PGI compiler and OpenACC parallelism on the SCC

January 30, 2019

Some problems are too big to be solved by a single core. As a first step into parallel programming, we're going to look at OpenACC. OpenACC itself will have two aspects of interest to us. The first is its ability to parallelise over many compute cores. If, say, one had a CPU node with 16 cores, OpenACC can utilise all 16 of those cores in parallel This is called 'multicore parallelism' and is very similar to OpenMP. On the other hand, if one had a GPU, OpenMP is unable to exploit that hardware, but OpenACC can use it with minimal (but vital) change to the CPU code. We will learn how to write OpenACC code for multicore architecture, and then learn how to modify the code so it will run on GPU.

I Working on the Shared Computing Cluster [SCC]

This entire section will serve as a tutorial for getting on to and working on the SCC. Read it over once and refer back to it when you need to.

I.1 Accessing the SCC

To access SCC, open a terminal and run:

```
ssh [username]@scc1.bu.edu
```

Your username is your BU account name. Once you're on SCC, make sure you're in a bash shell. If you're in a bash shell by default, your command prompt will include your username:

```
[username]@scc1 ~]
```

If that's roughly what it looks like, you're all set. If your default shell is csh, it'll look like this:

```
scc1:~ %
```

If your command line looks like this, just run the command bash to get to the appropriate shell. Once you're there, let's navigate to our class project's active directory:

```
cd /projectnb/paralg/
```

Create your own directory with your kerberos username and enter it:

```
mkdir [username]
cd [username]
```

This is your personal space to store source code and do development.

I.2 Navigating the SCC

You need to be aware of what is known as your 'bash environment'. That is, all the Linux command that your bash shell can see, as well as all the code libraries it is able to utilise. The Linux commands are things you should spend some of your own time researching. There are more than enough examples here `https://courses.cs.washington.edu/courses/cse390a/14au/bash.html` for reference. For example, if you are lost and don't know which directory you are in, type `pwd`. You should see:

```
/projectnb/paralg/[username]
```

The command `pwd` is 'print working directory'. As a general rule, if there is something you want to do with files or data on a computer, there is a command for it. You just need to find it!

There are also many libraries and compilers on the `SCC`. Libraries are collections of code that perform specific tasks (such as matrix inversion or data writing) that have been compiled on the `SCC` for you. You need to know how to link to these libraries and compilers if you want to use them. For the moment, all we need is the `PGI` compiler so that our `OpenACC` code will compile.

I.3 Modules on the SCC

To see what's available on the SCC in your current bash environment, use the `module` command with the list qualifier

```
module list
```

Here's an example of the output from `module list` on a researcher's account

```
[howarth@scc2] module list
Currently Loaded Modulefiles:
1) gcc/4.9.2 2) gsl/2.3 3) openmpi/2.1.0_gcc4.8.1 4) openblas/0.2.20
5) gmp/6.1.2 6) fftw/3.3.6 7) cuda/9.2 8) libcurl/7.48.0
9) emacs/25.3 10) lapack/3.8.0 11) mpfr/4.0.1 12) autoconf/2.69
13) eigen/3.3.4 14) git/2.6.3 15) cmake/3.12.3 16) pgi/18.4
17) hdf5/1.8.18_gcc-6.2.0 18) automake/1.15
```

Yours will look very different. What you will need is the `PGI` compiler. Type

```
module avail pgi
```

to see what is available. You should see three versions. We will be using the latest, 18.4, so type

```
module load pgi/18.4
```

To make sure it is loaded, re run the command that lists all your currently loaded module files. It should be the same as before, with the addition of the `PGI` compiler. There's a way to make your bash environment load the `PGI` compiler every time you log in. If you can't work it out, ask in class.

I.4 Compiling on the SCC

The `SCC` has two different parts: compute nodes and login nodes. One generally writes, compiles and performs short tests on the login nodes. One runs large (many cores, GPUs) or long (longer than 1 minute) on the compute nodes. Let's ensure that your environment is good by compiling and running a simple program on the login node.

In order to edit code on a remote machine, you need an editor. Your choice here is entirely up to you, but the most popular editors are `emacs` and `vim`. These editors are almost always available on computing clusters. Indeed, if you execute the queries

```
module avail vim
module avail emacs
```

you'll see that these editors are indeed available. Create a file named `hello.cpp` and fill it with the following code:

```
#include <iostream>

int main(int argc, char** argv){
  std::cout << "Hello world!" << std::endl;
  return 0;
}
```

To compile it, invoke the `g++` compiler via the following command

```
g++ hello.cpp -o hello
```

The `g++` compiler was instructed to compile the source code `hello.cpp` and direct the output (the executable) to a file named `hello`. To run it, execute

```
./hello
```

Did the program do what you expected?

I.5 Compiling `OpenACC` on SCC

```
qrsh -l h_rt=1:00:00 -P isingmag
```

Next, let's use the `PGI` compiler to make a parallel version of a serial program. This file is called `integral.cpp` and, for the moment, it's just serial `g++` so will compile with

```
g++ integrate.cpp -o integrate
```

```cpp
#include <stdio.h>
#include <iostream>
#include <math.h>

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

  int N=1000000000;
  double lower = -100.0;
  double upper = 100.0;
  double interval = upper - lower;
  double increment = interval/N;
  double sum = 0.0;

  double time = -clock();

  //Integrate from -pi to pi
  for(int dx=0; dx<N; dx++) {

    //The x-axis value
    double x = lower + increment*(dx + 0.5);
    sum += increment * exp(-x*x);

  }

  time += clock();

  printf("Integral of exp(-x*x) from x=%.2e to %.2e = %.16f, error = %e.%%\n",
   lower, upper, sum, 100*(sum - sqrt(M_PI))/sqrt(M_PI));

  return 0;
}
```

This is just a simple Reimann integral of the $\exp(-x^2)$ function. If you run it, it should take about two seconds and evaluates to around $\sqrt{\pi}$. The idea behind the function is very simple: split the function into small slices and add them up. In the code above, this is done in serial. However, addition is commutative $a+b=b+a$ so the order in which we perform this sum is immaterial to the final answer. We will use this commutativity to parallelise the sum, but first, we must compile with `PGI`. Here's the command:

```
pgc++ -Minfo=accel -ta=multicore integrate.cpp -o integrate
```

If that command works, you're good to add some accelerator code. The code we must add is very straightforward and is known as pragma. Take a look at the additions:

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

  // OpenACC initialise
  #pragma acc init
  int N=1000000000;

...

  //Integrate from -pi to pi
#pragma acc parallel loop
  for(int dx=0; dx<N; dx++) {
```

When you recompile it, you will get some output like this:

```
main:
17, Generating Multicore code
21, #pragma acc loop gang
17, Generating implicit reduction(+:sum) >
```

The compile flag `-Minfo=accel` gives you description of the acceleration code that `PGI` generated for you. `-ta=multicore` instructs `PGI` that the 'target accelerator' is multicore. The main word to pay attention to here is 'reduction'. The compiler sees each iteration of the loop as independent, and assigns a value of `sum=0.0` to each iteration of the loop. After all the iterations have been performed, `PGI` graciously sums up all the individual values of `sum` and returns the result.

Next, let's use a GPU. Ordinarily, porting code from CPU to GPU requires a lot of thought and energy. The format of the loops must be changed, and special memory allocation functions must be called to transfer data from the CPU to the GPU. But not so with `OpenACC`. The compile flag this time is:

```
pgc++ -Minfo=accel -ta=tesla integrate.cpp -o integrate
```

and your output should look like:

```
main:
17, Accelerator kernel generated
Generating Tesla code
17, Generating reduction(+:sum)
21, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
17, Generating implicit copy(sum)
```

However, if you run this code, you'll get an error. The host node on the SCC does not have a GPU, so next we learn how to access one.

I.6 Running Jobs on SCC

To get hold of a GPU on the SCC, run the following command:

```
qrsh -l h_rt=00:30:00 -pe omp 1 -P paralg -l gpus=1.0 -l gpu_c=6.0
```

This gives you a GPU with compute capability 6.0, and one CPU core, for 30 minutes. It will be charged to the paralg group. Run the GPU code you just compiled, see if it works. If not, take a look here for some clues `http://www.bu.edu/tech/support/research/system-usage/running-jobs/`

II PGI Accelerated Poisson Relaxation

We will now put the GPU and `PGI` compiler to good use. We turn to the problem described in lecture 3 (Jan 29), namely the 2D Poisson relaxation problem. The code for this example can be found in

```
/projectnb/paralg/OpenACC/Poisson2D
```

wherein you will find a makefile and several source code files. Copy these files to a place inside your directory. To make the executables, ensure you have the `PGI` compiler in your module list, and ensure that in the Makefile you have the tesla acceleration type uncommented, as shown:

```
#ACCEL_TYPE=PGI-multicore
ACCEL_TYPE=PGI-tesla
#ACCEL_TYPE=PGI
```

Then type

```
make v3
```

to make the `poisson2d_v3` executable. Get a GPU node, run the code, observe the speed-up, and then study comments in the source code of `poisson2d_v3.c` to understand how it was done. Here's a picture to illustrate the code.