What is MPI?

- A *message-passing library specification*
  - extended message-passing model
  - not a language or compiler specification
  - not a specific implementation or product
- For parallel computers, clusters, and heterogeneous networks
- Full-featured
- Designed to provide access to advanced parallel hardware for end users, library writers, and tool developers

- Credits for Slides: Rusty Lusk, Mathematics and Computer Science Division, Argonne National Laboratory

Where Did MPI Come From?

- Early vendor systems (Intel's NX, IBM's EUI, TMC's CMMD) were not portable (or very capable)
- Early portable systems (PVM, p4, TCGMSG, Chameleon) were mainly research efforts
  - Did not address the full spectrum of issues
  - Lacked vendor support
  - Were not implemented at the most efficient level
- The MPI Forum organized in 1992 with broad participation by:
  - vendors: IBM, Intel, TMC, SGI, Convex, Meiko
  - portability library writers: PVM, p4
  - users: application scientists and library writers
  - finished in 18 months

Novel Features of MPI

- Communicators encapsulate communication spaces for library safety
- Datatypes reduce copying costs and permit heterogeneity
- Multiple communication modes allow precise buffer management
- Extensive collective operations for scalable global communication
- Process topologies permit efficient process placement, user views of process layout
- Profiling interface encourages portable tools

MPI References

- The Standard itself:
  - at `http://www.mpi-forum.org`
  - All MPI official releases, in both postscript and HTML
- Books:
  - *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, 2nd Edition, by Gropp, Lusk, and Skjellum, MIT Press, 1999. Also *Using MPI-2*, w. R. Thakur
  - *MPI: The Complete Reference,* 2 vols, MIT Press, 1999.
  - *Designing and Building Parallel Programs*, by Ian Foster, Addison-Wesley, 1995.
  - *Parallel Programming with MPI*, by Peter Pacheco, Morgan-Kaufmann, 1997.
- 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

Compiling and Running MPI Programs

- To compile and run MPI programs one uses special commands
  - mpicc compiles and includes the MPI libraries
  - mpirun sets up environment variables for runing
    - mpirun –np # prog
- One can also configure the set of nodes to be used
- For details on this and on user level configuration of the 2 MPI versions MPICH and LAM see the references in http://discov.cs.kent.edu/resources/doc/mpiref.htm
- For examples from Pachero see http://nexus.cs.usfca.edu/mpi/

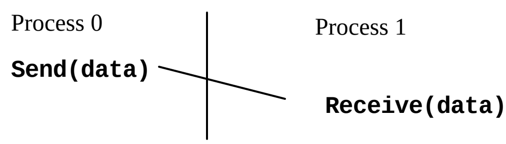
Hello (C)

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

int main( argc, argv )
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( "I am %d of %d\n", rank, size );
    MPI_Finalize();
    return 0;
}
```

MPI Basic Send/Receive

- We need to fill in the details in



- Things that need specifying:
  - How will "data" be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?

Some Basic Concepts

- Processes can be collected into groups
- Each message is sent in a context, and must be received in the same context
  - Provides necessary support for libraries
- A group and context together form a communicator
- A process is identified by its rank in the group associated with a communicator
- There is a default communicator whose group contains all initial processes, called **`MPI_COMM_WORLD`**

MPI Datatypes

- The data in a message to send or receive is described by a triple (address, count, datatype), where
- An MPI *datatype* is recursively defined as:
  - predefined, corresponding to a data type from the language (e.g., MPI_INT, MPI_DOUBLE)
  - a contiguous array of MPI datatypes
  - a strided block of datatypes
  - an indexed array of blocks of datatypes
  - an arbitrary structure of datatypes
- There are MPI functions to construct custom datatypes, in particular ones for subarrays

MPI Tags

- Messages are sent with an accompanying user-defined integer *tag*, to assist the receiving process in identifying the message
- Messages can be screened at the receiving end by specifying a specific tag, or not screened by specifying **`MPI_ANY_TAG`** as the tag in a receive
- Some non-MPI message-passing systems have called tags "message types". MPI calls them tags to avoid confusion with datatypes

MPI Basic (Blocking) Send

MPI_SEND(start, count, datatype, dest, tag, comm)

- The message buffer is described by (**`start, count, datatype`**).
- The target process is specified by **`dest`**, which is the rank of the target process in the communicator specified by **`comm`**.
- When this function returns, the data has been delivered to the system and the buffer can be reused. The message may not have been received by the target process.

MPI Basic (Blocking) Receive

MPI_RECV(start, count, datatype, source, tag, comm, status)

- Waits until a matching (both **`source`** and **`tag`**) message is received from the system, and the buffer can be used
- **`source`** is rank in communicator specified by **`comm`**, or **`MPI_ANY_SOURCE`**
- **`tag`** is a tag to be matched on or **`MPI_ANY_TAG`**
- receiving fewer than **`count`** occurrences of **`datatype`** is OK, but receiving more is an error
- **`status`** contains further information (e.g. size of message)

MPI is Simple

- Many parallel programs can be written using just these six functions, only two of which are non-trivial:
  - **MPI_INIT**
  - **MPI_FINALIZE**
  - **MPI_COMM_SIZE**
  - **MPI_COMM_RANK**
  - **MPI_SEND**
  - **MPI_RECV**

Collective Operations in MPI

- Collective operations are called by all processes in a communicator
- **MPI_BCAST** distributes data from one process (the root) to all others in a communicator
  - MPI_Bcast (buffer, count, datatype, root, comm);
- **MPI_REDUCE** combines data from all processes in communicator and returns it to one process
  - MPI_Reduce(sendbuf, recvbuf, count, datatype, operation, root, comm);
- In many numerical algorithms, **SEND/RECEIVE** can be replaced by **BCAST/REDUCE**, improving both simplicity and efficiency

Example: PI in C - 1

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
   int done = 0, n, myid, numprocs, i, rc;
   double PI25DT = 3.141592653589793238462643;
   double mypi, pi, h, sum, x, a;
   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);
   while (!done)  {
     if (myid == 0) {
       printf("Enter the number of intervals: (0 quits) ");
       scanf("%d",&n);
     }
     MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
     if (n == 0) break;
```


Example: PI in C - 2

```
    h   = 1.0 / (double) n;
     sum = 0.0;
     for (i = myid + 1; i <= n; i += numprocs) {
       x = h * ((double)i - 0.5);
       sum += 4.0 / (1.0 + x*x);
     }
     mypi = h * sum;
     MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                 MPI_COMM_WORLD);
     if (myid == 0)
       printf("pi is approximately %.16f, Error is .16f\n",
               pi, fabs(pi - PI25DT));
   }
   MPI_Finalize();
  return 0;
}
```

Alternative Set of 6 Functions

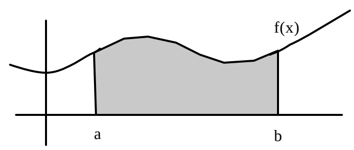
- Using collectives:
  - **`MPI_INIT`**
  - **`MPI_FINALIZE`**
  - **`MPI_COMM_SIZE`**
  - **`MPI_COMM_RANK`**
  - **`MPI_BCAST`**
  - **`MPI_REDUCE`**

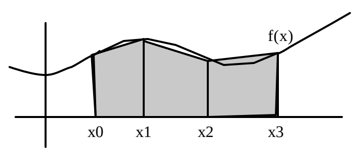
Exercises

- Modify hello program so that each process sends the name of the machine it is running on to process 0, which prints it.
  - See source of cpi or fpi in mpich/examples/basic for how to use **`MPI_Get_processor_name`**
- Do this in such a way that the hosts are printed in rank order

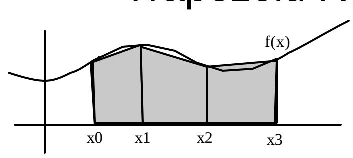
Trapezoid Rule

- Numerical Integration (Quadrature)
  - approximate the area under the curve by calculating the area of rectangles (the Rectangle Rule) or trapezoids (the Trapezoidal Rule) that fit close to the curve.





Trapezoid Rule



- The base of each trapezoid is h = x1-x0 = x2-x1 etc.
- The area formed by one trapezoid is
  - area of one trapezoid = ½ * h * (f(left) + f(right))
- The area under the curve is:

  Area = ½ * h * (f(x0) + f(x1)) + ½ * h * (f(x1) + f(x2)) + ½ * h * (f(x2) + f(x3))
- which simplifies to

  Area = h * { ½f(x0) + f(x1) + f(x2) + ½f(x3) }

Parallelizing Trapezoid Rule

- Divide interval [a,b] into np parts, one for each processor.
- Each processor performs the trapezoidal rule on its part.

Serial and Parallel Versions

- Serial
- Parallel

Adaptive Quadrature

- Adaptive quadrature allows the program to calculate the new value for the integral with a different number of trapezoids eac h time.
- The program terminates when the final result is "close enough".
- Pseudocode for a sequential program:
- new = 1;
- diff = 100000;
- numtraps = 1;
- limit = 0.001;
- while ((diff > limit) && (numtraps < 2048)) {
- old = new;
- numtraps = numtraps*2;
- calculate (new) ;
- diff = abs((new-old)) / new;
- }
- print(new);

Dot products – Block Decomposition

{ a0 a1 a2 a3 a4 a5 a6 a7 } x { b0 b1 b2 b3 b4 b5b 6 b7 } = a0*b0 + a1*b1 + … + a6*b6 + a7*b7

- Serial
- Parallel
- Parallel with Allreduce

Matrix- Vector Multiplication – version 1

a00 a01 a02 a03 a04 a05 a06 a07
a10 a11 a12 a03 a04 a05 a06 a07
a20 a21 a22 …
a30 a31 a32 …
a40 a41 a42 …
a50 a51 a52 …
a60 a61 a62 …
a70 a71 a72 a73 a74 a75 a76 a77

x

b0
b1
b2
b3
b4
b5
b6
b7

c0
c1
c2
c3
c4
c5
c6
c7

- Block-row distribution of the matrix
- Copy of vector on every process
- Each process calculates its corresponding portion of the result vector

How to get the data to where needed

- If the matrix is located in a single process at the start, can use MPI_Scatter to send the rows to all processes.
- (Watch out for how the matrix is stored – in C it is row-major!)
  - MPI_Scatter(
  - void* send_data,
  - int send_count,
  - MPI_Datatype send_type,
  - void* recv_data,
  - int recv_count,
  - MPI_Datatype recv_type,
  - int root,
  - MPI_Comm comm);

Vector Example:

```
/* data starts at process 0 */
float vector[8], local_vector[2];
…
MPI_Scatter( vector, 2, MPI_FLOAT,
             local_vector, 2, MPI_FLOAT,
             0, MPI_COMM_WORLD);
```

would send 2 elements to each process and store them into local_vector;

- If the vector is initially distributed in block fashion among all processes, can use MPI_Gather to get a copy of the whole vector into the root process.
  - MPI_Gather(
  - void* send_data,
  - int send_count,
  - MPI_Datatype send_type,
  - void* recv_data,
  - int recv_count,
  - MPI_Datatype recv_type,
  - int root,
  - MPI_Comm comm);

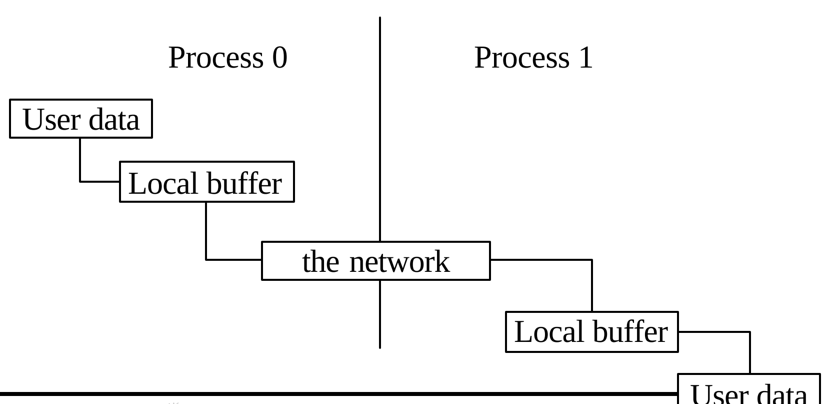
- If the vector is initially distributed in block fashion among all processes, can use MPI_Allgather to get a copy of the whole vector into the **every** process.
  - MPI_Allgather(
  - void* send_data,
  - int send_count,
  - MPI_Datatype send_type,
  - void* recv_data,
  - int recv_count,
  - MPI_Datatype recv_type,
  - MPI_Comm comm);

C Versions of Matrix-Vector Multiply

- Serial
- Parallel

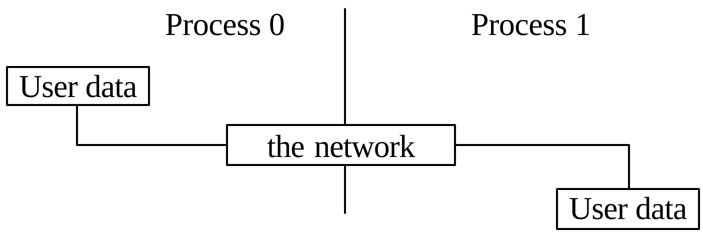
Buffers

- When you send data, where does it go? One possibility is:



Avoiding Buffering

- It is better to avoid copies:



This requires that **`MPI_Send`** wait on delivery, or that **`MPI_Send`** return before transfer is complete, and we wait later.

Blocking and Non-blocking Communication

- So far we have been using *blocking* communication:
  - `MPI_Recv` does not complete until the buffer is full (available for use).
  - `MPI_Send` does not complete until the buffer is empty (available for use).
- Completion depends on size of message and amount of system buffering.

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 is called "unsafe" because it depends on the availability of system buffers

Some Solutions to the "unsafe" 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)`**

More Solutions to the “unsafe” Problem

- Supply own space as buffer for send

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**

MPI’s Non-blocking Operations

- Non-blocking operations return (immediately) “request handles” that can be tested and waited on.

```
MPI_Isend(start, count, datatype,
     dest, tag, comm, request)
MPI_Irecv(start, count, datatype,
     dest, tag, comm, request)
MPI_Wait(&request, &status)
```

- One can also test without waiting:

```
MPI_Test(&request, &flag, status)
```

Multiple Completions

- It is sometimes desirable to wait on multiple requests:

```
MPI_Waitall(count, array_of_requests,
   array_of_statuses)
MPI_Waitany(count, array_of_requests,
   &index, &status)
MPI_Waitsome(count, array_of_requests,
   array_of indices, array_of_statuses)
```

- There are corresponding versions of **`test`** for each of these.

Communication Modes

- MPI provides multiple *modes* for sending messages:
  - Synchronous mode (**`MPI_Ssend`**): the send does not complete until a matching receive has begun. (Unsafe programs deadlock.)
  - Buffered mode (**`MPI_Bsend`**): the user supplies a buffer to the system for its use. (User allocates enough memory to make an unsafe program safe.
  - Ready mode (**`MPI_Rsend`**): user guarantees that a matching receive has been posted.
    - Allows access to fast protocols
    - undefined behavior if matching receive not posted
- Non-blocking versions (**`MPI_Issend`**, etc.)
- **`MPI_Recv`** receives messages sent in any mode.

Other Point-to Point Features

- **`MPI_Sendrecv`**
- **`MPI_Sendrecv_replace`**
- **`MPI_Cancel`**
  - Useful for multibuffering
- Persistent requests
  - Useful for repeated communication patterns
  - Some systems can exploit to reduce latency and increase performance

MPI_Sendrecv

- Allows simultaneous send and receive
- Everything else is general.
  - Send and receive datatypes (even type signatures) may be different
  - Can use `Sendrecv` with plain Send or `Recv` (or `Irecv` or Ssend_init, …)
  - More general than "send left"

Process 0	Process 1
`SendRecv(1)`	**`SendRecv(0)`**

Collective Operations in MPI

- Collective operations must be called by all processes in a communicator.
- **`MPI_BCAST`** distributes data from one process (the root) to all others in a communicator.
- **`MPI_REDUCE`** combines data from all processes in communicator and returns it to one process.
- In many numerical algorithms, **`SEND/RECEIVE`** can be replaced by **`BCAST/REDUCE`**, improving both simplicity and efficiency.

MPI Collective Communication

- Communication and computation is coordinated among a group of processes in a communicator.
- Groups and communicators can be constructed “by hand” or using topology routines.
- Tags are not used; different communicators deliver similar functionality.
- No non-blocking collective operations.
- Three classes of operations: synchronization, data movement, collective computation.

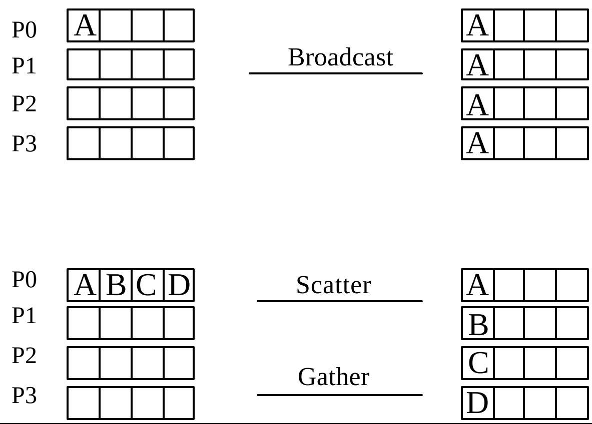
Synchronization

- `MPI_Barrier( comm )`
- Blocks until all processes in the group of the communicator **`comm`** call it.

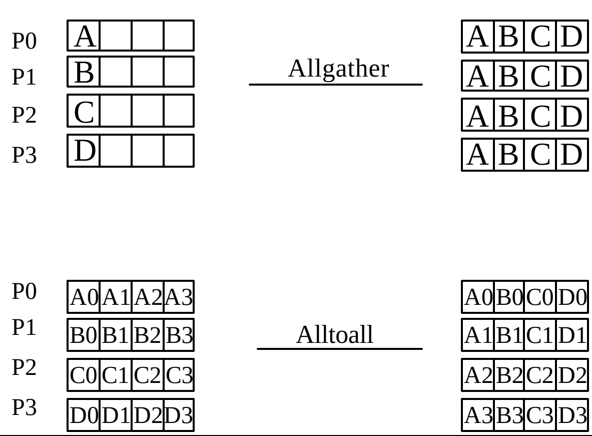
Synchronization

- `MPI_Barrier( comm, ierr )`
- Blocks until all processes in the group of the communicator **`comm`** call it.

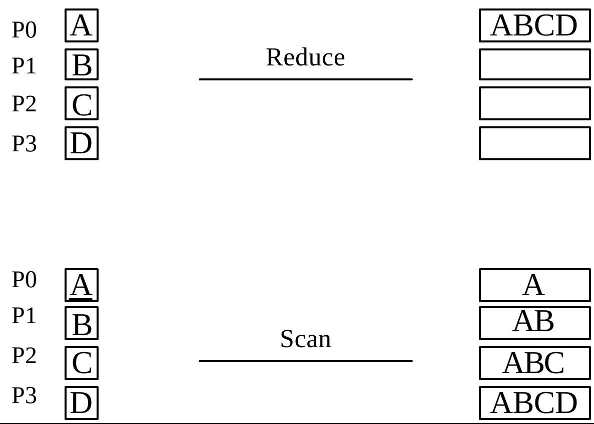
Collective Data Movement



More Collective Data Movement



Collective Computation



MPI Collective Routines

- Many Routines: **`Allgather, Allgatherv, Allreduce, Alltoall, Alltoallv, Bcast, Gather, Gatherv, Reduce, Reduce_scatter, Scan, Scatter, Scatterv`**
- **`All`** versions deliver results to all participating processes.
- V versions allow the hunks to have different sizes.
- **`Allreduce`**, **`Reduce`**, **`Reduce_scatter`**, and **`Scan`** take both built-in and user-defined combiner functions.

MPI Built-in Collective Computation Operations

• **`MPI_Max`**	Maximum
• **`MPI_Min`**	Minimum
• **`MPI_Prod`**	Product
• **`MPI_Sum`**	Sum
• **`MPI_Land`**	Logical and
• **`MPI_Lor`**	Logical or
• **`MPI_Lxor`**	Logical exclusive or
• **`MPI_Band`**	Binary and
• **`MPI_Bor`**	Binary or
• **`MPI_Bxor`**	Binary exclusive or
• **`MPI_Maxloc`**	Maximum and location
• **`MPI_Minloc`**	Minimum and location

How Deterministic are Collective Computations?

- In exact arithmetic, you always get the same results
  - but roundoff error, truncation can happen
- MPI does *not* require that the same input give the same output
  - Implementations are encouraged but not required to provide *exactly* the same output given the same input
  - Round-off error may cause slight differences
- Allreduce does guarantee that the *same* value is received by all processes for each call
- Why didn't MPI mandate determinism?
  - Not all applications need it
  - Implementations can use "deferred synchronization" ideas to provide better performance

Defining your own Collective Operations

- Create your own collective computations with:
  MPI_Op_create(user_fcn, commutes, &op);
  MPI_Op_free(&op);

  user_fcn(invec, inoutvec, len, datatype);
- The user function should perform:

  inoutvec[i] = invec[i] op inoutvec[i];

  for i from 0 to len-1.
- The user function can be non-commutative.

Blocking and Non-blocking

- Blocking
  - `MPI_Recv` does not complete until the buffer is full (available for use).
  - `MPI_Send` does not complete until the buffer is empty (available for use).
- Non-blocking operations return (immediately) "request handles" that can be tested and waited on.

```
MPI_Isend(start, count, datatype, dest, tag, comm,
 request)
MPI_Irecv(start, count, datatype, dest, tag, comm,
 request)
MPI_Wait(&request, &status)
```

  - One can also test without waiting:

```
MPI_Test(&request, &flag, status)
```

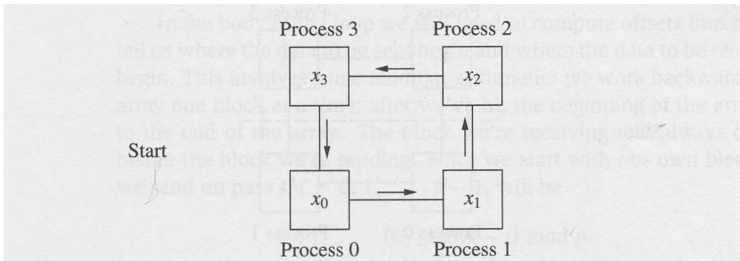
Persistent Requests

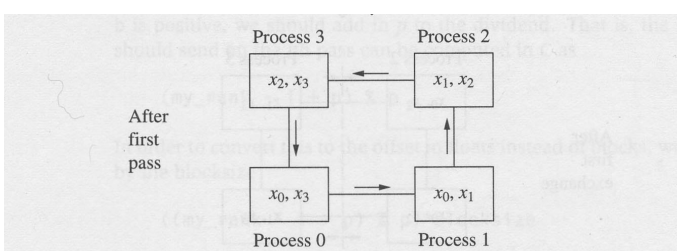
- Persistent requests
  - Useful for repeated communication patterns
  - Some systems can exploit to reduce latency and increase performance

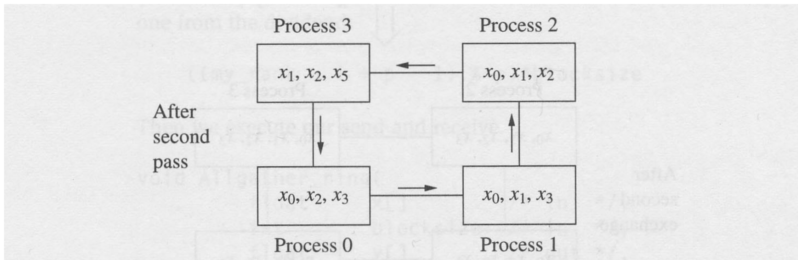
Communication Modes

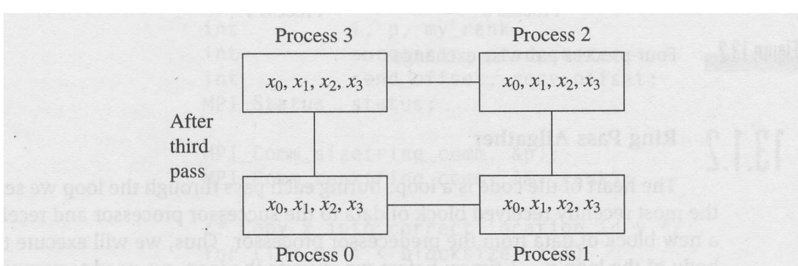
- MPI provides multiple *modes* for sending messages:
  - Synchronous mode (**`MPI_Ssend`**): the send does not complete until a matching receive has begun. (Unsafe programs deadlock.)
  - Buffered mode (**`MPI_Bsend`**): the user supplies a buffer to the system for its use. (User allocates enough memory to make an unsafe program safe.
  - Ready mode (**`MPI_Rsend`**): user guarantees that a matching receive has been posted.
    - Allows access to fast protocols
    - undefined behavior if matching receive not posted
- Non-blocking versions (**`MPI_Issend`**, etc.)
- **`MPI_Recv`** receives messages sent in any mode.

Ring Based Allgather






Process 3
Process 2
x_1, x_2, x_5
x_0, x_1, x_2
After
second
pass
x_0, x_2, x_3
x_0, x_1, x_3
Process 0
Process 1


Process 3
Process 2
x_0, x_1, x_2, x_3
x_0, x_1, x_2, x_3
After
third
pass
x_0, x_1, x_2, x_3
x_0, x_1, x_2, x_3
Process 0
Process 1

Advanced Communication Examples

- All_gather Ring
  - Blocking
  - Nonblocking
  - Persistent
  - Synchronous
  - Ready
  - Buffered
- Examples

MPICH Goals

- Complete MPI implementation
- Portable to all platforms supporting the message-passing model
- High performance on high-performance hardware
- As a research project:
  - exploring tradeoff between portability and performance
  - removal of performance gap between user level (MPI) and hardware capabilities
- As a software project:
  - a useful free implementation for most machines
  - a starting point for vendor proprietary implementations

MPICH Architecture

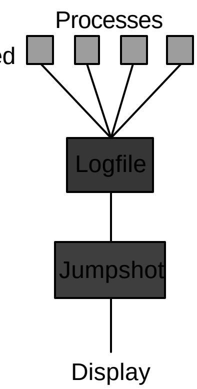
- Most code is completely portable
- An “Abstract Device” defines the communication layer
- The abstract device can have widely varying instantiations, using:
  - sockets
  - shared memory
  - other special interfaces
    - e.g. Myrinet, Quadrics, InfiniBand, Grid protocols

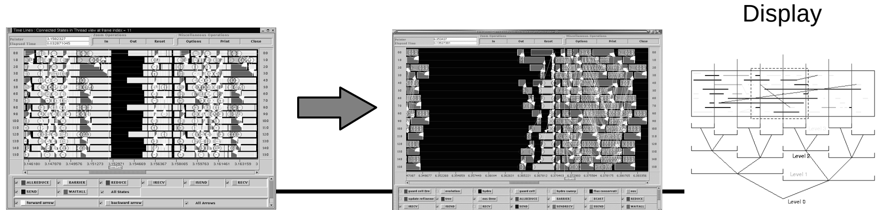
Getting MPICH for your cluster

- http://www.mcs.anl.gov/mpi/mpich
- Either MPICH-1 or
- MPICH-2

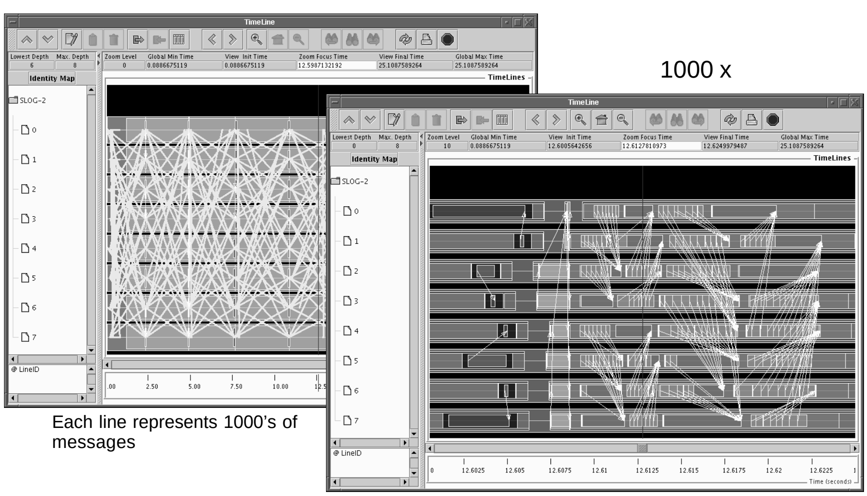
Performance Visualization with Jumpshot

- For detailed analysis of parallel program behavior, timestamped events are collected into a log file during the run.
- A separate display program (Jumpshot) aids the user in conducting a post mortem analysis of program behavior.





Using Jumpshot to look at FLASH at multiple Scales



1000 x

Each line represents 1000's of messages

Detailed view shows opportunities for optimization

What's in MPI-2

- Extensions to the message-passing model
  - Dynamic process management
  - One-sided operations (remote memory access)
  - Parallel I/O
  - Thread support
- Making MPI more robust and convenient
  - C++ and Fortran 90 bindings
  - External interfaces, handlers
  - Extended collective operations
  - Language interoperability

MPI as a Setting for Parallel I/O

- Writing is like sending and reading is like receiving
- Any parallel I/O system will need:
  - collective operations
  - user-defined datatypes to describe both memory and file layout
  - communicators to separate application-level message passing from I/O-related message passing
  - non-blocking operations
- I.e., lots of MPI-like machinery

MPI-2 Status

- Many vendors have partial implementations, especially I/O
- MPICH2 is nearly complete, not completely tested
- Expect completion by Thanksgiving

Some Research Areas

- MPI-2 RMA interface
  - Can we get high performance?
- Fault Tolerance and MPI
  - Are intercommunicators enough?
- MPI on 64K processors
  - Umm…how do we make this work :)?
  - Reinterpreting the MPI “process”
- MPI as system software infrastructure
  - With dynamic processes and fault tolerance, can we build services on MPI?

High-Level Programming With MPI

- MPI was designed from the beginning to support libraries
- Many libraries exist, both open source and commercial
- Sophisticated numerical programs can be built using libraries
  - Solve a PDE (e.g., PETSc)
  - Scalable I/O of data to a community standard file format

Higher Level I/O Libraries

- Scientific applications work with structured data and desire more self-describing file formats
- netCDF and HDF5 are two popular “higher level” I/O libraries
  - Abstract away details of file layout
  - Provide standard, portable file formats
  - Include metadata describing contents
- For parallel machines, these should be built on top of MPI-IO

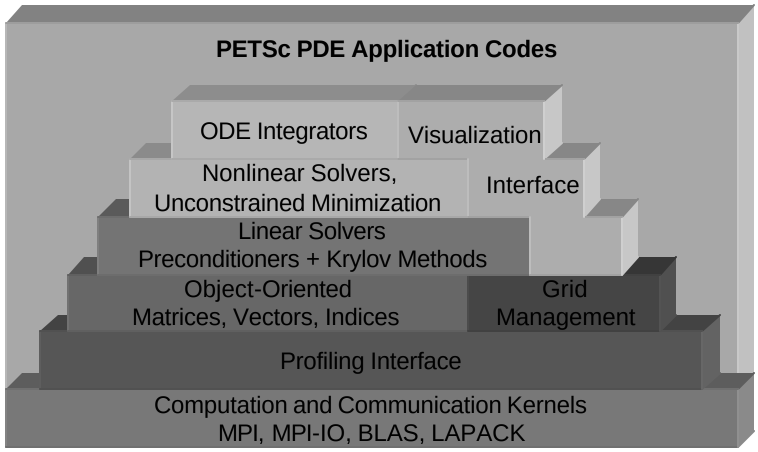
Exercise

- Jacobi problem in 2 dimensions with 1-D decomposition
  - Explained in class
  - Simple version – fixed number of iterations
  - Fancy version – test for convergence

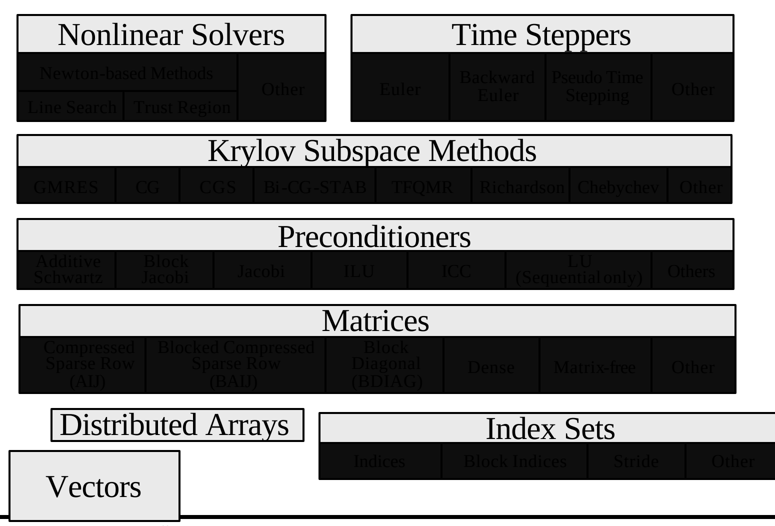
The PETSc Library

- PETSc provides routines for the parallel solution of systems of equations that arise from the discretization of PDEs
  - Linear systems
  - Nonlinear systems
  - Time evolution
- PETSc also provides routines for
  - Sparse matrix assembly
  - Distributed arrays
  - General scatter/gather (e.g., for unstructured grids)

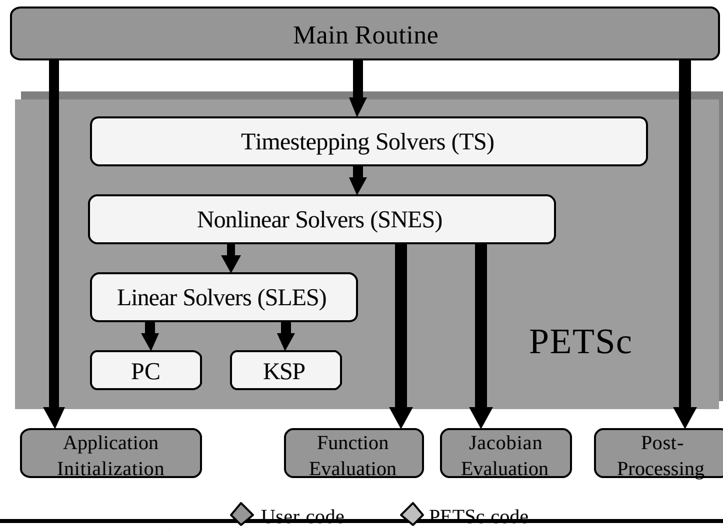
Structure of PETSc



PETSc Numerical Components



Flow of Control for PDE Solution



Poisson Solver in PETSc

- The following 7 slides show a complete 2-d Poisson solver in PETSc. Features of this solver:
  - Fully parallel
  - 2-d decomposition of the 2 -d mesh
  - Linear system described as a sparse matrix; user can select many different sparse data structures
  - Linear system solved with any user-selected Krylov iterative method and preconditioner provided by PETSc, including GMRES with ILU, BiCGstab with Additive Schwarz, etc.
  - Complete performance analysis built-in
- Only 7 slides of code!