Computer Science 400
Parallel Processing
Siena College
Fall 2008

Topic Notes: Message Passing Interface (MPI)

The Message Passing Interface (MPI) was created by a standards committee in the early 1990's.

- motivated by the lack of a good standard
  - everyone had their own library
  - PVM demonstrated that a portable library was feasible
  - portablity and efficiency were conflicting goals
- The MPI-1 standard was released in 1994, and many implementations (free and proprietary) have since become available
- MPI specifies C and Fortran interfaces (125 functions in the standard), more recently C++ as well
- parallelism is explicit - the programmer must identify parallelism and implement a parallel algorithm using MPI constructs
- MPI-2 is an extention to the standard developed later in the 1990's and there are now some implementations

MPI Terminology

- *Rank* – a unique identifier for a process
  - values are 0...$n-1$ when n processes are used
  - specify source and destination of messages
  - used to control conditional execution
- *Group* – a set of processes, associated with a *communicator*
  - processes in a group can take part in a collective communication, for example
  - we often use the predefined communicator specifying the group of all processors in a communication: `MPI_COMM_WORLD`
  - the communicator ensures safe communication within a group - avoid potential conflicts with other messages

- *Application Buffer* - application space containing data to send or received data
- *System Buffer* - system space used to hold pending messages

Major MPI Functions

MPI Simple Program - basic MPI functions

We begin with a simple "Hello, World" program.

See: `/cluster/examples/mpihello`

MPI calls and constructs in the "Hello, World" program:

- `#include <mpi.h>` - the standard MPI header file
- `MPI_Init(int *argc, char *argv[])` - MPI Initialization
- `MPI_COMM_WORLD` - the global communicator. Use for `MPI_Comm` args in most situations
- `MPI_Abort(MPI_Comm comm, int rc)` - MPI Abort function 3
- `MPI_Comm_size(MPI_Comm comm, int *numprocs)` - returns the number of processes in a given communicator in `numprocs`
- `MPI_Comm_rank(MPI_Comm comm, int *pid)` - returns the rank of the current process in the given communicator
- `MPI_Get_processor_name(char *name, int *rc)` - returns the name of the node on which the current process is running
- `MPI_Finalize()` - clean up MPI

The model of parallelism is very different from what we have seen. All of our processes exist for the life of the program. We are not allowed to do anything before `MPI_Init()` or after `MPI_Finalize()`. We need to think in terms of a number of copies of the *same program* all starting up at `MPI_Init()`.

To run, compile with `mpicc`, run with `mpirun` according to the instructions on the course web page.

MPI Point-to-Point message functions

There are just a few that we'll use frequently:

- `MPI_Send/MPI_Recv` - standard blocking calls (may have system buffer)
- `MPI_Isend/MPI_Irecv` - standard nonblocking calls

- wait calls for nonblocking communications: `MPI_Wait, MPI_Waitall, MPI_Waitsome, MPI_Waitany`

And there are many variations that we won't likely use much:

- `MPI_Ssend/MPI_Issend` - synchronous blocking/nonblocking send
- `MPI_Bsend/MPI_Ibsend` - buffered blocking/nonblocking send - programmer allocates message buffer with `MPI_Buffer_attach`
- `MPI_Rsend/MPI_Irsend` - ready mode send - matching receive *must* have been posted previously
- `MPI_Sendrecv` - combine send/recv into one call before blocking
- also: `MPI_Probe` and `MPI_Test` calls

Blocking Point-to-point Communication

A simple MPI program that sends a single message using blocking communication:

See: `/cluster/examples/mpimsg`

- All MPI calls return a status value, and it's a good idea to check it as is done in this example for the `MPI_Init` call.
  - Most class examples will not be thorough in this to keep things looking simpler.
  - For our purposes, any MPI error will cause the program to terminate with an error message, so it usually is not that important to us.
  - When developing large-scale software, we often wish to return error codes rather than crash the whole program, so error checking becomes more important there.
  - The error checking includes a messy little chunk of code to print out appropriate messages, so it's probably worth putting this into your own error reporting function if you want to use it.
- `MPI_Status status` - structure which contains additional info following a receive. We often ignore it, but we will see some instances where it comes in handy.
- `MPI_Send(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm)` - blocking send - does not return until the corresponding receive is completed. sends `count` copies of data of type `type` located in `buf` to the processor with pid `dest`.
- `MPI_Recv(void *buf, int count, MPI_Datatype type, int src, int tag, MPI_Comm comm, MPI_Status status)` - blocking receive - does not return until the message has been received. `src` may be specific PID or `MPI_ANY_SOURCE` which matches, well, a message from any source.

- `MPI_Datatype` examples: `MPI_CHAR, MPI_INT, MPI_LONG, MPI_FLOAT, MPI_DOUBLE, MPI_BYTE, MPI_PACKED`

Non-blocking Point-to-point Communication

A slightly more interesting MPI program that sends one message from each process with non-blocking messages:

See: `/cluster/examples/mpiring`

- `MPI_Request request` - structure which contains info needed by nonblocking calls to check on their status or to wait for their completion.
- `MPI_Isend(void *buf, int count, MPI_Datatype type, int dest, int tag, MPI_Comm comm, MPI_Request *req)` - nonblocking send - returns immediately. `buf` must not be modified until a wait function is called using this request.
- `MPI_Irecv(void *buf, int count, MPI_Datatype type, int source, int tag, MPI_Comm comm, MPI_Request *req)` - nonblocking receive - returns immediately. `buf` must not be used until a wait function is called using this request.
- `MPI_Wait(MPI_Request *req, MPI_Status *status)` - wait for completion of message which had `req` as its request argument. Additional info such as source of a message received as `MPI_ANY_SOURCE` is contained in `status`.

The `MPI_ANY_SOURCE` option is used in this modified version of the example:

See: `/cluster/examples/mpiring_anysource`

Collective Communication

We often need to perform operations at a higher level than simple sends and receives.

See: `/cluster/examples/mpicoll`

- `MPI_Barrier(MPI_Comm comm)` - synchronize procs
- `MPI_Bcast(void *buf,int count,MPI_Datatype type,int root,MPI_Comm comm)` - broadcast - sends `count` copies of data of type `type` located in `buf` on proc `root` to `buf` on all others.
- `MPI_Reduce(void *sendbuf,void *recvbuf,int count, MPI_Datatype type,MPI_Op op,int root,MPI_Comm comm)` - combines data in `sendbuf` on each proc using operation `op` and stores the result in `recvbuf` on proc `root`
- `MPI_Allreduce()` - same as reduce except result is stored in `recvbuf` on all procs
- `MPI_Op` values - `MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND, MPI_LOR, MPI_BOR, MPI_LXOR, MPI_BXOR, MPI_MAXLOC, MPI_MINLOC` plus user-defined

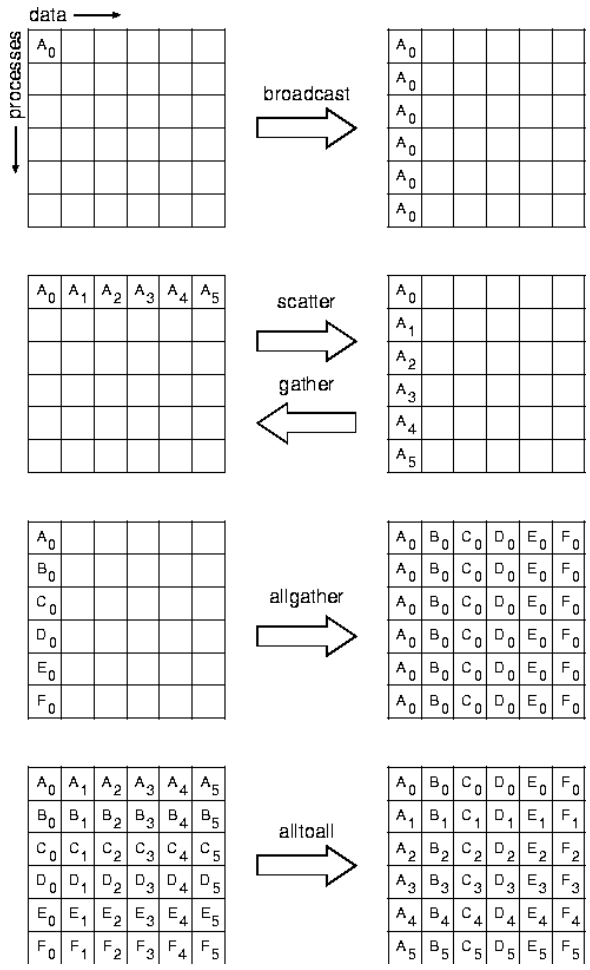
- `MPI_Scan(void *sendbuf, void *recvbuf, int count, MPI_Datatype type, MPI_Op op, MPI_Comm comm)` - parallel prefix scan operations

Scatter/Gather – Higher-level Collective Communication

See: `/cluster/examples/mpiscatgath`

- `MPI_Scatter(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)` - `root` sends `sendcount` items from `sendbuf` to each processor. Each processor receives `recvcount` items into `recvbuf`
- `MPI_Gather(void *sendbuf, int sendcount, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)` - each proc sends `sendcount` items from `sendbuf` to `root`. `root` receives `recvcount` items into `recvbuf` from each proc
- `MPI_Scatterv/MPI_Gatherv` work with variable-sized chunks of data
- `MPI_Allgather/MPI_Alltoall` variations of scatter/gather

To understand what is going on with the various broadcast and scatter/gather functions, consider this figure, taken from the MPI Standard, p.91



Sample MPI Applications

Conway's Game of Life

The Game of Life was invented by John Conway in 1970. The game is played on a field of cells, each of which has eight neighbors (adjacent cells). A cell is either occupied (by an organism) or not. The rules for deriving a generation from the previous one are:

- Death: If an occupied cell has 0, 1, 4, 5, 6, 7, or 8 occupied neighbors, it dies (of either boredom or overcrowding, as the case may be)
- Survival: If an occupied cell has 2 or 3 occupied neighbors, it survives to the next generation
- Birth: If an unoccupied cell has 3 occupied neighbors, it becomes occupied.

The game is very interesting in that complex patterns and cycles arise. Do a google search to find plenty of Java applets you can try out.

I like the one here:

`http://www.math.com/students/wonders/life/life.html`

My implementation is not graphical, so it's a lot less fun. It plays the game, but only computes statistics.

Serial version: **See:** `/cluster/examples/life`

MPI version: **See:** `/cluster/examples/mpilife`

- Since our memory is not shared, we only allocate enough memory on each process to hold the rows that will be computed by that process, plus a "ghost" row on each side that will allow simple computation of our rows.
- When we need to get a global count of some statistic, such as the count of live cells at the start, we use a reduction.
- The communication is done with two pairs of sends and receives. Here, we use nonblocking calls, then wait for their completion with the waitall call.

Matrix-Matrix Multiplication

Matrix-matrix multiplication using message passing is not as straightforward as matrix-matrix multiplication using shared memory and threads. Why?

- Since our memory is not shared, which processes have copies of the matrices?
- Where does the data start out? Where do we want the answer to be in the end?
- How much data do we replicate?
- What are appropriate MPI calls to make all this happen?

The MPI version of Conway's Game of Life used a distributed data structure. Each process maintains its own subset of the computational domain, in this case just a number of rows of the grid. Other processes do not know about the data on a given process. Only that data that is needed to compute the next generation, a one-cell overlap, is exchanged between iterations.

Think about that – no individual process has all of the information about the computation. It only works because all processes are cooperating.

The "slice by slice" method of distributing the grid was chosen only for its simplicity of implementation, both in the determination of what processes are given what rows, and the straightforward communication patterns that can be used to exchange boundary data. We could partition in more complicated patterns, but there would be extra work involved.

The possiblities for the matrix-matrix multiply are numerous. Now the absolute easiest way to do it would be to distribute the matrix `A` by rows, have `B` replicated everywhere, and then have `C` by rows. If we distributed our matrices this way in the first place, everything is simple:

See: `/cluster/examples/matmult_mpi_toosimple`

This program has very little MPI communication – this is by design, as we distributed our matrices so that each process would have exactly what it needs.

Unfortunately, this is not likely to be especially useful. More likely, we will want all three matrices distributed the same way.

To make the situation more realistic, but still straightforward, let's assume that our initial matrices `A` and `B` are distributed by rows, in the same fashion as the Life simulator. Further, the result matrix `C` is also to be distributed by rows.

The process that owns each row will do the computation for that row. What information does each process have locally? What information will it need to request from other processes?

Matrix multiplication is a pretty "dense" operation, and we to send all the columns of `B` to all processes.

See: `/cluster/examples/matmult_mpi_simple`

Note that we only initialize rows of `B` on one process, but since it's all needed on every process, we need to broadcast those rows.

Can we do better? Can we get away without storing all of `B` on each process? We know we need to send it, but we we do all the computation that needs each row before continuing on to the next?

See: `/cluster/examples/matmult_mpi_better`

Yes, all we had to do was rearrange the loops that do the actual computation of the entries of `C`. We can broadcast each row, use it for everything it needs to be used for, then we move on. We save memory!

Even though we do the exact same amount of communication, our memory usage per process goes from $O(n^2)$ to $O(\frac{n^2}{p})$.