ECE 574 – Cluster Computing
Lecture 11

Vince Weaver

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

vincent.weaver@maine.edu

26 February 2019

Announcements

- Homework #4, working on grading
- Midterm will be next Thursday (the 7th). More details as it gets closer
- Project, will post more info on it soon

HW#4 Review

- Debugging – Valgrind for memory bugs!
  `valgrind ./sobel_coarse butterfinger.jpg`
- Helgrind – `valgrind --tool=helgrind`
  can find locking bugs
- Valgrind breaks on recent PAPI sometimes, I've actually contributed a patch to them to fix that
- You might get some false positives due to the way the provided code uses malloc, switch to calloc and it will be happier (on Linux the malloc code is likely getting zeros

by default which is why the result was still looking OK)

HW#4 Review

- Low-level C is a pain. Things like passing pointers to double-indexed arrays, and `(void *)` casting.
  I'd like to say you'll never see this, but if you ever get a job doing Linux kernel or similar low level work there's a lot of this that goes on.
- Hopefully you'll find OpenMP is a lot simpler.
- Some results on a 10848x10824 NASA image I found:

bench	Load	convolve	combine	store
before	945,172	20,972,969	1,740,545	865,404
coarse(2)	952,647	10,752,946	1,785,945	882,353
fine 1	960,527	10,582,954	12,303,506	921,339
fine 2		5,418,575	6,255,203	
fine 8	935,998	1,491,921	3,574,811	928,533
fine 16		729,125	2,097,431	
fine 32		627,906	714,431	

- Should see some speedup, even if not perfect.
  Be sure your joins are *after* both threads started.
- Max speedup? Below, significant time in load/store

combine so even if perfect convolution...

```
Load time: 98257
Convolve time: 871411
Combine time: 266956
Store time: 107583
```

- Question: was an example of deadlock.

Shared Memory vs Distributed Systems

Reminder: shared memory has one copy of OS and all programs see one unified memory space.

Shared Memory

- OpenMP is nice to use. But what if your problem won't fit on a single machine?
- How big can a shared-memory machine be?
- SGI UV systems at least 4096 cores and 16TB running one Linux image

  `http://www.techeye.net/hardware-2/sgi-builds-pittsburgh-4096-processor-core-16tb-shared-memory-su`

- Digression about SGI

- Use special NUMA-Linux architecture to spread cache coherence across multiple machines.
- Origin TM and Onyx2 TM Theory of Operations Manual

Linux limitations

- Linux currently maxes out to 4096 or so.
- Somewhat dated "Scaling Linux to the Extreme" paper problems: cache contention could bring machine to halt (if a global idle counter, each thread trying to increment once a second)
  lock contention, page cache
- What are the challenges? Locking contention?
- Benefits?

(Relatively) easy to code?

Easier to port code

Many libaries do it for you. For example, OpenBLAS.

Eventually you hit the limit

What's the alternative?

Distributed System

- Communicate over a network
- Many systems each with own memory, communicate via Message passing
- Each node has own copy of operating system
- How do they communicate?

Network Topology

- Packet-switching vs bus
- Ring, mesh, star, line, tree, fully connected
- Cube, hypercube
- Mesh networks and routing
- Routing. Fully connected? Crossbar?

Network Types

- Latency vs Bandwidth
- Top500 in Jun 2015:

interconnect	#
infiniband FDR	160
10GB ethernet	83
infiniband QDR	73
gigabit ethernet	63
Cray Gemini	15

- Ethernet – 10/100/1Gb/10GB/40Gb/s

- InfiniBand – low latency, most common in supercomputers
  copper or fiber, GB/s

	SDR	DDR	QDR	FDR-10	FDR	EDR
4x link	8	16	32	**40**	54	96
12x link	23	48	96	120	163	290

- Cray Gemini – Mesh/torus – 64Gb/s
- Fibrechannel
- Older: custom, Myrinet

Programming a distributed System

- Can you implement by hand?
- Sort of how you can use pthread directly?
- Yes, use ssh (like rsh) to run copy of your program on all machines
- Then write custom network code to open sockets and communicate among them all
- Network code is a pain
- Just crying out for abstraction

Message Passing Interface (MPI)

Abstraction for sending chunks of data around network. You can put together an array of 100 floats, and say ”send this to process Y” and like magic it appears there.

MPI

- Message Passing Interface
- Distributed Systems
- 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 Missippi 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
- Any other options? PVM was a predecessor
- Python Bindings, Java bindings, Matlab

MPI

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

  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

Sample code

```c
/* 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();
}
```