

ECE 574 – Cluster Computing

Lecture 18

Vince Weaver

<https://web.eece.maine.edu/~vweaver>

vincent.weaver@maine.edu

16 October 2026

Announcements

- HW#4 will be graded
- HW#5 will be graded soon as well hopefully
- Don't forget Midterm on Monday
- No class on Wednesday (Engineering Job Fair)

HW#4 (simd) Review

- Baseline Measurement. Not super optimized

Sobel X result:

CYCLES: 818434990 INSTRUCTIONS 2446327758

IPC = 2.99

Sobel Y result:

CYCLES: 818943325 INSTRUCTIONS 2446326513

IPC = 2.99

Combine result:

CYCLES: 433921658 INSTRUCTIONS 640537994

IPC = 1.48

HW#4 (simd) – autovec

- Auto-vectorizes. gcc traditionally not great here
- Looking at objdump you can see AVX instructions (by default limited to SSE2)
- Haswell was one of first machines with AVX2 support
- There is a command line argument that will tell gcc to report where it tried and failed to vectorize

HW#4 (simd) – autovec results

- Speedup: $x=0.8$, $y=0.8$, combine=1
- Note, it got **worse** though IPC slightly better
- Possibly serial code could be written a bit differently to make vectorization easier

Sobel X result:

CYCLES: 1020508450 INSTRUCTIONS 3010858249

IPC = 2.95

Sobel Y result:

CYCLES: 1020537880 INSTRUCTIONS 3010856507

IPC = 2.95

Combine result:

CYCLES: 434301942 INSTRUCTIONS 602859321

IPC = 1.39

HW#4 (simd) – intrinsics

- Hand-done with intrinsics
- Provided code

Sobel X result:

CYCLES: 1382386878 INSTRUCTIONS 2885411820

IPC = 2.09

Sobel Y result:

CYCLES: 1382997639 INSTRUCTIONS 2885411704

IPC = 2.09

Combine results:

CYCLES: 72028246 INSTRUCTIONS 84777784

IPC = 1.18

- Speedup: $x=0.6$, $y=0.6$, $\text{combine}=6x$

HW#4 Summary

- Note: on zen3 results similar but 10x faster on combine
- Mentioned earlier, combine lends a lot easier to simd
- The sobel code while it works is wasting a lot due to trying to do one pixel at a time (but in parallel). Could be much better if we did 16 pixels at a time instead, but doing simd is hard

HW#5 (pthreads) Review – Code

- Usually the difficult part is splitting up the bounds
This year that wasn't too bad
- The Coarse code people mostly did fine on
- The Fine code caused the most trouble

HW#5 (pthreads) Review – Measurements

- When looking at speedup, use the real value from time, not the other ones
- Why was it not perfectly linear? Remember Amdahl's Law?

Speedup	2	4	8	16	32	64
real	1.8	3.2	4.5	5.1	5.0	5.4
convole	1.8	4.5	7.2	13.5	14.9	14.2
combine	2.0	5.5	9.3	18.0	18.3	17.0

HW#5 – Parameter Setup

- The most trouble was trying to get the parameters properly into a struct to pass in
- Each thread **needs** its own struct.
If you have one argument struct, pass it to a thread, then modify it and try to pass to another, there's a race condition where all the threads are using the same struct and any changes you make in the main thread will be seen by all
- You are free to add members to the argument struct. You probably will have to for when you call compare.

HW#5 – Struct Allocation

- Generally the way to do this is `calloc()` or `malloc()` (I know, low-level C)
- Some people were using VLA (Variable Length Arrays) where you can do something like declare

```
int num_threads=4;  
struct thread_arguments[num_threads];
```

- This is a relatively new feature of C so I forget it will work (originally array sizes needed to be a constant). Do note if you use this, the array gets allocated when declared, if you later change `num_threads` it doesn't go back and re-allocate the array size and in that case your code might overrun the buffer.

HW#5 Review – Deadlock Question

- The example code problem is an example where deadlock can occur
- The problem is both threads at the same time can take lock1/lock2 and then when they get to the next locks lock2/lock1 they both get stuck waiting for the other to be free, but since they are both cross-waiting it will never happen, no progress is made, deadlock
- Why not just re-arrange the locks? In real life lock ordering issues aren't always so simple

HW#5 Review – Deadlock Sample

this type of thing might happen in an OS

thread1

```
printf("\b"); // (beeps)
...
console_lock(); // to safely
...           // print
...
beep();
...
sound_lock(); // to get
              // sound hw access
```

thread2

```
play_sound();
....
sound_lock(); // get sound
.....       // hw access
error happens
print_error();
console_lock(); // to safely
               // print
```

Distributed Programming History

- Parallel Virtual Machine (PVM) early implementation
- MPI was competitor
- MPI won somehow? Other proposals never took off?
- There are other ways to have compute clusters, like Hadoop/Spark Map-Reduce, but this is not often used in HPC

Message Passing Interface (MPI)

- Abstraction for sending data between separate processes
- The interesting part is these processes can be on different nodes in a cluster
- You can put together an array of 100 floats, and say “send this to process Y” and like magic it appears there.
- You don't have to worry about setting up networks, network addresses or ports, or decoding streams of bytes
- The code is also portable, so can move to another cluster without a total rewrite

MPI (Message Passing Interface) History

- MPI 1.0 – 1994. MPI 3.0 – 2012
- MPI 1.2 widely used. MPI2.0 is complicated and adoption not as high as it could be.
- MPICH – CH stands for Chameleon – Argonne and Mississippi State
- MVAPICH – from Ohio State, based on MPICH
- OpenMPI – merger of 3 MPI implementations:
 - FT-MPI from the University of Tennessee,
 - LA-MPI from Los Alamos National Laboratory, and
 - LAM/MPI from Indiana University
- Python Bindings, Java bindings, Matlab

MPI

Some references

<https://hpc-tutorials.llnl.gov/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
- `MPI_Init()` called before anything else
can pass in command line arguments that get sent to all, but this isn't always supported
- `MPI_Finalize()` at the end
- Error handling – most errors just abort

Compiling/Running MPI Code

- use `mpicc` to compile
gcc or other compiler underneath, just sets up includes and libraries for you.
- `mpirun -n 4 ./test_mpi`
- `mpiexec` is possibly preferred over `mpirun`
- tools like `slurm` can handle running things for you

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
- You can find out a processes rank

```
MPI_Comm_rank(MPI_Comm comm, int *rank);
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
```
- You can find the number of ranks (processes):

```
MPI_Comm_size(MPI_Comm comm, int *size);
```

This gets all the processes in a communicator, but if you use MPI_COMM_WORLD it gets all of them

How does Rank relate to Threads?

- Each process is given a rank number, 0 .. N
- The processes can be on different machines/nodes, but don't have to be
- Ranks are **not** threads. They are processes. They do not share memory and communicate via distributed messages.
- To confuse things, there's no reason you can't program so that ranks contain multiple threads (using pthreads or OpenMP in addition to MPI)

Other Possibly Useful Commands

- `MPI_Get_processor_name()` – gets name of machine running on
- `MPI_Get_version()` – gets MPI version
- `MPI_Initialized()` – see if MPI has been initialized (useful if you have optional modules)

Error Handling

- `MPI_SUCCESS` means good result
- 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

- These let you transfer data from one rank to another
- Generally one sender and one receiver

Point to Point Concerns

- 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.

Point to Point Operations (more)

- 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 – send data block

- blocking – `MPI_Send(buffer, count, type, dest, tag, comm);`
- non-block – `MPI_Isend(buffer, count, type, dest, tag, comm, request);`
- Parameters
 - 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
 - Tag – arbitrary integer uniquely identifying message. Can pick yourself. 0-32767 guaranteed, can be higher.
 - Communicator – can specify subgroups. Usually use `MPI_COMM_WORLD`
 - request – on non-blocking this is a handle to the request that can be queried later to see that status

MPI_Recv – receive data block

- block – `MPI_Recv(buffer, count, type, source, tag, comm, status);`
- non-block – `MPI_Irecv(buffer, count, type, source, tag, comm, request);`
- Parameters
 - buffer – pointer to the data buffer
 - count – number of items expected
 - type – MPI predefines again
 - source – rank to receive from. Also can be `MPI_ANY_SOURCE`
 - Tag – arbitrary integer uniquely identifying message.
 - Communicator – can specify subgroups. Usually use `MPI_COMM_WORLD`
 - status – status of the receive, a struct in C
has the source, tag, and info on bytes received
 - request – on non-blocking this is a handle to the request that can be queried later to see that status

Handling non-blocking

- `MPI_Wait()` – wait until done
- `MPI_Test()` – test to see if done
- Use the request value in conjunction with them

Fancier blocking send/receives

- Lots, with various type of blocking and buffer attaching and synchronous/asynchronous
- For example `MPI_Ssend()` will wait until it receives confirmation that the remote process got the data

Sample code

```
/* MPI Send Example */
#include <stdio.h>
#include "mpi.h"

#define ARRAYSIZE 1024*1024

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

    int numtasks, rank;
    int result,i;
    int A[ARRAYSIZE];
    MPI_Status Stat;
    int count;

    result = MPI_Init(&argc,&argv);
    if (result != MPI_SUCCESS) {
        printf ("Error starting MPI program!.\n");
        MPI_Abort(MPI_COMM_WORLD, result);
    }

    MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

Sample code part 2

```
printf("Number of tasks= %d My rank= %d\n",
      numtasks,rank);

if (rank==0) {
    /* Initialize Array */
    printf("Initializing array\n");
    for(i=0;i<ARRAYSIZE;i++) {
        A[i]=1;
    }

    for(i=1;i<numtasks;i++) {
        printf("Sending %d ints to %d\n",ARRAYSIZE,i);
        result = MPI_Send(A,          /* buffer */
                          ARRAYSIZE, /* count */
                          MPI_INT,    /* type */
                          i,          /* destination */
                          13,         /* tag */
                          MPI_COMM_WORLD);
    }
}
```

Sample code part 3

```
else {
    result = MPI_Recv(A,                /* buffer */
                     ARRAYSIZE,        /* count */
                     MPI_INT,          /* type */
                     0,                 /* source */
                     13,               /* tag */
                     MPI_COMM_WORLD,
                     &Stat);
    result = MPI_Get_count(&Stat, MPI_INT, &count);
    printf("\tTask %d: Received %d ints from task %d with tag %d \n",
          rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);
}

int sum=0,remote_sum=0;

for(i=rank*(ARRAYSIZE/numtasks);
    i<(rank+1)*(ARRAYSIZE/numtasks);i++) {
    sum+=A[i];
}
```

Sample code part 2

```
if (rank==0) {  
  
    for(i=1;i<numtasks;i++) {  
        result = MPI_Recv(&remote_sum, /* buffer */  
                           1, /* count */  
                           MPI_INT, /* type */  
                           MPI_ANY_SOURCE, /* source */  
                           13, /* tag */  
                           MPI_COMM_WORLD,  
                           &Stat);  
        result = MPI_Get_count(&Stat, MPI_INT, &count);  
        printf("\tTask %d: (%d) Received %d int from task %d with tag %d \n",  
               rank,remote_sum,count, Stat.MPI_SOURCE, Stat.MPI_TAG);  
        sum+=remote_sum;  
    }  
    printf("Total: %d\n",sum);  
}
```

Sample code part 4

```
else {  
    printf("\tRank %d Sending %d\n",rank,sum);  
    result = MPI_Send(&sum,          /* buffer */  
                     1,              /* count */  
                     MPI_INT,        /* type */  
                     0,              /* destination */  
                     13,             /* tag */  
                     MPI_COMM_WORLD);  
}  
MPI_Finalize();  
}
```