ECE 574 – Cluster Computing
Lecture 13

Vince Weaver

`https://web.eece.maine.edu/~vweaver`

`vincent.weaver@maine.edu`

13 March 2025

Announcements

- HW#4, HW#5: still grading
- Midterm: also still grading
- HW#6 was posted: due March 28th

HW#6 Preview

- Getting MPI going can be difficult especially if you don't have a strong C background
- I'm describing here a suggested way to get the coarse code going that is known to work
- We'll also have HW#7 where we do some fine-grained work

HW#6 – Broadcasting the Image Size

- First get rank and num_ranks
- Load the jpeg, but only in Rank 0.
  Could you load it in all? Why or why not?
- Remember this is message passing, so the other ranks do not have any idea what image you are loading or the size, we need to send a message with this info
- First we need to tell the other ranks the sizes: `image.xsize`, `image.ysize`, `image.depth`
- Why? So we can allocate the space for `image.pixels`.

Normally the jpeg load does this for us, but since we only load in rank 0 we have to manually `calloc()` on the other ranks

- How do you send the 3 sizes?
  - Just send 3 integers.
  - Could set up custom struct but not worth it.
  - Most straightforward way is to create an array of 3 ints, and `MPI_Bcast()` it (instead you could send/recv or send one at a time, but probably more code / less efficient)
  - Make sure you use `MPI_INT` as the type

◦ Make sure `image.x`, etc, get set to the values that you sent across

HW#6 – Allocating Space

- Allocate space for the input image on all ranks, something like

```
image.pixels=calloc(image.x*image.y*image.depth,sizeof(char));
```

- Note that `sobel_x.pixels`, etc, will also get allocated but the provided code does that for you.

HW#6 – Broadcasting Image Data

- Use `MPI_Bcast()` to broadcast image data from rank 0 to other ranks.
- Note that Bcast acts as a send from the root source (usually root 0) but as a receive on all other ranks (there's no need to separately have the other ranks receive)

```
result = MPI_Bcast(image.pixels,        /* buffer */
           image.x*image.y*image.depth,/* count  */
           MPI_CHAR,                   /* type   */
           0,                   /* root source  */
           MPI_COMM_WORLD);
```

- Be sure to broadcast to `image.pixels`, not `image`, otherwise you'll overwrite the struct data

- Be sure you are sending `MPI_CHAR` not some other type, as that will also do bad things
- It can be tricky getting this all working. I've added some simple checksum code so you can verify the right data is sent to all ranks before moving on to the next part

HW#6 Guide – Run the Convolution

- You'll need to split up the work like we did with the pthread code
- Splitting it up at the row level is probably best
- You know your rank and total
- If there are 4 ranks, then each rank should calculate ysize/4 rows
- Simplest way is to have ystart as `rank*(ysize/num_ranks` and yend as `(rank+1)*(ysize/num_ranks`
- This looks like it might overlap, but if in loop use $<$ it

should be OK (is it problem if you do overlap?)

- So if 4 ranks then the assignments would be:
  - 0: 0 . . . ((ysize/4))
  - 1: (ysize/4) . . . ((ysize/4)*2)
  - 2: ((ysize/4)*2) . . . ((ysize/4)*3)
  - 3: ((ysize/4)*3) . . . ysize
- Note it's easy to have off by one errors: if your loop is `for(y=ystart;y<=yend;y++)` then you'd actually not a -1 on the end of the limits
- Printing the limits is a good way to help debug when you have errors

- Limits being wrong is a common cause for segfaults/crashing
- Be sure the leftover rows get calculated if not an even division of ranks into rows! Easiest way to do that is check if you're in the last rank and just make the ending row the end of the image
- Be sure you handle skipping y=0 and y=(ysize-1) cases. Do that *after* you calculate the initial split or you can miss lines

HW#6 Guide – Gathering Back the Results

- Once your sobel is done, need to gather back to root
- Note: it's probably easiest to gather the results into a new image (so gather from sobelx.image into sobelx_new.image). Gathering from sobelx.image into sobelx.image itself is tricky
- MPI_Gather();

```
count=(sobel_x.ysize/num_ranks)*sobel_x.xsize*sobel_x.depth;
MPI_Gather(sobel_x.pixels,                   /* source buffer */
           count,                                /* count */
           MPI_CHAR,                             /* type */
           sobel_x_new.pixels,                   /* receive buffer */
           count,                                /* count */
           MPI_CHAR,                             /* type */
```

```
        0,                                   /* root source */
        MPI_COMM_WORLD);
```

- Note: gathers by default will gather from the start of an array, whereas your convolve code probably puts things at an offset (TODO: plot)
- There are three ways to do this once your convolve is done
  - The easiest way is to place the output at the start of the result array (by subtracting y_start from your y_value). (note be sure this is the actual y_start, not one that's been adjusted for the border)
  - Another way is to do a `memcpy()` (or maybe an

`memmove()` to move the results to the start of the array,

- Another way is you can change the source buffer to point to the proper offset in the data array sobel_x.pixels[rank*total_size/num-ranks]

HW#6 Guide – Finishing Up

- After sobel_x, also do sobel_y
- For this homework just do combine step serially in rank#0
- Will do fine-grained combine in HW#7
- Write out result. Remember to only write out on rank#0 (what happens if do this on all ranks?)

HW#6 Guide – Handling Non-Multiple Image Data

- Getting this to work at all is the important first step. However the code above only works properly if you your ysize is an even multiple of the number of ranks.
- Butterfinger is easy, as 320x320 nicely divides by a lot of sizes, but the spacestation example is not as nice
- The most straightforward way is to replace `MPI_Gather()` with `MPI_Gatherv()`
- You will need to set up two additional arrays:

  - One with offsets
  - Ones with lengths
- This is tricky as these are in bytes, so you can re-use your yoffsets from before but you'll need to multiply by xsize and depth. Since we're evenly dividing the offset should be fairly trivial
- For the last rank just have the length have the extra part. This is mildly tricky to calculate, use the % operator to get the remainder and add it on.

HW#6 – Common Failures

- Top of image there, rest is black – usually this means you haven't adjusted your data before gathering, and gather is grabbing from the top of your image which is empty on non-rank0
- Top is fine, but weirdly offset and maybe rainbow for rest – this happens if you gather in (ysize*xsize*3)/ranks chunks rather than (ysize/ranks)*xsize*3. Those look like they are the same, but it's an integer divide so truncating means the latter will grab things in a non-

multiple of the rowsize.

- Looks correct but md5sum doesn't match – this is usually because you forgot to handle the top/bottom border, or else your ystart/yend ranges have small gaps in them

Additional notes on MPI

- Hard to think about. Running on different machine, so setting variables *does not* get set on all, like it does with OpenMP or pthreads
- Tricky: before you can send to rest, they have to know how big of an area to allocate to store it in. How will they know this?
- MPI does not give good error messages. OpenMPI worse than MPICH. Will often get segfault, hang forever, or

weird stuff where it runs 4 single-threaded copies of program rather than one 4-threaded

- Many of the commands are a bit non-intuitive

MPI Debugging (HW#6) notes

- MPI is *not* shared memory
- Picture having 4 nodes, each running a copy of your program *without* MPI.

  Also picture the various MPI routines as a network socket (or web browser query).

  Things initialized the same in all will have same values, no need to initialize.

  Things initialized in only one node will need to be somehow broadcast for the values to be the same in all.

- Problems debugging memory issues.
  Valgrind should work, but Debian compiles MPI with checkpoint support which breaks Valgrind :(
  Mpirun supposed to have -gdb option, doesn't seem to work.
- What does work is `mpiexec -n num xterm -e gdb ./your_app` but this depends on you running X11 plus logging into Haswell-EP with X forwarding (-Y) enabled
- The bug most people hit is improper bounds, leading to segfault. You can debug that with printfs of your bounds
- MPI does give useful error messages sometimes

- Some of the problem is malloc/calloc

Other MPI Notes

- `MPI_Gather( sendarray, 100, MPI_INT, rbuf, 100, MPI_INT, root, comm);`
  rbuf ignored on all but root
- All collective ops are blocking by default, so you don't need an implicit barrier
- MPI_Gather(), same as if each process did an MPI_Send() and the root note did in a loop MPI_Receive() incrementing the offset.

- `MPI_Gather()` aliasing
  cannot gather into same pointer, will get an aliasing error
  Can use `MPI_IN_PLACE` instead of the send buffer on rank0.
  Why is this an error? Partly because you cannot alias in Fortran. Just avoids potential memory copying errors.
  What happens if your gathers overlap?
- Can you handle non-even buffer sizes with MPI_Gather?
  No. Two options.
  - One, just handle in one of other threads (either master

or send/receive from other)

◦ Two, use `MPI_Gatherv()` where you specify the displacement and sizes of what you want to gather

MPI and slurm

- HW `#SBATCH --tasks-per-node=4`
- -N = number of nodes
- -n = number of tasks, default is one task per node?
- N=4 tasks-per-node=4, 16
  N=4 tasks-per-node=4, sbatch -n 8, 16 (N=nodes, n=tasks)
  N=4 tasks-per-node=4, sbatch -N 8, 32

nothing, sbatch -N 8, 32

nothing, sbatch -n 8, (8, 2 nodes * 4 each)

nothing, sbatch -N 8 -n 8 (8, 8 nodes * 1 each)

SLURM update

- Probably not an issue this year, but be careful if your job somehow gets stuck and runs forever. It might keep other people's jobs from running.
- The provided slurm scripts in theory timeout after 10 minutes
- If your job gets stuck, be nice and kill it (using `scancel`)
- In theory I could set a hard limit on the cluster but I don't for now as I'm worried it might break something
- If something does go wrong with the cluster e-mail me

to let me know

Reliability in HPC

Good reference is a class I took a long time ago, CS717 at Cornell:

`http://greg.bronevetsky.com/CS717FA2004/Lectures.html`

Sources of Failure

- Software Failure
  - Buggy Code
  - System misconfiguration
- Hardware Failure
  - Failed capacitors
  - Loose wires
  - Tin whiskers (lead-free solder)
  - Lightning strike
  - Radiation

  - Moving parts wear out
- Malicious Failure
  - Hacker attack
- Environment issues
  - Fire in datacenter
  - Loss of cooling during heat wave

Types of fault

- Permanent Faults – same input will always result in same failure
- Transient Faults – go away, temporary, harder to figure out

What do we do on faults?

- Detect and recover?
- Just fail?
- Can we still get correct results?

Metrics

- MTBF – mean time before failure
- FIT (failure in Time)

  One failure in billion hours. 1000 years MTBF is 114FIT. Zero error rate is 0FIT but infinite MTBF Designers just FIT because additive.
- Nines. Five nines 99.999% uptime (5.25 minutes of downtime a year)

  Four nines, 52 minutes. Six nines 31 seconds.
- Bathtub curve

Architectural Vulnerability factor

- Some bit flips matter less
- (branch predictor) others more (caches) some even more (PC)
- Parts of memory that have dead code, unused values
- Low mantissa bits in floating bit numbers
- Colors in graphics shown for only a frame

Things you can do for reliable Hardware

Hardware Replication / Redundancy

- Lock step – Have multiple machines / threads running same code in lock-step Check to see if results match. If not match, problem. If replicated a lot, vote, and say most correct is right result.
- RAID – (redundant array of inexpensive disks)
- Memory checksums – caches, busses
- Power conditioning, surge protection, backup generators, UPS

- Hot-swappable redundant hardware

Lower Level (Inside your Computer)

- Replicate units (ALU, etc)
- Replicate threads or important data wires
- CRCs and parity checks on all busses, caches, and memories

Lower-Level Problems

Soft errors/Radiation

- Chips so small, that radiation can flip bits. Thermal and Power supply noise too.
- Soft errors – excess charge from radiation. Usually not permanent.
- Sometime called SEU (single event upset)

Radiation

- Neutrons: from cosmic rays, can cause "silicon recoil" Can cause Boron (doped silicon) to fission into Li and alpha.
- Alpha particles: from radioactive decay
- Cosmic rays – higher up you are, more faults Denver 3-5x neutron flux than sea level. Denver more than here. Airplanes. Satellites and space probes are radiation-hardened due to this.
- Smaller devices, more likely can flip bit.

Shielding

- Neutrons: 3 feet concrete reduce flux by 50%
- alpha: sheet of paper can block, but problem comes from radioactivity in chips themselves

Case Studies

- "May and Woods Incident" first widely reported problem. Intel 2107 16k DRAM chips, problem traced to ceramics packaging downstream of Uranium mine.
- "Hera Problem" IBM having problem. ^{210}Po contamination from bottle cleaning equipment.
- "Sun e-cache" Ultra-SPARC-II did not have ECC on cache for performance reasons. High failure rate.