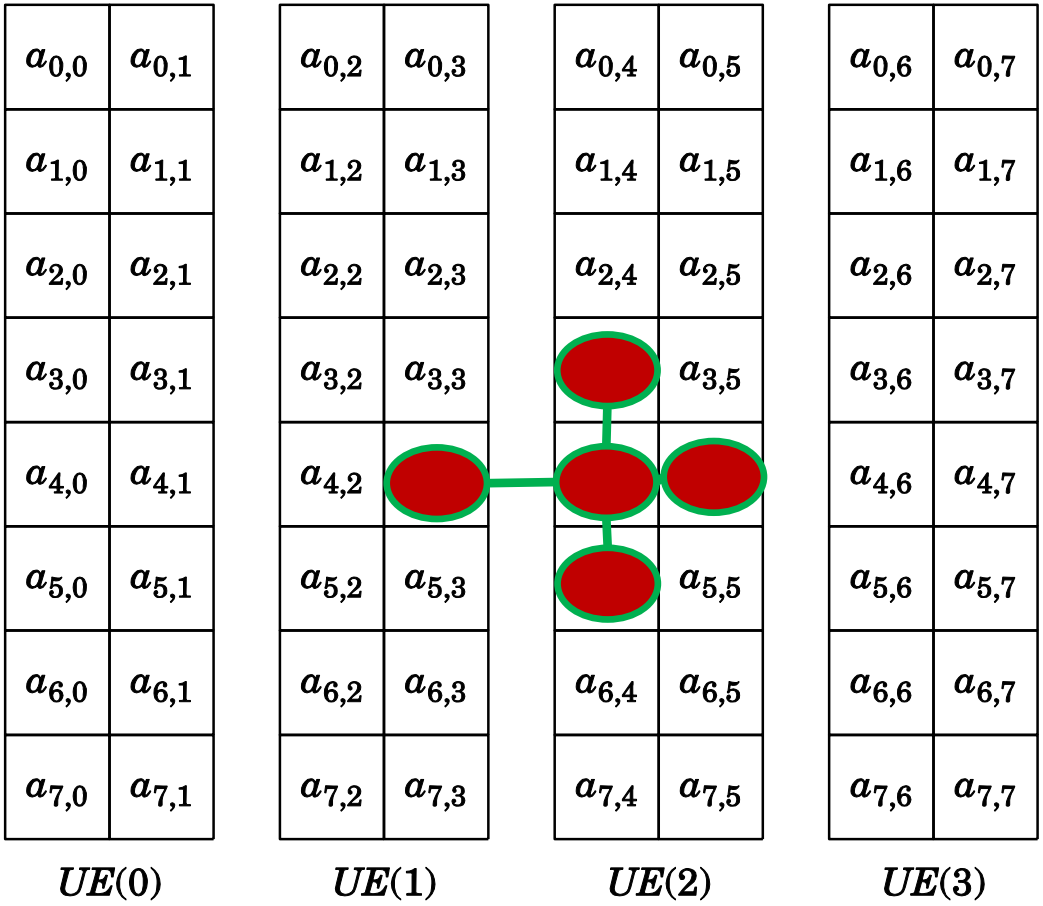
- Realistic problems are 2D or 3D; require move complex data distributions.
- We need to parallelize the computation by partitioning this index space
- Example: Consider a 2D domain over which we wish to solve a PDE using an explicit finite difference solver . The figure shows a five point stencil …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





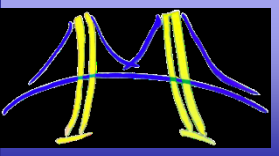
Partitioned Arrays: Column block distribution

- Split the non-unit-stride dimension (P-1) times to produce P chunks, assign the i^{th} chunk to P_i. WIth N = n * n, P = p * p
- In a 2D finite-differencing program (exchange edges), how much do we have to communicate? 2*n = **2*sqrt(N) messages** per processor



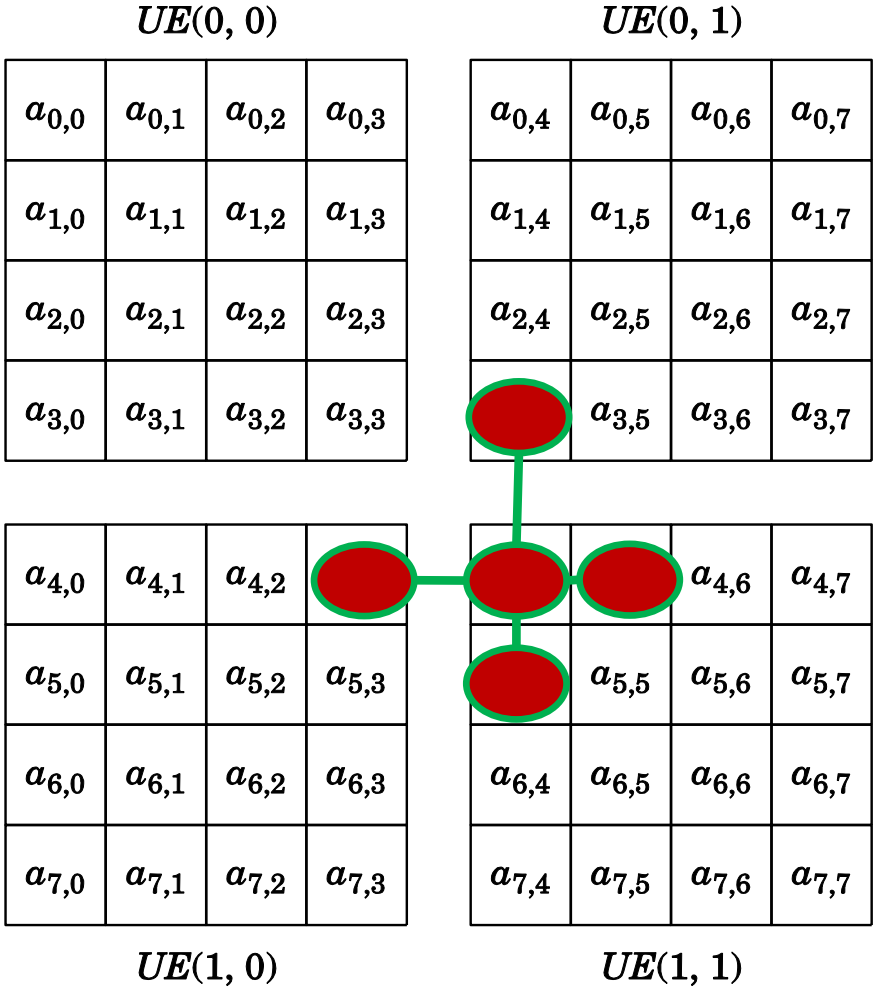
P is the # of processors

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



Partitioned Arrays: Block distribution

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

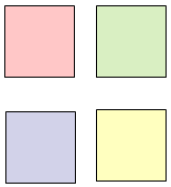


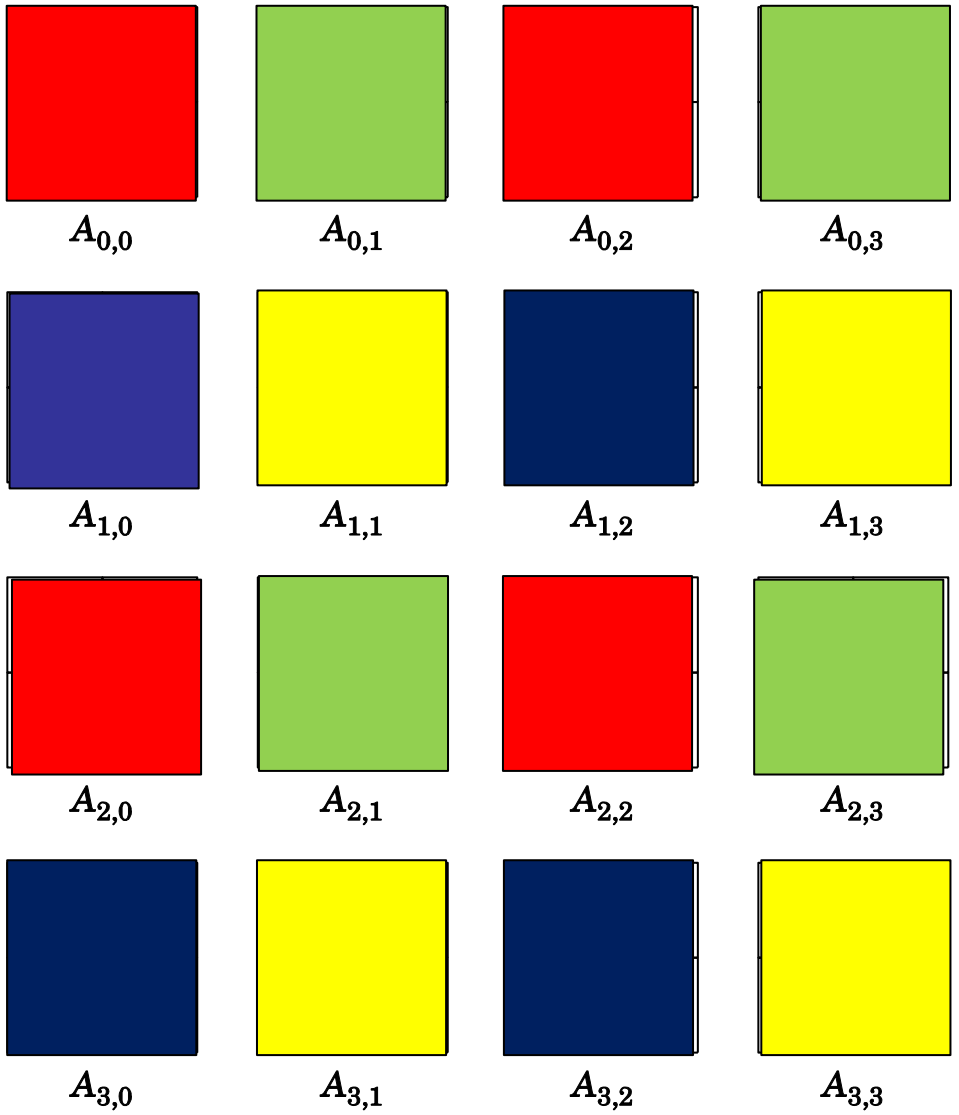
P is the # of processors

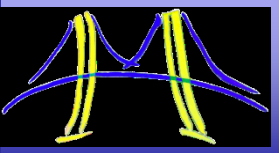
Partitioned Arrays:
block cyclic distribution

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

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

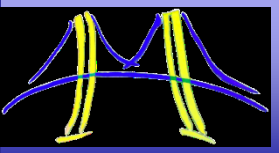




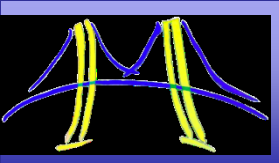


Exercise 4: Transpose

- Goal
  - Explore interaction of partitioned arrays and message passing
- Program
  - We provide a matrix transposition program … which is one of the simplest examples of a program based on partitioned arrays.
  - Notice how the SPMD pattern interacts with the partitioned array pattern.
  - Modify the program to use isend/irecv and overlap communication with local transpose to maximize aggregate bandwidth

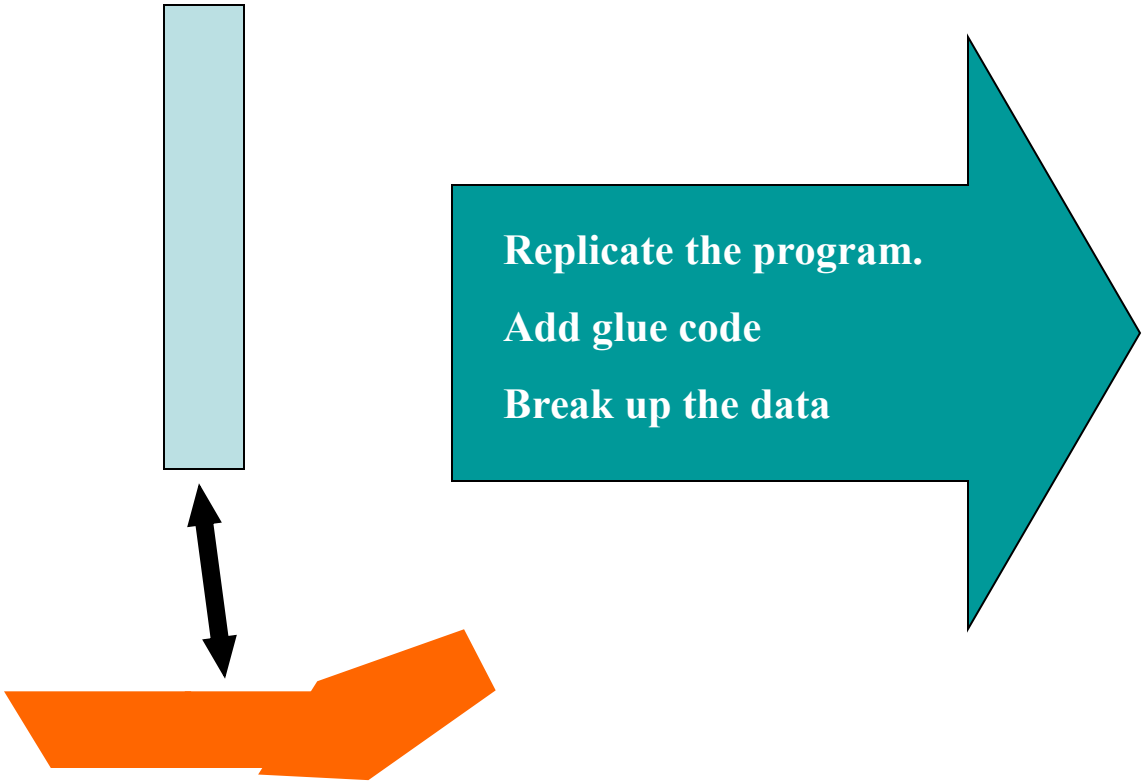


MIXING MPI AND OPENMP



How do people mix MPI and OpenMP?

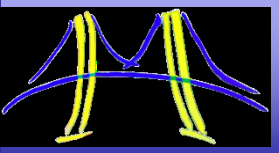
A sequential program working on a data set



•Create the MPI program with its data decomposition.

• Use OpenMP inside each MPI process.

72

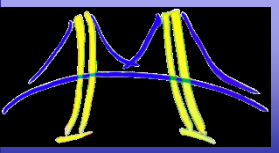


Pi program with MPI and OpenMP

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

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

73

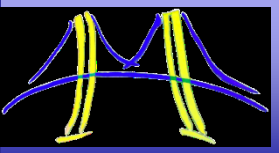


Key issues when mixing OpenMP and MPI

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

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

74

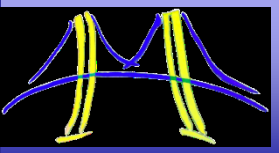


Dangerous Mixing of MPI and OpenMP

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

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

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

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

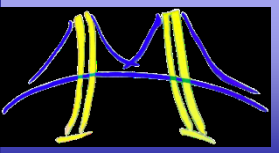

75



Messages and threads

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

76



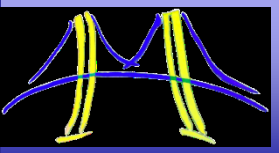
Safe Mixing of MPI and OpenMP

Put MPI in sequential regions

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

// a whole bunch of initializations

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

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

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

consume(U, mpi_id);
```

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

77



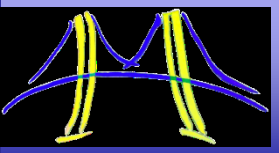
Safe Mixing of MPI and OpenMP

Protect MPI calls inside a parallel region

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

// a whole bunch of initializations

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

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

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

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

78



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

79

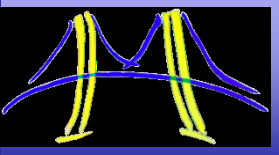
CLOSING COMMENTS

Parallel hardware trends

Top 500: total number of processors (1993-2000)

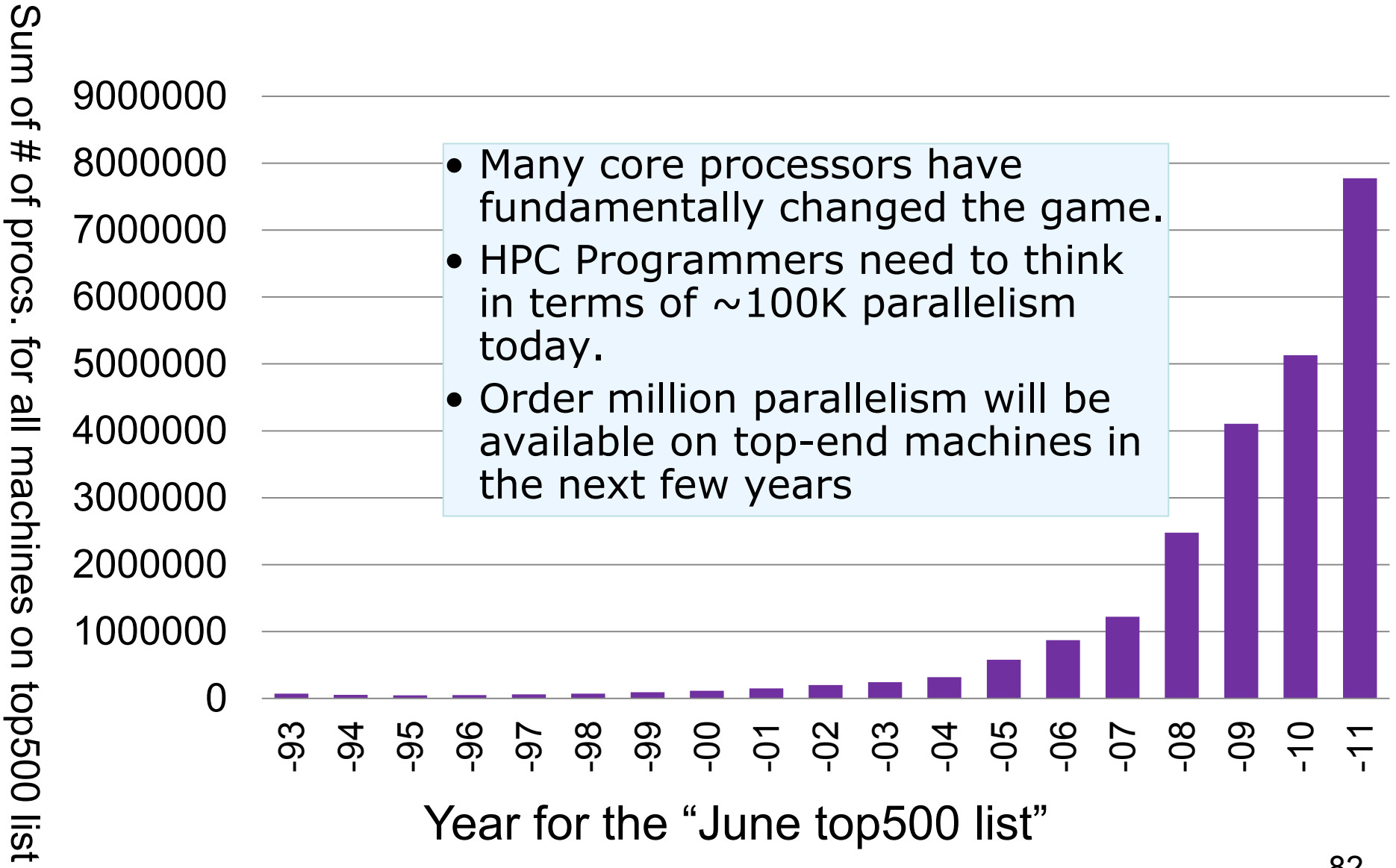
The downward trend from ‘93 to ‘95 is due to the old TMC SIMD machines failing and leaving the list.

Source: the “June lists” from www.top500.org



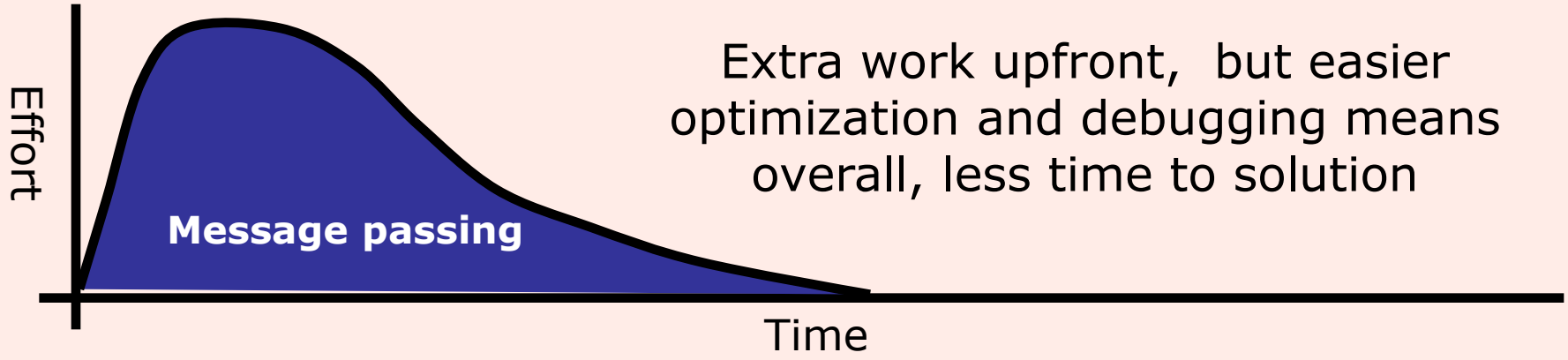
Parallel hardware trends

Top 500: total number of processors (1993-2011)

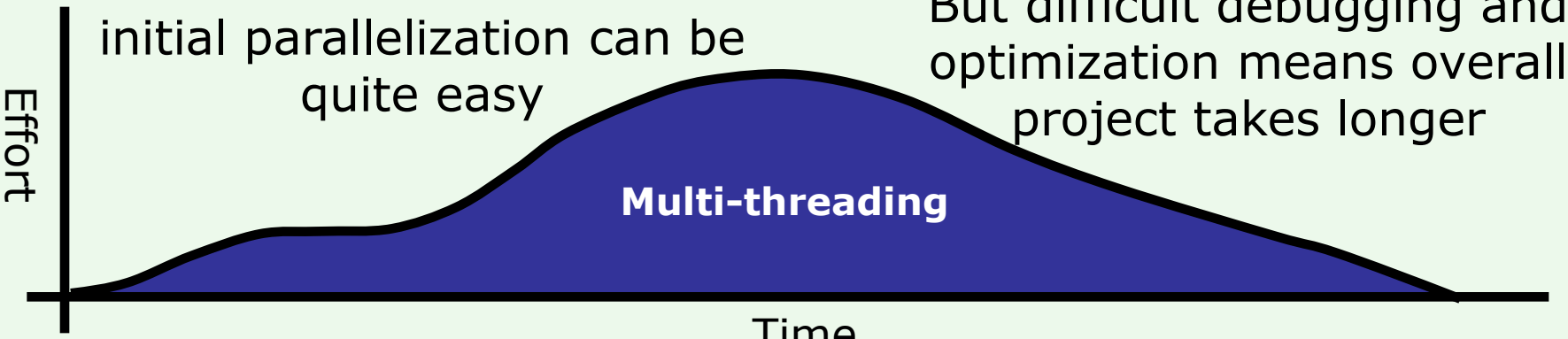


- Many core processors have fundamentally changed the game.
- HPC Programmers need to think in terms of ~100K parallelism today.
- Order million parallelism will be available on top-end machines in the next few years

Does a shared address space make programming easier?



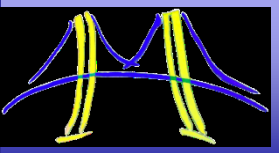
Extra work upfront, but easier optimization and debugging means overall, less time to solution



But difficult debugging and optimization means overall project takes longer

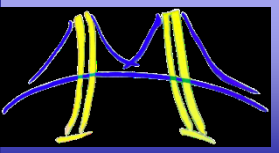
Proving that a shared address space program using semaphores is race free is an NP-complete problem*

*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, 2003



Closing comments

- Question conventional wisdom.
  - Do we really need cache coherence? If the memory hierarchy can't be hidden, isn't it better to expose the hierarchy so I can control it?
  - Debugging and Maintenance costs more than coding. So extra work up front to organize a problem to exploit the concurrency (e.g. decomposing and distributing data structures) shouldn't be such a big deal.
  - SW lives longer than HW. So why would anyone use a non-portable, non-standard programming model? That's just nuts!!
- As you move forward through the course ….
  - Notice that the patterns used in creating parallel code only weakly depend on the programming model. I can do loop parallelism with MPI, message passing with pthreads, kernel parallelism with OpenMP.
  - So learn multiple programming models and enjoy them … but don't obsess about them. Ultimately, it's the design patterns and learning how to apply them to different problems that matter.

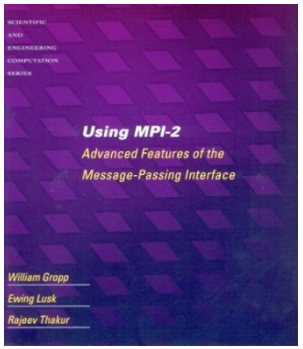


MPI References

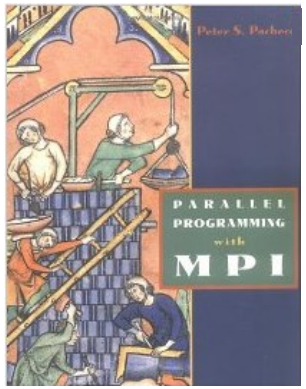
- The Standard itself:
  - at http://www.mpi-forum.org
  - All MPI official releases, in both postscript and HTML
- Other information on Web:
  - at http://www.mcs.anl.gov/mpi
  - pointers to lots of stuff, including other talks and tutorials, a FAQ, other MPI pages

Books for learning MPI

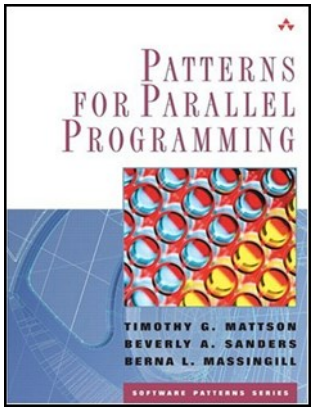
- *Using MPI-2: Portable Parallel Programming with the Message-Passing Interface*, by Gropp, Lusk, and Thakur, MIT Press, 1999..

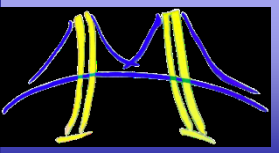


- *Parallel Programming with MPI*, by Peter Pacheco, Morgan-Kaufmann, 1997.



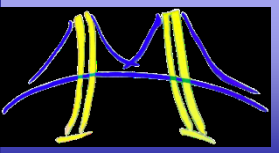
- *Patterns for Parallel Programing*, by Tim Mattson, Beverly Sanders, and Berna Massingill.





MPI core functions

```
#include <mpi.h>
int size, rank, argc, err_code=0;
char **argv;

MPI_Init (&argc, &argv);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Comm_size (MPI_COMM_WORLD, &size);
MPI_Abort(MPI_COMM_WORLD,      err_code);
MPI_Finalize();
```



Basic MPI communication

Blocking send/receive

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

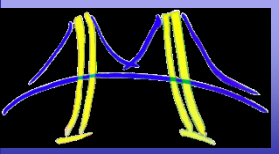
- **MPI_Recv** (void* buf, int count, MPI_Datatype datatype,
  int source, int tag, MPI_Comm comm, MPI_Status* status)

Nonblocking send/receive

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

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

- **MPI_Wait**(MPI_Request *request, MPI_Status *status)



Collective Communication

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

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

- **MPI_Sendrecv**(void *sendbuf, int sendcount,
  MPI_Datatype sendtype, int dest, int sendtag,
  void *recvbuf, int recvcount, MPI_Datatype recvtype,
  int source, int recvtag, MPI_Comm comm,
  MPI_Status *recv_status)

- **MPI_Barrier**(MPI_Datatype datatype)