ECE 574 – Cluster Computing
Lecture 13

Vince Weaver

http://web.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

21 March 2017

Announcements

- HW#5 Finally Graded
  Had right idea, but often result not an *exact* match.
  Often due to edge conditions.
  Does that matter? Approximate computing?
- HW#6 also Finally graded
  - Comment your code! Even if it's just a few new lines, say what it is supposed to be doing. Parallelism is never trivial.
  - Have to put "parallel" either in separate directive, or

in sections.

- Also time measurement outside parallel area (time in each section is the same with or without threads, the difference is they can happen simultaneously). i.e. be sure to measure wall clock, not user, time
- Don't nest parallel! remove sections stuff for fine.
- Also, does it makes sense to parallelize the most inner loop of 3?
- Also what if you mark variables private that shouldn't be? scope!
- Also if have sum marked private in inner loop, need

to make sure it somehow gets added on the outer (reduction).

- Be careful with bracket placement. Don't need one for a for, for example.
- Also, remember as soon as you do parallel everything in the brackets runs on X threads. So if you parallel, have loops, then a for... those outer loops are each running X times so you're calculating everything X times over. This isn't a race condition because we don't modify the inputs so it doesn't matter how many times we calc each output.

- HW#7 (MPI) will be assigned after midterm
- Short Midterm Thursday
- Project Ideas due 30 March (next Thursday)
- Spent most of break trying to make PAPI faster. Turned up a lot of Linux kernel bugs. One involved pthreads, fun. Trying to get a paper out of it.
- Cluster/Infiniband digression. Managed to get the haswell/broadwell cluster connected by Infiniband over IP (20GB/s!) but can't get OpenMPI to use the connection.

Midterm Review

- Can bring one page (8.5" by 11") of notes. Otherwise closed notes, computers, cell-phones, Beowulf cluster, etc.
- Performance
  - Speedup, Parallel efficiency
  - Strong and Weak scaling
- Definition of Distributed vs Shared Memory
- Know why changing order of loops can make things faster

- Pthread Programming
  - Know about race condition, deadlock
  - Know roughly the layout of a pthreads program. (define pthread_t thread structures, pthread_create, pthread_join)
  - Know why you'd use a mutex.
- OpenMP Programming
  - parallel directive
  - scope
  - section
  - for directive

MPI continued

Some references

`https://computing.llnl.gov/tutorials/mpi/`

`http://moss.csc.ncsu.edu/~mueller/cluster/mpi.guide.pdf`

`https://cvw.cac.cornell.edu/MPIcc/default`

Writing MPI code

- `#include "mpi.h"`
- Over 430 routines
- use `mpicc` to compile

  gcc or other compiler underneath, just sets up includes and libraries for you.
- mpirun -n 4 ./test_mpi
- MPI_Init() called before anything else
- MPI_Finalize() at the end
- Error handling – most errors just abort

Communicators

- You can specify communicator groups, and only send messages to specific groups.
- `MPI_COMM_WORLD` is the default, means all processes.

Rank

- Rank is the process number.
- `MPI_Comm_rank(MPI_Comm comm, int size)`
  `MPI_Comm_rank(MPI_COMM_WORLD, &rank);`
- You can find the number of processes:
  `MPI_Comm_size(MPI_Comm comm, int size)`

Error Handling

- MPI_SUCCESS (0) is good
- By default it aborts if any sort of error
- Can override this

Timing

- MPI_Wtime(); wallclock time in double floating point. For PAPI-like measurements
- MPI_Wtick();

Point to Point Operations

- Buffering – what happens if we do a send but receiving side not ready?
- Blocking – blocking calls returns after it is safe to modify your send buffer. Not necessarily mean it has been sent, may just have been buffered to send. Blocking receive means only returns when all data received
- Non-blocking – return immediately. Not safe to change buffers until you know it is finished. Wait routines for

this.

- Order – messages will not overtake each other. Send #1 and #2 to same receive, #1 will be received first

- Fairness – no guarantee of fairness. Process 1 and 2 both send to same receive on 3. No guarantee which one is received

MPI_Send, MPI_Recv

- block – MPI_Send(buffer,count,type,dest,tag,comm)
- non-block – MPI_Isend(buffer,count,type,dest,tag,comm,rec
- block – MPI_Recv(buffer,count,type,source,tag,comm,statu
- non-block – MPI_Irecv(buffer,count,type,source,tag,comm,r
- buffer – pointer to the data buffer
- count – number of items to send

- type – MPI predefines a bunch. MPI_CHAR, MPI_INT, MPI_LONG, MPI_DOUBLE, etc.
  can also create own complex data types

- destination – rank to send it to

- source – rank to receive from. Also can be MPI_ANY_SOURCE

- Tag – arbitrary integer uniquely identifying message. Can pick yourself. 0-32767 guaranteed, can be higher.

- Communicator – can specify subgroups. Usually use

MPI_COMM_WORLD

- status – status of message, a struct in C
- request – on non-blocking this is a handle to the request that can be queried later to see that status

Fancier blocking send/receives

- Lots, with various type of blocking and buffer attaching and synchronous/asynchronous

Efficient way of getting data to all processes

- master send to each individual, take a while
- some sort of tree, 0 to 1 and 2, 1 sends to 3 and 4, etc.
- use broadcast instead

Collective Communication

- All must participate or there can be problems.
- Do not take tag arguments
- Can only operate on MPI defined data types, not custom
- Operations
  - Synchronization – all processes wait
  - Data Movement – broadcast, scatter-gather

    scatter = take one structure and split among processes

    gather = take data from all processes and combine it
  - Reduction – one process combines results of all others

MPI_Barrier()

- All processes wait at this point.
- `MPI_Barrier (comm)`

MPI_Bcast()

- `MPI_Bcast (&buffer,count,datatype,root,comm)`
- Sends data from the *root* process to each other process.
- Is blocking; when encountering a Bcast all nodes wait until they have received the data.

MPI_Scatter() / MPI_Gather()

- `MPI_Scatter (&sendbuf,sendcnt,sendtype,&recvbuf` `recvcnt,recvtype,root,comm)`
- Copies `sencnt` sized chunks of sendbuf to each processes `recvbuf`
- `MPI_Gather (&sendbuf,sendcnt,sendtype,&recvbuf,` `recvcount,recvtype,root,comm)`
- Have to take care if area sending not a multiple of your number of ranks

MPI_Reduce()

- `MPI_Reduce( void* send_data, void* recv_data, int count, MPI_Datatype datatype, MPI_Op op, in root, MPI_Comm communicator)`
- Operations
  - MPI_MAX,MPI_MIN – max, min
  - MPI_SUM – sum
  - MPI_PROD – product
  - MPI_LAND, MPI_BAND – logical/bitwise and
  - MPI_LOR,MPI_BOR – logical/bitwise OR

- MPI_LXOR,MPI_BXOR – logical/bitwise XOR
- MPI_MAXLOC,MPI_MINLOC – value and location
- Can also create custom

MPI_Allgather()

Gathers, to all.

Equivalent of gathering back to root, then rebroadcasting to all.

MPI_Allreduce()

- Like an MPI_Reduce followed by an MPI_Bcast
- `MPI_Allreduce( void* send_data, void* recv_data int count, MPI_Datatype datatype, MPI_Op op, MP communicator)`
- Once the reduction is done, broadcasts the results to all processes

MPI_Reduce_scatter()

MPI_Alltoall()

Scatter data from all to all

MPI_Scatterv()

Vector scatter. Send non-contiguous chunks. In addition to regular scatter parameters, a list of start offsets and lengths.

MPI_Scan()

Lets you do partial reductions.

Custom Data Types

You can create custom data types that aren't the MPI default, sort of like structures.

Open question: can you just cast your data into integers and uncast on the other side?

Groups vs Communicators

Can create custom groups if you don't want to broadcast to all.

Virtual Topologies

- Map to a geometric shape (grid or graph)
- Doesn't have to match underlying hardware

Examples

See the provided tar file with example code.

Running MPI code

- `mpiexec -n 4 ./test_mpi`
- You'll often see `mpirun` instead. Some implementations have that, but it's not the official standard way.

Send Example

- mpi_send.c
- Run with `mpirun -np 4 ./mpi_send`
- Sends 1 million integers (each with value of 1) to each node
- Each adds up 1/4th then sends only the sum (a single int) back
- Notice this is a lot like pthreads where we have to do a

lot of work manually.

Blocking vs NonBlock Example?

TODO

Wtime Example

- `mpi_wtime.c`
- Same as previous example. but with timing
- Unlike PAPI, the time is returned as a floating point value

Barrier Example

- `mpi_barrier.c`
- Each machine sleeps some time based on rank
- All wait at barrier until last one arrives

Bcast Example

- `mpi_bcast.c`
- Same buffer on each machine
- At the broadcast function, one sends its version of the buffer and the rest wait until they receive the value.
- In the end they all have the same value

Scatter Example

- `mpi_scatter.c`
- Instead of sending all of A, breaks it into chunks and sends it to B in each rank.

Gather Example

- `mpi_gather.c`
- Each rank has its own copy of A which it sets to entirely it's rank number
- Then a gather happens on rank0, of one int each. So what should B have in it? (0, 1, 2, 3, ...)

Reduce Example

- `mpi_reduce.c`
- Instead of waiting in a loop for tasks finishing and then adding up the results one by one, use a reduction instead.
- Many MPI routines are convenience things that could be done by a sequence of separate commands.