hpcbootcamp

Jan 16, 2020

Contents

This programming bootcamp on high performance computing (HPC) will cover the principles and practice of writing parallel programs.

CHAPTER 1

List of topics

- Day 1: Computer architecture, Measuring performance, Optimizing serial code, Parallel hardware
- Day 2: Parallel programming, Writing OpenMP programs, Profiling parallel applications
- Day 3: Writing MPI programs, Parallel performance, Optimizing parallel performance
- Day 4: Task-based programming models, Charm++

1.1 Day 1: Introduction

`Lecture slides`

1.1.1 Using GDB

The GNU Debugger, or GDB, is a freely available tool that you will find almost everywhere. Using a debugger, you will be able to stop execution of your program, step through execution, and observe the state of your program. You can also do post-mortem analysis on core dump files, which are left by the operating system when your program suffers a catastrophic crash.

We will use this `program` for learning gdb:

```
#include <stdlib.h>
#include <stdio.h>
#include <sys/time.h>

#define EXP 6
#define SIZE (1 << EXP)
#define SWAP(a, b) { tmp = a; a = b; b = tmp; }
int comparisons = 0;

/* TUTORIAL: The fill() and init() functions can be ignored. */
/** Fill an array of size n with values between 0 and n. */
```

(continues on next page)

(continued from previous page)

```
void fill(int * array, size_t n) {
    int i;

    for (i = 0; i < n; ++i) {
        array[i] = random() % n;
    }
}

/** Initialize the random number generator using the current time and fill the array. */
void init(int * array, size_t n) {
    struct timeval tv;
    gettimeofday(&tv, NULL);
    srandom(tv.tv_sec);
    fill(array, SIZE);
}

/** Partition the array in to values less than and greater to the pivot, and return the middle index. */
int partition(int * array, size_t n, int pivot) {
    int i, tmp;

    SWAP(array[n - 1], array[pivot]);

    int index = 0;
    for (i = 0; i < n - 1; ++i) {
        ++comparisons;
        if (array[i] < array[n - 1]) {
            SWAP(array[i], array[index]);
            ++index;
        }
    }

    SWAP(array[n - 1], array[index]);

    return index;
}

/** Sort the given array in ascending order. */
void quicksort(int * array, size_t n) {
    /* TUTORIAL: You can see the entire array here by typing "print n" to get the value of n,
       TUTORIAL: followed by "print (int[XXX])*array", where XXX equals the current value of n. */
    int pivot;

    if (n < 2) return;
    pivot = random() % n;
    int middle = partition(array, n, pivot);

    /* TUTORIAL: What happens when you use the step command here? How about the next command? */
    /* NOTE: Incidentally, this step can be parallelized with the SECTIONS and SECTION directives of OpenMP */
    quicksort(array, middle);
    quicksort(array + middle + 1, n - middle - 1);
}
```

(continues on next page)

(continued from previous page)

```
int main(int argc, char ** argv) {
    int i;
    int array[SIZE];

    init(array, SIZE);

    /* TUTORIAL: Consider setting your first breakpoint here to skip past the␣
↪initialization routine. */
    quicksort(array, SIZE);

    printf("%d comparisons to sort array\n", comparisons);
    printf("average case %d comparisons\n", SIZE * EXP);
    printf("worst case %d comparisons\n", SIZE * SIZE);

    return 0;
}
```

To invoke gdb, type:

```
$ gcc -g -o quicksort quicksort.c
$ gdb quicksort
```

You should investigate the following commands in gdb, and understand how and when they should be applied. Commands generally lie in two categories: controlling execution and inspecting the state of the program. First though, is the list command:

- list: List source code. You can list a particular function by using "list function-name". This is useful when using the GDB interface from the command line.

Controlling execution

Breakpoints

A breakpoint is a place in the code where GDB will stop execution. The most simple type of breakpoint unconditionally stops the program every time that point is reached during execution. There are other kinds of breakpoints, such as conditional breakpoints and temporary breakpoints, that we will not cover here.

- break: Set a break point.
- info break: List all break points.
- delete: Delete break points.

Running

- run: Run your program from the beginning. This will run the program until completion, the next breakpoint, or until some error occurs.
- continue: Continue execution until completion, the next breakpoint, or until some error occurs.
- step: Execute the next line of code, including entering functions.
- next: Execute the next line of code, but skip over function calls.

Inspecting state

When a function is called in a program, it pushes down the current program address, function parameters, and local variables for the function on a *function call stack*. Each entry on this stack is called a *stack frame*. The stack serves as a space to allocate temporary variables and let the program know how to return from the function. Among other things, the stack can tell us how the program got to its current state when it crosses a breakpoint or crashes.

Stack

- backtrace: Print the stack trace, from the current function down to the main() function (in C).
- up and down: Go up and down in the function call stack to examine function parameters and local variables.

Memory

- print: Print the current state of a given variable or expression. The print command is smart enough to print arrays and structs properly. You can even make function calls as part of the expression.
- display: Like print, but this produces output every time the program stops. This prints nothing if the expression is not valid.

To see the contents of the array, you will have to cast the pointer as an array so that GDB can display it correctly:

```
display (int[SIZE])*array
```

1.1.2 Using Gprof

Compile the program with -pg:

```
$ gcc -pg -O3 -o pgm pgm.c
```

When you run the program, a gmon.out file will be created. Then, we run gprof to obtain text output:

```
$ gprof pgm gmon.out
```

We will use this program to trying gprof:

```
#include <stdio.h>
#define MAX 500000

/* This program is from "More Programming Pearls", Jon Bentley, 1988 */

int prime(int n);

int prime(int n) {
  int i;

  for (i= 2; i < n; i++)
    if (n % i == 0)
      return 0;

  return 1;
}

int main() {
  int i;

  for (i= 2; i <= MAX; i++)
```

(continues on next page)

(continued from previous page)

```
    if (prime(i))
      printf("%d is prime\n", i);

  return 0;
}
```

You can download all the variations using these links: `primes.zip`

1.1.3 Makefiles

`make` makes it easy to compile large code bases. A Makefile lets us specify what needs to be compiled and what are the dependencies.

An example of a simple Makefile is below:

```
CC     = gcc
CFLAGS = -g

all: quicksort

quicksort: quicksort.c
        $(CC) $(CFLAGS) -o quicksort quicksort.c

clean:
        - rm quicksort *.o *~
```

In the `make directory` for today's lab, there are eight files comprising a hypothetical program. Don't try to figure out what the program does, as it doesn't do anything interesting. Instead, write a Makefile for the whole program that separately compiles each .c file and generates a single executable file. Then add a target clean to the Makefile that removes executable and object files, and then add another target main.tgz that creates a tarfile with that name containing all header and source files. The command:

```
$ tar -czvf <tarfilename> <listoffiles>
```

will create a compressed tar file containing those files.

1.2 Day 2: Writing OpenMP programs

`Lecture slides`

1.2.1 Using slurm

Slurm is used to submit jobs on batch systems. Below is a sample slurm script:

```
#!/bin/bash

#SBATCH -N 1
#SBATCH -t 00:10
#SBATCH --reservation=bootcamp

export OMP_NUM_THREADS=1
```

(continues on next page)

(continued from previous page)

```
cd $HOME/hpcbootcamp

time ./pgm
```

Jobs are submitted to the batch system using sbatch:

```
$ sbatch submit.sh
```

You can check the status of your jobs by typing:

```
$ squeue -u <username>
```

1.2.2 Timing OpenMP programs

We can use `omp_get_wtime()` to time OpenMP programs. Below is a simple example:

```
#include <omp.h>

double elapsed = 0.0;
double tmp;

tmp = omp_get_wtime();
#pragma omp parallel for
for (int i=0; i<n; i++) {
  // do work
}
elapsed = omp_get_wtime() - tmp;

printf("time spent in loop: %f\n", elapsed);
```

1.2.3 GDB and OpenMP

GDB has extensions for debugging multithreaded programs, and we will briefly cover them here.

We will use this file for the `gdb` exercise.

Thread-specific Commands

A thread is a call stack and a program counter. Most programs only have a single thread of execution, but when we start writing parallel programs with OpenMP there can be very many threads. We need a means to display threads, switch between them, and issue debugger commands on them.

- info thread: Print out information about all current threads. If the program is in a parallel loop, then all threads may be active. If the program is in a single-threaded section, all but one of the threads will be in some idle state.
- thread: Print the current thread, or switch to another thread. GDB numbers its threads starting at 1. So, GDB might refer to a thread as "1", while OpenMP refers to the thread as "0", and the operating system itself may have other means for referring to the thread. For example, "thread 1" will switch to thread 1, which is typically the OpenMP master thread.
- `thread apply`: Run a command on one or more threads. For instance, "thread apply all print i" will print the value of variable i on all threads.

How Other Commands Interact With Threads

- `break`: By default, a breakpoint will stop all threads when any single thread meets that breakpoint. For instance, if this is the first iteration of a parallel loop, other threads may not have even reached that point yet. GDB can be configured to let the other threads run.
- break ... thread ...: Set a breakpoint at a given line for a given thread. This can not be used if the given thread does not exist currently. For example, "break gdb-omp.c:18 thread 1" will break execution of the tutorial program at line 18 on thread 1.
- `step` and `next`: The step and next commands increment execution in the current thread. The other threads will run free until execution stops in the current thread at the appropriate line as per step and next semantics. These may not work the way you expect across loop boundaries due to the stack mangling that needs to happen to support multithreaded code.

1.2.4 Adding OpenMP pragmas to a sequential code

The purpose of this exercise is to gain experience in shared memory parallel programming with OpenMP. For this project you are to write a parallel implementation of a program to simulate the Game of Life.

The game of life simulates simple cellular automata. The game is played on a rectangular board containing cells. At the start, some of the cells are occupied, the rest are empty. The game consists of constructing successive generations of the board. The rules for constructing the next generation from the previous one are:

- death: cells with 0,1,4,5,6,7, or 8 neighbors die (0,1 of loneliness and 4-8 of over population)
- survival: cells with 2 or 3 neighbors survive to the next generation.
- birth: an unoccupied cell with 3 neighbors becomes occupied in the next generation.

For this project the game board has finite size. The x-axis starts at 0 and ends at X_limit-1 (supplied on the command line). Likewise, the y-axis start at 0 and ends at Y_limit-1 (supplied on the command line).

A sequential version of the program can be foumnd `here`.

To compile the program, you type:

```
$ gcc -O3 -o life life-seq.c
```

Input

Your program should read in a file containing the coordinates of the initial cells. Sample files are located here: life.data.1, life.data.2, and life.data.1000x1000.

The command line of the program looks like:

```
$ ./life life.data.1 100 250 250
```

The solution file is `here`.

1.3 Day 3: Writing MPI programs

`Lecture slides`

1.3.1 Using MPI

To load a particular MPI, you can use the module commamd. For example,

```
$ module load openmpi/gnu
```

You can then use `mpicc` to compile your MPI program:

```
$ mpicc -O3 -o pgm pgm.c
```

`mpicc` is a wrapper and the command below shows what it does:

```
$ mpicc -show
```

To run an MPI program, you type:

```
$ mpirun -np <#processes> ./pgm
```

1.3.2 Writing a simple MPI program

The exercise is to write a simple ping benchmark in MPI.c. You can start with this boilerplate code:

```
#include <mpi.h>
int main(int argc, char ** argv) {
  int processes, rank;
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &processes);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  ...

  MPI_Finalize();
  return 0;
}
```

1.3.3 Using mpiP

You can load the mpip module by typing:

```
$ module load libunwind
$ module load mpip
```

To link mpiP to your program you need to add these libraries on your link line:

```
-L$(MPIP_LIBDIR) -lmpiP -lm -lbfd -L$(LIBUNWIND_LIBDIR) -Wl,-rpath,$(LIBUNWIND_
↪LIBDIR) -lunwind
```

1.3.4 Using HPCToolkit

1.4 Day 4: Other programming models

Lecture slides

CHAPTER 2

Resources

- Using HPC resources at UMD
- Introduction to parallel computing tutorial
- MPI tutorials
- OpenMP tutorial

CHAPTER 3

Indices and tables

- genindex
- modindex
- search