Advanced MPI

John Urbanic
Parallel Computing Specialist
Pittsburgh Supercomputing Center

MPI Advanced Features

In the MPI Basics talk we only touched upon the 120+ MPI-1 routines and we didn't use any MPI-2. Here we will discuss some of the advanced features that you are likely to use.

- Non-Blocking Communications
- Communicators
- User Defined Data Types
- Single-sided Communications
- Scatter/Gather
- Dynamic Process Management
- MPI-IO
- Wtime
- Other languages
- MPI 3.0

MPI Evolution

1993: Supercomputing 93 - draft MPI standard presented.

1994: Final version of MPI-1.0 released

1995: MPI-1.1

1996: MPI-2

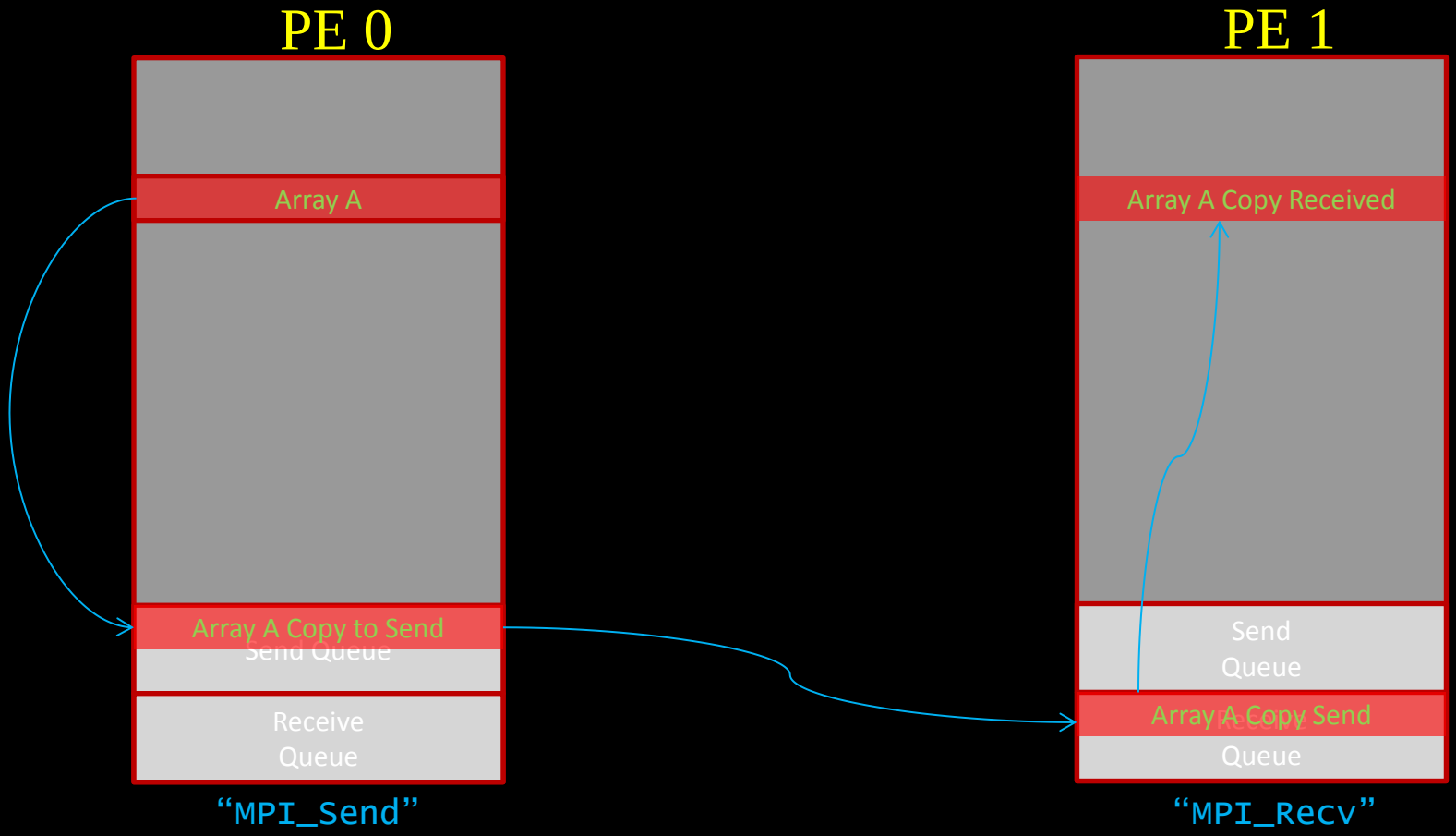
1997: MPI-1.2

2008: MPI-1.3, MPI-2.1

2009: MPI-2.2

2012: MPI-3.0 standard approved

Default MPI Messaging



Non-Blocking Messaging: Deadlocks & Performance

We didn't really give non-blocking messaging it's fair shake in the MPI Basics talk. Non-blocking sends and receives can make sending massive numbers of messages, or large messages, possible without deadlock or buffer overruns.

Imagine how this piece of code behaves as the message size becomes very large:

```
if(rank==0{
        MPI_Send(x to process 1)
        MPI_Recv(y from process 1)
}
if(rank==1){
         MPI_Send(y to process 0);
         MPI_Recv(x from process 0);
}
```

- At some point, the Send() will block until the other PE can absorb it. Both sends end up blocking each other in a deadlock.

- Also very important is the fact that *MPI_Isend()* does not need to make an extra copy to protect the integrity of any not-quite-sent data. This can be a huge win.

Non-Blocking Routines

As we said before, it is no big deal to modify your typical blocking algorithm to do this, and you may find yourself having to do so if your growing dataset starts causing buffer and memory problems.

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

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

int MPI_Test(MPI_Request *request, int *flag, MPI_Status *status)

int MPI_Wait(MPI_Request *request, MPI_Status *status)
```

```
Also:
MPI_Waitall()
MPI_Waitany()
MPI_Waitsome()
MPI_Testall()
MPI_Testany()
MPI_Testsome()
```

Non-Blocking Shifter

Here is our “shifter” exercise done with non-blocking routines.

```
#include "mpi.h"

main(int argc, char **argv){

  int size, rank, send, value_sent, value_recieved, tag=5;
  MPI_Request reqs[2];

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  send = rank+1;
  if (rank == size-1) send = 0;

  value_sent = rank+100;

  MPI_Irecv(&value_recieved, 1, MPI_INT, MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, &reqs[0]);
  MPI_Isend(&value_sent, 1, MPI_INT, send, tag, MPI_COMM_WORLD, &reqs[1]);

  /* We _could_ do things here while we wait and poll with MPI_Test() */

  MPI_Waitall(2, reqs, MPI_STATUSES_IGNORE);

  printf("PE: %d, Value %d\n",rank,value_recieved);

  MPI_Finalize();

}
```

Mixing Blocking and Non-Blocking Messaging

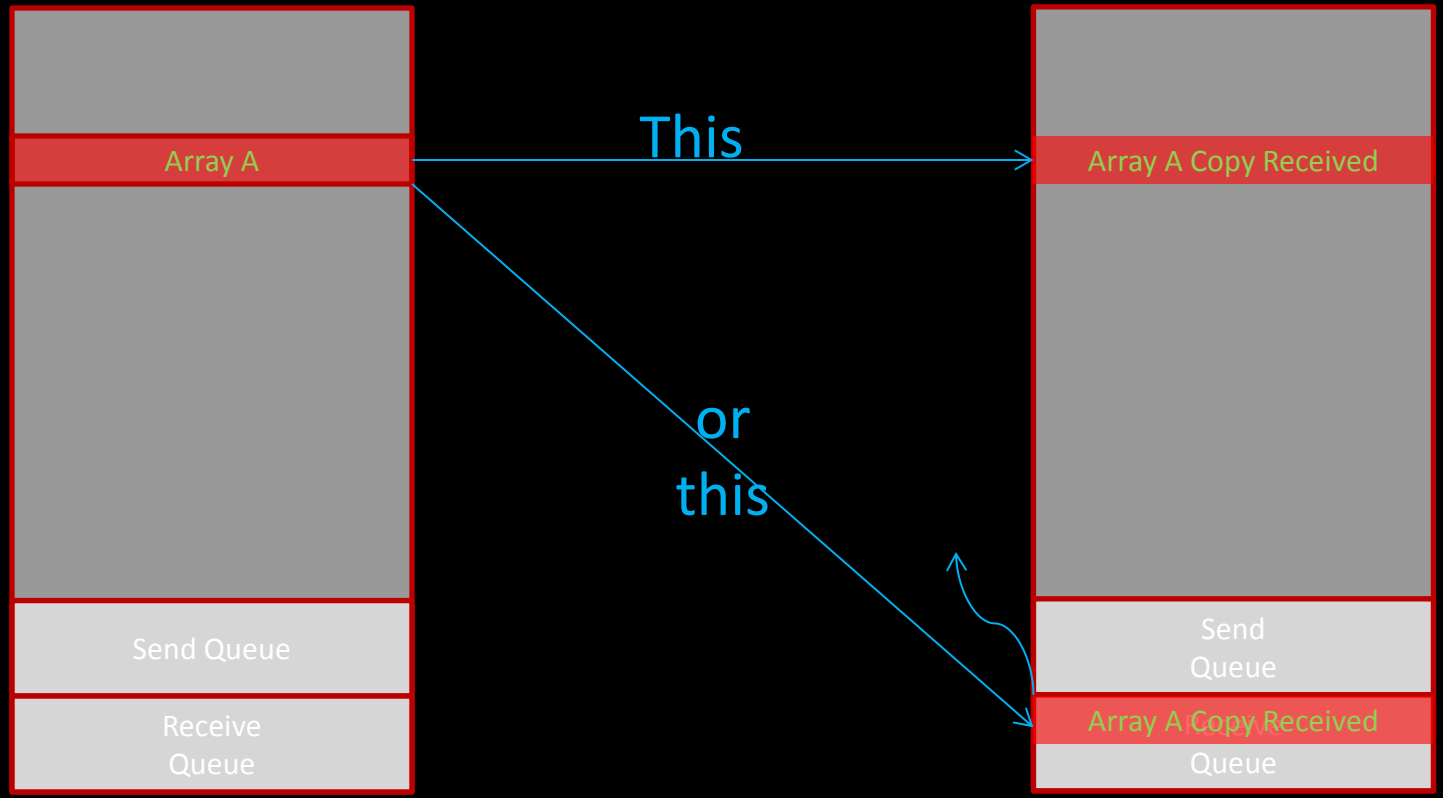
You can mix and match blocking and non-blocking routines:

```
if(rank==0{
        MPI_Isend(x to process 1)
        MPI_Recv(y from process 1)
}
if(rank==1){
         MPI_Isend(y to process 0);
         MPI_Recv(x from process 0);
}
```

This solves our deadlock (and improves our performance: no extra buffer copy). Irecv() might save us another buffer copy.

Optimized MPI Messaging

Depending on your choice of routines, buffer sizes and implementation, you could achieve either of the more direct transfer scenarios below. Obviously more efficient that the baseline blocking routines, but with more attention to your part.



Communicators

Communicators can often make it much easier to separate logically different actions in your code. In MPI you will usually create and define Groups from which you define your communicators. There are a variety of related routines, but you may only need one or two.

This following example illustrates how a group consisting of all but the 0 process of the World group is created. Then a communicator (comm_slave) is formed for that new group. The new communicator is used in a collective call, and then all processes execute a collective call in the MPI_COMM_WORLD context.

Communicators

```
main(int argc, char **argv)   {

     int my_pe, count, count2;
     void *send_buf, *recv_buf, *send_buf2, *recv_buf2;
     MPI_Group group_world, group_slave;
     MPI_Comm comm_slave;
     static int ranks[] = {0};  /* Ranks to exclude, just 0 here */

     MPI_Init(&argc, &argv);
     MPI_Comm_rank(MPI_COMM_WORLD, &my_pe);

     MPI_Comm_group(MPI_COMM_WORLD, &group_world);
     MPI_Group_excl(group_world, 1, ranks, &group_slave);
     MPI_Comm_create(MPI_COMM_WORLD, group_slave, &comm_slave);

     /* reduce just for slaves */
     if(my_pe != 0){
                         MPI_Reduce(send_buf, recv_buff, count, MPI_INT, MPI_SUM, 1, comm_slave);
     }

     MPI_Reduce(send_buf2, recv_buff2, count2, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

     MPI_Comm_free(&comm_slave);
     MPI_Group_free(&group_world);
     MPI_Group_free(&group_slave);

     MPI_Finalize();

}
```

User Defined Data Types

User defined data types can be both organizationally useful and almost a necessity when dealing with certain data structures. For example, by now you know how to send a specific byte or number from an array. You can also use the number of items parameter to send a consecutive row of such numbers. But, to send a column (actually, the terms “column” and “row” here vary depending on whether we are using C or Fortran due to row or column major ordering – a detail that we deal with in our examples) we would have to resort to skipping around the array manually. Or, we can define a column type that has a stride built in. Likewise, we can define array sub-blocks or more esoteric structures. User defined types are fairly flexible because of a number of MPI Routines to facilitate these definitions.

User Defined Data Types

There are two steps to using a defined data type:

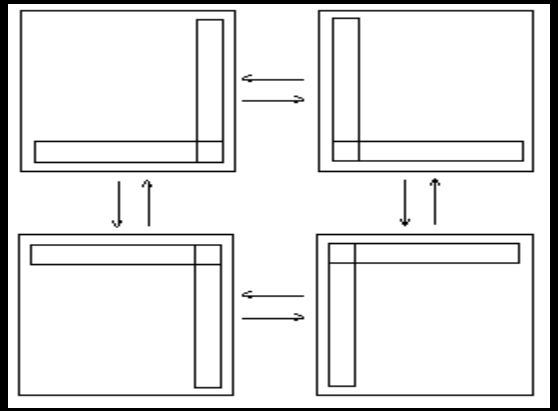
1) Create the type using one of the multiple creation routines

`MPI_Type_contiguous(count…)`	Could be used in our current Laplace.
`MPI_Type_vector(count,…,stride…)`	Adds a stride. Could be used for row/column.
`MPI_Type_indexed()`	Varying strides and block sizes.
`MPI_Type_create_subarray()`	What you would guess and very flexible.
`MPI_Type_create_struct()`	Most general.
`… more`	

2) Commit the type

```
MPI_Type_commit()
```

User Defined Data Types



```
MPI_Datatype row, column ;
MPI_Type_vector ( COLUMNS, 1, 1, MPI_DOUBLE, &row );
MPI_Type_vector ( ROWS, 1, COLUMNS, MPI_DOUBLE , &column );
MPI_Type_commit ( &row );
MPI_Type_commit ( &column );
.
.
//Send top row to up neighbor (what we've been doing)
MPI_Send(Temperature[1,1], 1, row, dest, tag, MPI_COMM_WORLD);
.
//Send last column to right hand neighbor (in a new 2D layout)
MPI_Send(Temperature[1,COLUMNS], 1, column, dest, tag, MPI_COMM_WORLD);
```

Scatter / Gather

These are somewhat related to Bcast and Reduce. Can all theoretically be done with send/recv.

Scatter sends data from a root PE to all of the other PEs. But not the same data; it sends a different portion (determined by count) of a send array to each PE.

Likewise, Gather gets data from all of the other PEs to a single PE, but retains them separately in sections of the receive buffer.

```
MPI_Scatter(void* send_data, int send_count, MPI_Datatype send_datatype, void* recv_data, int recv_count, MPI_Datatype
            recv_datatype, int root, MPI_Comm communicator)

MPI_Gather(void* send_data, int send_count, MPI_Datatype send_datatype, void* recv_data, int recv_count, MPI_Datatype
            recv_datatype, int root, MPI_Comm communicator)
```

There are several significantly different variations on these routines:

```
MPI_Scatterv()          Count can vary for each PE (uses an array for send_counts)
MPI_Gatherv() “”
MPI_Allgather()         Like Gather, but all processes receive result (not just root)
MPI_Allgatherv()        Allgather with varying count
MPI_Alltoall()          No root. Each process sends distinct data to each receiver.  Perfect for transpose or FFT.
MPI_Alltoallv()         Count can vary
MPI_Alltoallw()         Count, displacement and even datatype can vary (crazy flexible)
```

MPI_Wtime: trivial but awesome

This is about as simple as a routine gets, but it brings some much needed portability as a useful high-resolution timer. You will find yourself using it often.

```
double start_time, total_time;
start_time = MPI_Wtime();
.
....  stuff being timed  ...
.
total_time = MPI_Wtime() – start_time;
```

MPI_Wtick() will return the timer resolution. This is usually microseconds or better on HPC platforms.

MPI-2

MPI-2 added whole new areas of functionality well beyond those of MPI-1. Mainly:

- Single sided communication
- Dynamic process management
- I/O

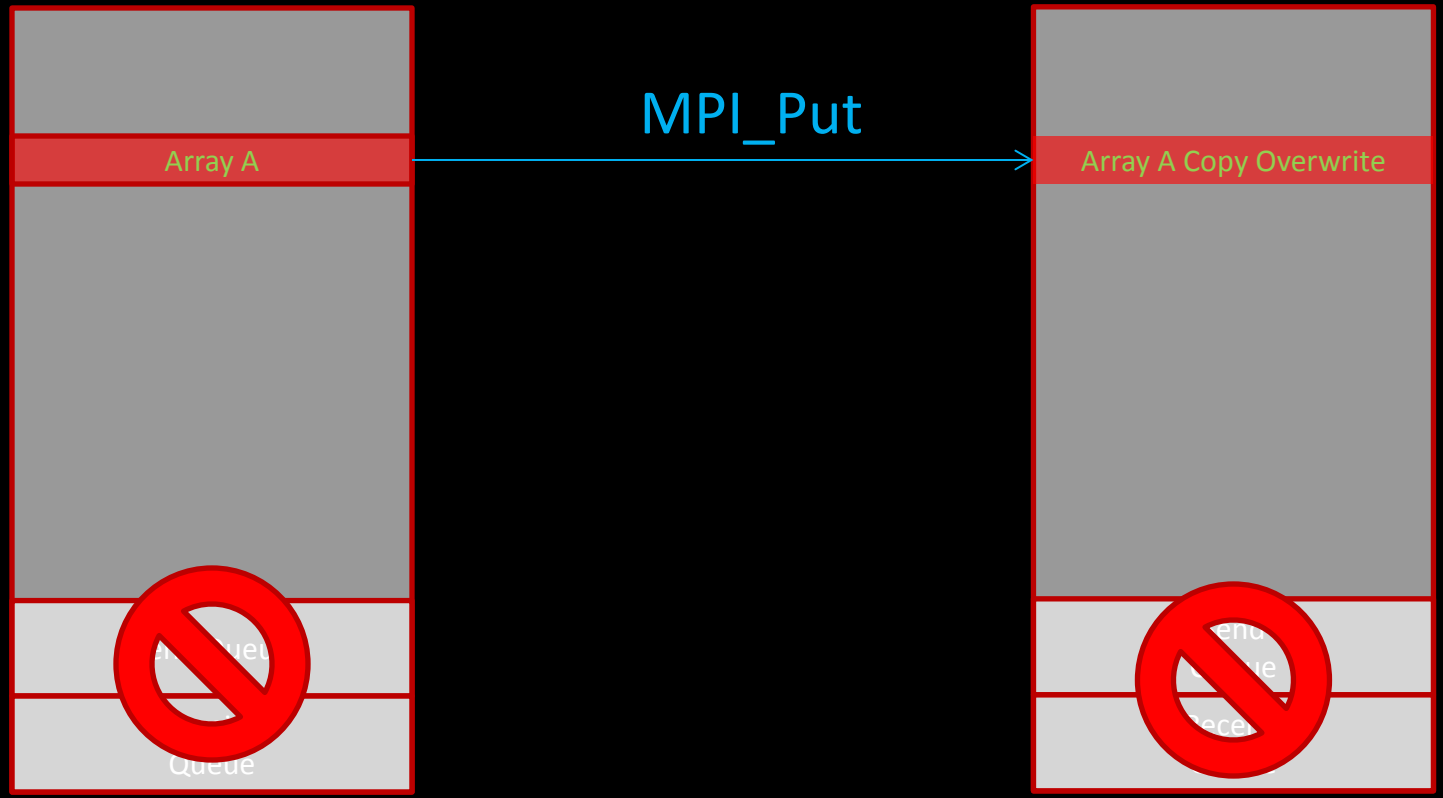
While single-sided communication has become widely implemented, dynamic process management isn't feasible for many environments, and efficient I/O is difficult to abstract away from the specific hardware implementation. As a result, while MPI-1.2 is universally adopted, MPI-2 is irregularly implemented.

One-sided Communication

- These can be very useful and efficient routines and are derived from the “shmem” message passing interface offered on several high performance platforms.
- You can often optimize portions of an MPI-1 code to use these in a fairly painless manner, but much like non-blocking messages, you will have to pay more attention to what state your transfer is in.
- Potential for subtle race conditions. Not as simple as it first appears.

The directness of Put

Now that you understand the normal MPI behavior, the efficiency of Put and Get should be obvious. What are not so obvious are the many ways to generate race conditions if your synchronization is not rigorous. Don't be afraid to use these routines, but definitely study some cases and look at what MPI-3 has added to the mix.



One-sided Communication

Define a ‘window’, which can be accessed by remote tasks, then use:

```
MPI_Put( origin_addr, origin_count, origin_datatype, target_addr,
         target_count,target_datatype, window )
MPI_Get(  ... )
MPI_Accumulate( ..., op, ... )
```

op is as in MPI_Reduce, but no user-defined operations are allowed.

Finding Pi with Single-sided Communication

Adapted from the excellent Argonne MPI Pages (which you should peruse)

```
#include "mpi.h"
#include <math.h>
int main(int argc, char *argv[])
{
   int n, myid, numprocs, i;
   double PI25 = 3.14159265358979323846264338327950288419716939937510;
   double mypi, pi, h, sum, x;
   MPI_WIN nwin, piwin;

   MPI_Init(&argc,&argv);
   MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
   MPI_Comm_rank(MPI_COMM_WORLD,&myid);

   //Windows only on PE 0
   if (myid == 0) {
     MPI_Win_create(&n, sizeof(int), 1, MPI_INFO_NULL, MPI_COMM_WORLD, &nwin);
     MPI_Win_create(&pi,sizeof(double), 1, MPI_INFO_NULL, MPI_COMM_WORLD, &piwin);
   }
   //But Win_create is collective operation so other PEs must still participate
   else {
     MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL, MPI_COMM_WORLD, &nwin);
     MPI_Win_create(MPI_BOTTOM, 0, 1, MPI_INFO_NULL, MPI_COMM_WORLD, &piwin);
   }

   while (1) {
       if (myid == 0) {
           printf("Enter the number of intervals: (0 quits) ");
           scanf("%d",&n);
                   pi = 0.0;
       }

       MPI_Win_fence(0, nwin);
       if (myid != 0)
            MPI_Get(&n, 1, MPI_INT, 0, 0, 1, MPI_INT, nwin);
       MPI_Win_fence(0, nwin);

       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_Win_fence(0, piwin);
       MPI_Accumulate(&mypi, 1, MPI_DOUBLE, 0, 0, 1, MPI_DOUBLE, MPI_SUM, piwin);
       MPI_Win_fence(0, piwin);

       if (myid == 0)
           printf("pi is approximately %.16f, Error is %.16f\n", pi, fabs(pi - PI25));

   }

   MPI_Win_free(&nwin);
   MPI_Win_free(&piwin);
   MPI_Finalize();

}
```

Dynamic Process Management

This paradigm can seem tempting, but as most MPP's require you to run on a specific number of PE's, adding tasks can often cause load balance problems – if it is even permitted. For optimized scientific codes, this is rarely a useful approach.

It can be nice for dynamic work farming algorithms, but these have also often been implemented in MPI 1.x styles with fairly simple queue mechanisms.

Dynamic Process Management

MPI_Comm_spawn creates a new group of tasks and returns an intercommunicator:

```
MPI_Comm_spawn(command, argv, numprocs, info, root, comm, intercomm, errcodes)
```

- Tries to start *numprocs* process running *command*, passing them command-line arguments *argv*.
- The operation is collective over *comm*.
- Spawnees are in remote group of *intercomm*.
- Errors are reported on a per-process basis in *errcodes*.
- *info* used to optionally specify hostname, archname, wdir, path, file.

MPI-IO

This part of MPI-2 has perhaps been the most exasperating. This introduces a parallel API that hopes to abstract away the significant issue of parallel file IO. However, allowing a programmer to chose an abstraction that does not reflect the reality of the filesystem is disastrous for performance. And, providing an interface flexible enough to capture most cases doesn't lend any guidance and becomes an implementation impossibility. Consequently, this part of the standard, even when implemented, is best avoided.

C++

- MPI committee deprecated the C++ binding in MPI 2.2
- It is now removed in MPI 3.0
- Poor overall integration in the C++ environment (especially the STL)
- Alternative: Boost.MPI
  - world.send(1, 1, 4055); (rank,tag,data)
  - Does have substantial performance penalty
  - No streams
  - Does not use MPI_Datatypes but instead serializes data structs with perf penelty
  - No default params
- A shame as a lot of MPI routine parameters can be either inferred by the compiler (type/size of sent/received data) or defaulted (tag = 0, comm = MPI_COMM_WORLD, status = MPI_STATUS_IGNORE)

Other Languages

Bindings are available for at least:

- Perl
- Python
- R
- Ruby
- Java
- .NET

Other Contents of MPI-2

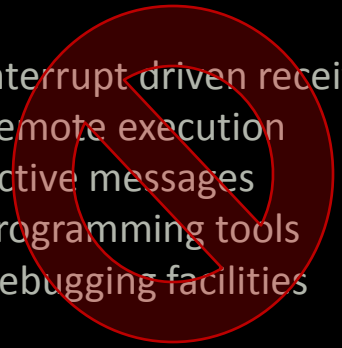
- Extended collective communication operations
- Some F90 support
- Thread support (for hybrid OpenMP/OpenACC)
  - Works as you might hope. Go for it.

MPI-3.0

Just recently approved

- Implementations are rapidly emerging
  - MPICH
  - Open MPI (Looks like most of it is there now)

- Returns to core philosophy of “The standard does not specify operations that require more operating system support than is currently standard.”
  - Interrupt driven receives
  - Remote execution
  - Active messages
  - Programming tools
  - Debugging facilities



MPI-3.0

What does it define? Primarily the features needed to support Exascale computing:

- Collective Communications and Topology
- Fault Tolerance
- Remote Memory Access
- Hybrid Programming

And a few areas that needed improvement:

- Fortran Bindings
- Tools Support

MPI-3.0

Collective Communications and Topology

Even O(n) scalability becomes an issue when n = millions of cores. Both time and buffers start to become costly. So, we start to require the uber-efficient versions of classic MPI functionality:

- Non-blocking collectives
  - MPI_Ibcast
  - MPI_Ireduce
  - MPI_Ialltoall
  - MPI_Ibarrier *a non-blocking barrier!*
  - *Etc.*
- Neighborhood collectives
  - Define your neighbors like a graph (simple Cartesian arrangement provided)
  - Routines participate only in that neighborhood
  - Look like normal routines, but communicator has a "topology" defined
  - MPI_NEIGHBOR_ALLGATHER, etc.

Drafts even mentioned possibility of deprecating the "v" versions of collectives ("MPI_Scatterv"). They did not do that, but it is a strong clue that you will want to avoid those if you are writing exascalable code.

MPI-3.0

Exascale Architecture Anticipation

- Fault tolerance
  - Not completely finished, but not going to be a transparent solution. More like a set of defined behaviors that allow an application to cope with PE failure.
- Remote Memory Access
  - Extends and defines RMA (one-sided communication) requirements in ways that enable more scalable and more flexible performance. Really important for PGAS implementations (UPC and Co-Array Fortran), but also for current RMA apps. Non-blocking RMA, strided gather/scatter, etc.
- Hybrid Programming
  - Improved definition of 2.0 threading mechanism for hybrid computing
    - OpenMP
    - OpenACC
    - Pthreads
    - PGAS languages (UPC, Co-Array Fortran)
    - Intel TBB
    - Cilk
    - CUDA
    - OpenCL
    - Intel Ct.

Hybrid Programming

(Most “complex” version: MPI_THREAD_MULTIPLE)

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

//Last thread of PE 0 sends its number to PE 1

main(int argc, char* argv[]){

   int provided, myPE, thread, last_thread, data=0, tag=0;
   MPI_Status status;

   MPI_Init_thread(&argc, &argv, MPI_THREAD_MULTIPLE, &provided);
   MPI_Comm_rank(MPI_COMM_WORLD, &myPE);

   #pragma omp parallel firstprivate(thread, data, tag, status)
   {
        thread = omp_get_thread_num();
        last_thread = omp_get_num_threads()-1;

        if ( thread==last_thread && myPE==0 )
             MPI_Send(&thread, 1, MPI_INT, 1, tag, MPI_COMM_WORLD);
        else if ( thread==last_thread && myPE==1 )
             MPI_Recv(&data, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &status);

        printf("PE %d, Thread %d, Data %d\n", myPE, thread, data);
   }

   MPI_Finalize();
}
```

```
% export OMP_NUM_THREADS=4
% ibrun -np 3 a.out
PE 0, Thread 0, Data 0
PE 1, Thread 0, Data 0
PE 2, Thread 0, Data 0
PE 2, Thread 3, Data 0
PE 0, Thread 3, Data 0
PE 1, Thread 3, Data 3
PE 0, Thread 2, Data 0
PE 2, Thread 2, Data 0
PE 1, Thread 2, Data 0
PE 0, Thread 1, Data 0
PE 1, Thread 1, Data 0
PE 2, Thread 1, Data 0
```

Output for 4 threads run on 3 PEs

MPI-3.0

Making your programming life easier

- Fortran 90 interface brought up to Fortran 2008 standards
  - Better type checking (how many of you could have used that today?)
  - Array subsections (these would be great for your Laplace code)
  - IERROR optional (looks like we beat the programming pedantics into submission)
  - and more…
- Tools Support
  - Standardize and extend the current hooks that MPI debuggers and profilers use. Only good news for programming environment moving forward.

Which features to use?

As of MPI-3 there are around 450 routines in the library. How do you know where to start?

1. Design your algorithm using MPI-1. In the uncommon event that you find real algorithmic bottlenecks with the capability there, then look deeper into MPI-2.

2. After you have turned your algorithm into working code, you might find that one-sided communication is a useful optimization. You can migrate your code to that level without a complete revision.

3. If you are ready to target 100K+ processors, then MPI-3 has useful capability. And you won't find it intimidating at that point.

Of course this is just a general guide. You may find a more advanced feature central to your needs, or you may wish to forego any unnecessary refactoring and just target a final optimized version. The key concept is that for the vast majority of codes, *once you have your data decomposition, the communication can be incrementally optimized*. And once you have experience with the MPI-1 level routines, you will find the additional routines to be familiar variations on known themes.