MODULE TWO: PROFILING

Speaker, Date
MODULE OVERVIEW

Topics to be covered

- Compiling and profiling sequential code
- Explanation of multicore programming
- Compiling and profiling multicore code
COMPILING SEQUENTIAL CODE
PGI COMPILER BASICS

pgcc, pgc++ and pgfortran

- The command to compile C code is ‘pgcc’
- The command to compile C++ code is ‘pgc++’
- The command to compile Fortran code is ‘pgfortran’
- The -fast flag instructs the compiler to optimize the code to the best of its abilities

$ pgcc -fast main.c
$ pgc++ -fast main.cpp
$ pgfortran -fast main.F90
The Minfo flag will instruct the compiler to print feedback about the compiled code.

- Minfo=accel will give us information about what parts of the code were accelerated via OpenACC.
- Minfo=opt will give information about all code optimizations.
- Minfo=all will give all code feedback, whether positive or negative.

```
$ pgcc -fast -Minfo=all main.c
$ pgc++ -fast -Minfo=all main.cpp
$ pgfortran -fast -Minfo=all main.f90
```
GCC COMPILER BASICS

gcc, gc++ and gfortran

- The command to compile C code is ‘gcc’
- The command to compile C++ code is ‘g++’
- The command to compile Fortran code is ‘gfortran’
- The -O2 flag sets the optimization level to 2 (a safe starting point)

```
$ gcc -O2 main.c
$ g++ -O2 main.cpp
$ gfortran -O2 main.F90
```
GCC COMPILER BASICS

Compiler feedback

- The -fopt-info flag will print limited compiler feedback
- The -flto-report flag will also print link-time optimizations, but should be used sparingly due to volume of information

```bash
$ gcc -O2 -fopt-info main.c
$ g++ -O2 -fopt-info main.cpp
$ gfortran -O2 -fopt-info main.f90
```
PROFILING SEQUENTIAL CODE
OPENACC DEVELOPMENT CYCLE

- **Analyze** your code to determine most likely places needing parallelization or optimization.

- **Parallelize** your code by starting with the most time consuming parts and check for correctness.

- **Optimize** your code to improve observed speed-up from parallelization.
PROFILING SEQUENTIAL CODE

Step 1: Run Your Code

Record the time it takes for your sequential program to run.

Note the final results to verify correctness later.

Always run a problem that is representative of your real jobs.

Terminal Window

```
$ pgcc -fast jacobi.c laplace2d.c
$ ./a.out
  0, 0.250000
 100, 0.002397
 200, 0.001204
 300, 0.000804
 400, 0.000603
 500, 0.000483
 600, 0.000403
 700, 0.000345
 800, 0.000302
 900, 0.000269
total: 39.432648 s
```
PROFILING SEQUENTIAL CODE

Step 2: Profile Your Code

Obtain detailed information about how the code ran.

This can include information such as:
- Total runtime
- Runtime of individual routines
- Hardware counters

Identify the portions of code that took the longest to run. We want to focus on these “hotspots” when parallelizing.

Sample Code: Conjugate Gradient

Total Runtime: 22.38 seconds

Matvec 83%
Waxpby 11%
Dot 6%

The “matvec” function is our dominate hotspot
PROFILING SEQUENTIAL CODE
Introduction to PGProf

- Gives visual feedback of how the code ran
- Gives numbers and statistics, such as program runtime
- Also gives runtime information for individual functions/loops within the code
- Includes many extra features for profiling parallel code
Profiling a simple, sequential code

Our sequential program will run on the CPU.

To view information about how our code ran, we should select the “CPU Details” tab.
Within the “CPU Details” tab, we can see the various parts of our code, and how long they took to run.

We can reorganize this info using the three options in the top-right portion of the tab.

We will expand this information, and see more details about our code.
PROFILING SEQUENTIAL CODE

CPU Details

- We can see that there are two places that our code is spending most of its time
- 21.49 seconds in the “calcNext” function
- 19.04 seconds in a memcpy function
- The c_mcopy8 that we see is actually a compiler optimization that is being applied to our “swap” function
PROFILING SEQUENTIAL CODE
PGPROF

- We are also able to select the different elements in the CPU Details by double-clicking to open the associated source code.

- Here we have selected the “calcNext:37” element, which opened up our code to show the exact line (line 37) that is associated with that element.
Step 2: Profile Your Code

Obtain detailed information about how the code ran.

This can include information such as:
- Total runtime
- Runtime of individual routines
- Hardware counters

Identify the portions of code that took the longest to run. We want to focus on these “hotspots” when parallelizing.

Lab Code: Laplace Heat Transfer

Total Runtime: 39.43 seconds

- swap: 19.04s
- calcNext: 21.49s
Step 3: Identify Parallelism

Observe the loops contained within the identified hotspots

Are these loops parallelizable? Can the loop iterations execute independently of each other? Are the loops multi-dimensional, and does that make them very large?

Loops that are good to parallelize tend to have a lot of iterations to map to parallel hardware.

```c
void pairing(int *input, int *output, int N) {
    for (int i = 0; i < N; i++)
        output[i] = input[i*2] + input[i*2+1];
}
```

<table>
<thead>
<tr>
<th>6</th>
<th>3</th>
<th>10</th>
<th>7</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
</table>
| 6 | 3 | 10 | 7 | 2 | 4 | input

<table>
<thead>
<tr>
<th>9</th>
<th>17</th>
<th>6</th>
</tr>
</thead>
</table>
| 9 | 17 | 6 | output
PROFILING MULTICORE CODE

What is multicore?

- Multicore refers to using a CPU with multiple computational cores as our parallel device.
- These cores can run independently of each other, but have shared access to memory.
- Loop iterations can be spread across CPU threads and can utilize SIMD/vector instructions (SSE, AVX, etc.).
- Parallelizing on a multicore CPU is a good starting place, since data management is unnecessary.
PROFILING MULTICORE CODE

Using a multicore CPU with OpenACC

- OpenACC’s generic model involves a combination of a host and a device.
- Host generally means a CPU, and the device is some parallel hardware.
- When running with a multicore CPU as our device, typically this means that our host/device will be the same.
- This also means that their memories will be the same.
PROFILING MULTICORE CODE
Using a multicore CPU with OpenACC

- OpenACC’s generic model involves a combination of a host and a device
- Host generally means a CPU, and the device is some parallel hardware
- When running with a multicore CPU as our device, typically this means that our host/device will be the same
- This also means that their memories will be the same
PROFILING MULTICORE CODE

Comping code for a specific parallel hardware

- The ‘-ta’ flag will allow us to compile our code for a specific, target parallel hardware
- ‘ta’ stands for “Target Accelerator,” an accelerator being another way to refer to a parallel hardware
- Our OpenACC code can be compiled for many different kinds of parallel hardware without having to change the code

```
$ pgcc -fast -Minfo=accel -ta=multicore laplace2d.c
```
```
calcNext:
    35, Generating Multicore code
    36, #pragma acc loop gang
```
The first difference we see in this multicore profile is that there is now a “timeline”

This timeline will show when our parallel hardware is being used, and how it is being used

Each of the blue bars represent a portion of our program that was run on the multicore CPU
PROFILING MULTICORE CODE

CPU Details

- Looking at our CPU Details, we can see that there is a lot more happening compared to our sequential program.

- For the most part, these extra details revolve around the need for the CPU cores to communicate with each other.
PROFILING MULTICORE CODE

PGPROF

- Just like earlier, we see our “calcNext” function
- We also see that PGPROF is reporting this function to take 61.72 seconds to run
- Looking at the program now, it looks like it performs much worse than the sequential version
PROFILING MULTICORE CODE

PGPROF
The program is actually performing better than the sequential version.

We are only looking at the “TOTAL” view, which means that PGPROF is combining information from all of our CPU cores.
PROFILING MULTICORE CODE

Observing a single thread

- Now we have selected to view a specific thread (for us, a thread would be a single CPU core)

- We can see that this single thread only spent 9.8 seconds running calcNext

- Each thread will take a similar amount of time and execute simultaneously, resulting in a faster run
LAB CODE
We will observe a simple simulation of heat distributing across a metal plate.

We will apply a consistent heat to the top of the plate.

Then, we will simulate the heat distributing across the plate.
The lab simulates a very basic 2-dimensional heat transfer problem. We have two 2-dimensional arrays, $A$ and $A_{\text{new}}$.

The arrays represent a 2-dimensional, metal plate. Each element in the array is a double value that represents temperature.

We will simulate the distribution of heat until a minimum change value is achieved, or until we exceed a maximum number of iterations.
We will take the average of the neighboring cells, and record it in *Anew*.

The **calcNext** function will iterate through all of the inner elements of array *A*, and update the corresponding elements in *Anew*.

We will take the average of the neighboring cells, and record it in **Anew**.

The **swap** function will copy the contents of *Anew* to *A*.
The swap function will copy the contents of Anew to A.
KEY CONCEPTS

In this module we discussed…

- Compiling sequential and parallel code
- CPU profiling for sequential and parallel execution
- Specifics of our Laplace Heat Transfer lab code
LAB GOALS

In this lab you will do the following…

- Build and run the example code using the PGI compiler
- Use PGProf to understand where the program spends its time
THANK YOU