ECE 574 – Cluster Computing
Lecture 13

Vince Weaver

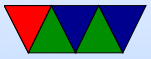
http://www.eece.maine.edu/~vweaver

vincent.weaver@maine.edu

15 October 2015

Announcements

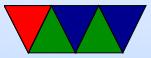
- Homework #3 and #4 Grades out soon
- Homework #5 will be posted soon
  Like HW#4, but OpenMP instead
- Midterm on the 20th



Homework #4 Review

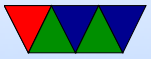
My results based on provided code:

- Single thread provided code:
  1.607s real / 1.52s user
  98ms load
  1264ms convolve
  136ms combine
  81ms store



- Coarse grained two-thread:
  1.022s real / 1.6s user (speedup=1.57 parallel effic=78.5%)
  99ms load
  665ms convolve (speedup=1.9, efficiency=95%)
  136ms Combine
  81ms Store

- Fine grained 8-thread:
  0.662s / 1.86s (speedup=2.5 parallel effic=31.25%)
  99ms load



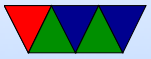
193ms convolve (speedup=6.5, effic=81.8%)

247ms combine (!!?)

81ms store

2-thread code

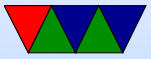
```
/* convolution */
 convolve_data[0].old=&image;
 convolve_data[0].new=&sobel_x;
 convolve_data[0].filter=&sobel_x_filter;

 result = pthread_create(
         &threads[0],
         NULL,
         generic_convolve,
         (void *)&convolve_data[0]);

/* same for y */
```



```
    pthread_join(threads[0],NULL);
        pthread_join(threads[1],NULL);
```

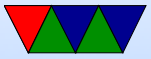
8-thread code

```
/* need an array of these.  why? */
/* shared memory, if change start/end it changes on all copies */
struct convolve_data_t {
        struct image_t *old;
        struct image_t *new;
        int *filter;
        int ystart;
        int yend;
};

Inside convolve:
    for(y=ystart;y<yend;y++) {


/* Then do sobel_x 8-way, join, then sobel_y 8-way, join */
```



```
for(i=0;i<NUM_THREADS;i++) {

                convolve_data[i].old=&image;
                convolve_data[i].new=&sobel_y;
                convolve_data[i].filter=(int *)sobel_y_filter;
                convolve_data[i].ystart=(image.y/NUM_THREADS)*i;
                convolve_data[i].yend=(image.y/NUM_THREADS)*(i+1);

                result = pthread_create(
                        &threads[i],
                        NULL,
                        generic_convolve,
                        (void *)&convolve_data[i]);
        }

        for(i=0;i<NUM_THREADS;i++) {
                pthread_join(threads[i],NULL);
        }
```

Did something similar for combine code, seems to have made something worse, didn't realize until did the PAPI

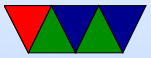


measurements.

How else could you improve things?

Do the combine in parallel too. Once we know a section has X and Y sobel done we can pass off that section to be combined (without waiting for all of X and Y to finish first). Needs much more complicated dependency tracking though.

Other ways to do it: Do RGB in parallel?



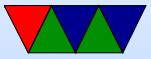
Midterm Prep

You can bring one piece of 8.5x11 paper with notes written on it.

Topics covered:

1. **Parallel Performance**
   Speedup/Parallel efficiency
   Strong vs Weak Scaling

2. **Shared Memory vs Distributed Systems**



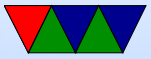
Tradeoffs.
Commodity Clusters

3. **Parallel Hardware**
   Downsides of hardware multithreading,
   Cache behavior (rows vs col major),
   SIMD

4. **Pthreads programming**
   Create. Join.
   Race conditions
   Deadlock

5. **OpenMP programming**
   Some code, what does it do
   for, sections
   static vs dynamic scheduling

6. **Brief MPI question**

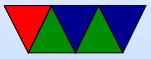


Final Project Preview

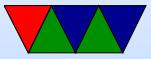
Something parallel related. Work in groups. Small writeup and presentation last week of classes.

Possible Ideas:

- Take code of interest and attempt to parallelize it (negative results are OK too).
- Write some parallel code in C and some other language (Python? Matlab? Java? FORTRAN?) and compare the performance



- Take some existing code and optimize it somehow. Use perf/PAPI, etc.
- Build a small cluster and show it off
- Power/Performance tradeoffs?
- Other tasks we haven't covered yet. GPUs. Big data. Anything HPC/Cluster related

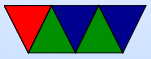


MPI continued

Some references

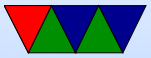
`https://computing.llnl.gov/tutorials/mpi/`

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



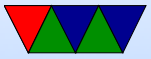
Writing MPI code

- `#include "mpi.h"`
- Over 430 routines
- use `mpicc` to compile
- mpirun -n 4 ./test_mpi
- MPI_Init() called before anything else
- MPI_Finalize() at the end



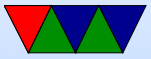
Communicators

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



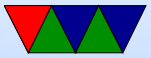
Rank

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



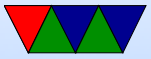
Error Handling

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



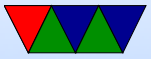
Timing

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



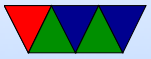
Point to Point Operations

- 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



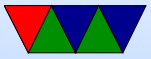
this.

- 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,req
- 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



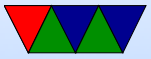
- type – MPI predefines a bunch. MPI_CHAR, MPI_INT, MPI_LONG, MPI_DOUBLE, etc.
  can also create own complex data types

- destination – rank to send it to

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

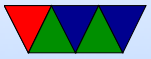
- Tag – arbitrary integer uniquely identifying message. Can pick yourself. 0-32767 guaranteed, can be higher.

- Communicator – can specify subgroups. Usually use



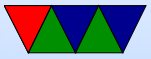
MPI_COMM_WORLD

- 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



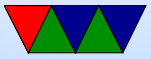
Fancier blocking send/receives

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

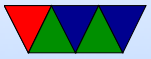


Sample code

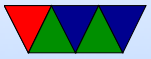
```
/* MPI Send Example */
#include <stdio.h>
#include "mpi.h"

#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);
```



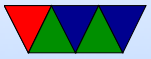
```
MPI_Comm_rank(MPI_COMM_WORLD,&rank);

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);
     }
}
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];
}

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);

    }
    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();
```



```
}
```