Hands On Session I: Parallel Programming Concepts

John Burkardt
Information Technology Department
Virginia Tech

.....

HPPC-2008
High Performance Parallel Computing Bootcamp

28 July 2008

This is the first hands on lab session for the HPPC bootcamp. We have not presented the details of how to use the two main parallel programming models yet, so today's session will primarily be laying the groundwork for what is coming.

Therefore we will

1. make sure you can access various computers and run a program on them;
2. force you to use an editor to write a program that produces a timing statistics
3. run a benchmark program that reports the MegaFLOPS rate on the computers we'll be using
4. run a program that reports MegaFLOPS rates for a variety of problem sizes, and try to profile the program as well
5. let the compiler try to **autoparallelize** some codes

You are encouraged to try these experiments on your laptop or desktop machine as well. You may be surprised to find that, in some cases, the processors on the supercomputers are no faster than what you have on your desk.

For most of the exercises, there is a source code program that you can use. The source code is generally available in a C, C++, FORTRAN77 and FORTRAN90 version, so you can stick with your favorite language. However, in some cases you may find it helpful to compare how things work in other languages.

The source codes are available in three ways:

- on the web page **http://www.scs.fsu.edu/~burkardt/vt2/hppc_2008/hppc_2008.html**
- on both System X and the SGI, in the directory **/home/BOOTCAMP/day1**;
- on a memory stick I can loan you

Normally, System X and the SGI are used indirectly, in batch mode. Users may log in to a special node to compile jobs, but all runs are made by submitting a shell script to a queueing system.

For one day only, we will be allowed to log in to a node of System X or the SGI, copy files there, compile them, and run them. We won't be running big jobs, but this will give you a chance to get a quick feeling for what the environments are like.

To log in to System X or the SGI, you have been assigned a username and password, which you can get from any of the staff members in the lab today. You will need this information for the later handson sessions as well.

The address of the SGI node we may use is **charon3.arc.vt.edu**.
The address of the System X node we may use is **sysx5.arc.vt.edu**.
To log in directly, use the **ssh** command, as in:

```
ssh charon3.arc.vt.edu
```

To transfer files, use the **sftp** command, as in:

```
sftp sysx5.arc.vt.edu
get /home/BOOTCAMP/day1/hello/hello.cc
quit
```

1 Hello, World!

Before you really get started programming, it's useful to take a simple program and run it through its paces. We will take the **Hello, world!** program as our toy program. Your assignment is to go through the motions of running this program.

Get a copy of the "hello" program.

1.1 Run "hello" locally

Let's assume this file is now sitting on your laptop or desktop computer.

You can start by trying to compile the program on your local system, if you have a commandline environment, and have compilers installed. You may find that compiling and correcting code on your local system is a good first step before using remote systems.

1.2 Run "hello" on System X or the SGI

Use the **ssh** program to log into System X or the SGI. Compile the **hello** program, run it, save the output to a file, and bring that output back to your home system, to prove you did it.

Here are the details of how to do that!

The **hello** program is already available on either machine, in a subdirectory of the **/home/BOOTCAMP** directory. If you want the FORTRAN90 version of the file, for instance, type

```
cp /home/BOOTCAMP/day1/hello/hello.f90 .
```

System X provides the Gnu compilers: gcc, g++, and gfortran; the SGI has the Gnu compilers too, but also has the Intel compilers: icc, icpc, and ifort.

A simple compile command might be

```
gcc hello.c
```

Compilation creates an executable program, called **a.out** by default. You run the executable program by typing

```
./a.out
```

Program output can be saved to a file using the > symbol:

```
./a.out > hello_output.txt
```

Congratulations! Now we can try the harder stuff.

2 Who Wants to Sum a Million?

You need to have access to an editor, you need to be able to write programs in some version of C or FORTRAN, and you need to be able to fix mistakes as the compiler reports them. This exercise asks you to go through these steps.

We are going to write a program that adds the numbers from 1 to 1,000,000.

While this might seem a good application of integer arithmetic, it's unlikely we would get the correct answer if we use integer arithmetic. Do you have an opinion on why?

If you want help getting started, there are partially written programs available. For instance, a C version can be gotten by:

```
cp /home/BOOTCAMP/day1/sum_million/sum_million.c .
```

Your first version of the program should be written so that:

- loop number 1 assigns the value of each entry of an array of size 1,000,000
- loop number 2 sums up the entries of the array.
- the sum and error are printed out

The correct answer is 500,000,500,000. Compare your result to this value. If there is a discrepancy, can you explain it? Can you fix it?

2.1 Compute the MegaFLOPS rate

In your second version of the program, we are going to take a simple stab at estimating the computational rate of the computer. We will do this by counting the number of FLOPs performed by the summation loop, and dividing by the time it takes.

We will ignore the loop overhead, and assume that exactly 1,000,000 floating point operations are carried out. Now we need a way to compute the time.

In FORTRAN90, and perhaps in FORTRAN77, you can get the elapsed cpu time in seconds by the following method:

```
call cpu_time ( ctime1 )
 code to be timed
call cpu_time ( ctime2 )
ctime = ctime2 - ctime1
```

In C and C++, you can get the elapsed cpu time in seconds by the following method:

```
ctime1 = ( double ) clock ( ) / ( double ) CLOCKS_PER_SEC;
 code to be timed
ctime2 = ( double ) clock ( ) / ( double ) CLOCKS_PER_SEC;
ctime = ctime2 - ctime1;
```

(The sample C and C++ programs include this as a function called **cpu_time**).

Use the appropriate method to time your summation loop, and then compute the MegaFLOPS rate at which your loop was carried out.

2.2 "Fatten" the loop

One reason that computers can be inefficient is when they spend as much time in bookkeeping as in computation.

It's possible that, in your loop, the computer is as busy updating the loop index I as it is in carrying out the addition. To investigate this possibility, try to **unroll** the loop. That is, we will do the loop in steps of 4 (luckily 1,000,000 is evenly divisible by 4 so there's no messy leftover numbers to sum.)

If your summation loop looks like this:

```
total = 0.0;
for ( i = 0; i < n; i++ )

  total = total + x[i];
```

make a new version that looks like this:

```
total = 0.0;
for ( i = 0; i < n; i = i + 4 )

  total = total + x[i] + x[i+1] + x[i+2] + x[i+3];
```

Both loops do the same thing, but one loop spends less time updating the loop index and checking the loop condition. Do you notice any difference in the timings?

3 The LINPACK Benchmark

This exercise asks you to run the LINPACK benchmark program. The program solves a 1000x1000 linear system **A*x=b**, estimates the floating point operations, measures the CPU time, and comes up with a MegaFLOPS rating for the computation

The work done is more typical of a scientific calculation than the simple summation loop of the previous exercise. There's more work done in each iteration of the loop, for instance. We expect to get a more reliable rating from this program.

Get a copy of the program. For instance

```
cp /home/BOOTCAMP/day1/linpack_bench/linpack_bench.cc .
```

Compile the program, run it, and note the MegaFLOPS rating.

Note that, for C and C++ files, you may need to include the math library in your compile statement. For example:

```
gcc linpack_bench.c -lm
```

3.1 LINPACK MegaFLOPS as a function of N

The problem size **N**=1000 could be in the "sweet spot" for the computer you are studying; that is, the MegaFLOPS rate is probably close to the maximum you will encounter.

To see how the MegaFLOPS rate depends on **N**, try running the program with a series of different values. I could suggest **N** = 25, 50, 100, 200, 400, 1000. Can you estimate the value of **N** for which the MegaFLOPS rate is about half of what is achieved at **N**=1000.

3.2 Matrix Storage and Memory Delays

Most versions of the program include a variable called **LDA**, which influences how the matrix **A** is stored. If **LDA** is equal to **N**, then the physical storage is exactly big enough to hold the matrix. However, another common choice for **LDA** is to set it equal to **N+1**. In this case, the storage of the matrix is slightly "staggered".

Many computer memories are "striped", or arranged so that an array of data is spread over several banks. Access to the array is quick when successively requested values lie in different banks. Depending on whether we are using FORTRAN or C, and whether we are accessing successive entries of a row or column, we are running through the data by increments of 1 or increments of **LDA**.

See if you can spot this memory bank problem. Run the program using values of **N** and **N**+1 for **LDA**. Report the resulting MegaFLOPS rates. Are they close?

This is an example, for a non-parallel program, in which communication issues affect computational speed. When we move to parallel execution, communication issues become even more serious!

4 The FFT program

This exercise will introduce you to an FFT (Fast Fourier Transform) program. Later in the week we will turn this program into a parallel version. For today, we just want to run and observe it.

The program sets up a complex data vector X of size N, computes the Fourier transform of the data, and the inverts the transform to recover X. Fourier transforms are commonly used on data to search for patterns of periodic behavior. The "Fast" Fourier Transform produces the result using a number of operations that is on the order of $N * \log(N)$. The original "slow" algorithm uses N^2 operations.

Since the program has an estimate for the number of floating point operations as a function of **N**, it can produce a MegaFLOPS rating for each value of **N** that it investigates.

4.1 FFT MegaFLOPS as a function of N

Get a copy of the program, compile and run it on the machine you are investigating, and save the output in a file. Look at the trend in the MegaFLOPS ratings. Depending on the machine you are using, you may find that you hit the "performance plateau" and are even beginning the eventual deterioration of performance for large **N**.

4.2 Where is the Work?

When we come back to this program, we are going to need to identify the part of the program where the work is concentrated. We do not want to waste time improving code that does not represent the big work.

This FFT program looks quite mysterious. There are just a few routines, but it's enough to be confusing. Use the **gprof** utility to try to determine where the execution time is being spent.

Here is a brief reminder of how to use **gprof** on, for example, the **C** version of the program:

```
gcc -pg fft.c -lm   fft needs the "math" library
./a.out             run the program
gprof a.out         have gprof make the report
gprof a.out | more  to see the report a page at a time
```

You should be able to see that the hard work of this program goes on in a single function. In order to get good parallel performance, we can ignore the rest of the program, and concentrate on this function!

5 The MD program

This exercise will introduce you to a program called MD (Molecular Dynamics). The purpose of this program is to simulate the behavior of a collection of N particles in a box, by computing their positions, velocities and accelerations at **STEP_NUM** regularly spaced times. Later in the week we will turn this program into a parallel version.

The program does some initialization, and then enters an integration loop. The results of each time step are summarized by printing the time, and the kinetic and potential energies (whose sum should be constant).

Begin by getting a copy of the program and transfer it to the computer you are interested in. Compile the program, run it, and save the output to a file. On my desktop, this program takes about 140 CPU seconds to run. What was your time?

5.1 Some small changes make a big difference

The function **update** takes a portion of the computational time. There is a single double loop there. It contains 12 floating point operations (additions and multiplications). Note that the computation of the position reads

```
pos(i,j) = pos(i,j) + vel(i,j) * dt + 0.5D+00 * acc(i,j) * dt * dt
```

but this is equivalent to the line

```
pos(i,j) = pos(i,j) + ( vel(i,j) + 0.5D+00 * acc(i,j) * dt ) * dt
```

where we have factored out one multiplication by **dt**.

Make this revision. Your new loop has 11 floating point operations. Rerun the program. Did the CPU time go down? Did it go down by a factor of roughly (11/12)?

If that change helped, then can you see any way to reduce the number of floating point operations to 10? Would it help to make a temporary variable equal to **0.5 * dt**?

5.2 Let an Autoparallelizer Look at the Code

For our last exercise, we are going to step up to the brink of using parallel programming. We are simply going to ask an automatic parallelizing feature of the compiler to look at some programs, and announce any loops that are candidates for parallel execution.

For this exercise, we will use the Intel compilers (icc, icpc or ifort) and so we must work on the SGI node, **charon3.arc.vt.edu**.

You use a modified compile statement to request the autoparallelization feature. Sample commands include:

```
icc -parallel md.c
ifort -fpp -parallel md.f
```

Get a copy of the codes for the **fft**, **heated_plate**, and **md** examples.

Compile each of these programs, and note the number of times the compiler indicates the existence of a parallelizable loop.

When I did this for the FORTRAN90 versions of the examples, the compiler found 0, 3 and 1 parallelizable loops, respectively. Your results might differ slightly.

During the coming lecture on OpenMP, we will see that we can parallelize loops that the autoparallelizer cannot. The autoparallelizing feature, by itself, can give you an indication of whether there are obvious opportunities for parallel execution in your program. A good compiler can also report why it did not consider parallelizing other loops.