1 What is Parallel Computing?

Using multiple processors in parallel (at the same time) to solve one large problem.

There are many different flavors of parallel computing, including shared memory, message-passing, SIMD (Single instruction, multiple data) (Note: a GPU is almost SIMD - it is SIMT (single instruction, multiple threads)).

1.1 Activity

Our goal is to add two vectors together. This is a toy example, so let’s suppose we have only 6 entries in each vector. If we were to solve the problem sequentially (i.e. if one person solves it by herself), what code construct would we use? A loop.

Let’s draw the three vectors on the board - that is our representation of memory. I will be the single processor. I compute one sum at a time and record the result.

```java
for (i = 0; i < 6; i++) {
    c(i) = a(i) + b(i);
}
```

Now, suppose we have more people who can work on it together. Let’s investigate three types of parallel computing. They are different in the way
they treat memory and in how independent the different processors/people are.

1. Shared Memory. Using something like POSIX threads with shared memory, we typically have a few processors, but not too many. So, for this example, we will use just two processors. [Get 2 volunteers]. We separate the problem into chunks - person A gets the first 3 entries, person B get the second three. Then the two people add their chunks of the vectors at their own paces. When both are done, the problem is solved.

2. SIMD (vector processing). For vector processing, there are typically lots and lots of processors, but they can’t behave independently – they must follow the same instructions at the same time. They simple have different data. So, let’s get 6 of us up at the board. Each of us is responsible for one sum. We all write on the board, but we must do everything in lock-step. I should add here that vector processing is done only on vector computers and that vector computers have fallen out of favor. However, the GPU is similar to a vector processor, so it is worth understanding SIMD.

3. Message-Passing (with the implication that memory is not shared). For message passing, we may have just a few processors, or we may have many processors. But the processors each have their own memory. Processors must send and receive messages to transfer data. Let’s suppose we have 3 processors. [Get 3 volunteers]. One processor is the “root” processor. It starts with all the data, sends the appropriate data to the appropriate processors, they perform their additions and send the results back to the root. Let’s do this with paper. [write it, distribute it, and collect it.]

What do I want you to learn from this course? I want you to learn how to “think in parallel”. When presented with a large problem, I want you to be able to answer the following questions:

- Can I speed up the code appreciably by using more processors?
- How can I divide my problem into computations that are, in some way, independent of each other?
• What types of parallel computation will benefit me? shared memory? distributed? etc?

• Given a hardware (and supporting software) set-up, what parallelization scheme is most appropriate?

• How can I minimize the amount of data transferred between processors (or between a computer and an external GPU)?

There are many problems that can be solved using parallel computing: weather forecasting, many bioinformatics problems (e.g. managing the large gene/protein databases, folding-at-home, biophysical modeling, optimization, etc.

2 Fish Schooling

We are going to focus on one particular problem - we are going to simulate a school of fish. Lots of times.

One of the most fascinating subjects in the life sciences is collective motion – how does a set of individuals make decisions about where to move? Let’s look at a 2007 National Geographic article entitled “The Genius of Swarms”. In it, there is an explanation by a biologist named Iain Couzin [read paragraph]. We are going to be implementing a model developed by him to track fish interacting with each other in a 2D ocean.

The basic idea, which was originated by a computer scientist in the 1980’s is this: We start with a bunch of fish. We simulated their movement through time. At each time step, each fish decides where it is headed, based on the position and heading of nearby fish. Each fish follows 3 simple rules:

1. Avoid crowding (very) nearby fish
2. Swim in the average direction of nearby fish
3. Stay close to nearby fish

The basic algorithm for each fish is to check for fish that are too close by. If there are any in this “zone of repulsion”, then move away. Otherwise,
look for fish a little further away, but that are within a “zone of attraction”. You want to move closer to those fish, but you also want to head in the direction they are heading. Those two desires may be in conflict, so choose some compromise heading.

[Demo a simulation with $r=0.25$, then with $r=16$].

Depending upon how you work that compromise between orientation and attraction, you will see different collective behaviors. That is the question we want to answer. How does the ratio of tendency-toward-orientation to tendency-toward-attraction affect the ability of the group to choose a common direction?

We have a parameter - the orientation-to-attraction ratio, which we will call $r$. We want to see its affect on the school.

To do this, we need to run the simulation on many different schools of fish for many different values of $r$. Why? We want to know the average behavior for lots of schools for each value of $r$. Thus, we must run many, many simulations.

So here is the first question. Why might we want to perform these computations in parallel? One reason is this: Each simulation is completely independent of every other simulation. Why not run multiple simulations at the same time? There is a term for this situation – this problem is what we call “embarrassingly parallel”. The simulations are independent, but then we need to combine the results and compute statistics about them. This is an ideal problem for a parallel computing course. It is not a toy problem - friends of mine have a journal article about their implementation of this solution on a GPU. But it is also straight-forward.

In this course, we will be writing code to simulate the schools and collect the appropriate statistics. First, sequentially, then in p-threads, and then on the GPU. We may or may not do so with MPI. We will step through this carefully and I will break the work up into manageable chunks.

3 Administrative Info

We will be working on NSCC and a set of computers called GPU0, GPU1, etc.
We will use Emacs as our editor. It has some interesting keyboard short-cuts. I have a summary of the most important ones on the website under “Emacs Hints”.

You will store your files on NSCC and your GPU computer. It is a good idea to back them up in your Personal directory.

We will be using the course website heavily.

Key points from the course website:

- programming projects once a week - due Thurs night at midnight. B or B+ for completing the required part. A for completing enough good extensions.
- office hours: W 2-4, R 2-5, F 1-3

4 C

C is a “low level” language - it operates very close to the hardware.

- minimalist language, but there are libraries
- compiling: preprocessing, compiling, linking
- header files, body files, and Makefiles
- you must manage the memory
- variables must have their types explicitly declared

Why would we use C for a parallel programming course?

We have on-line resources. Please take advantage of them.

4.1 Activity

We begin by logging on to the cluster. All nodes of the cluster share the same file storage space. Only the first node (n0) has the editor. We want to
execute our code on our node (n3 for now). So, we log onto the first node, start up the editor, then log onto another node to compile and run the code.

1. Log into your account on nscc using the command:
   ```bash
   ssh -Y nscc.colby.edu
   ```

2. The first time you log on, you need to run a program that will make it easier to travel between nodes.
   ```bash
   cluster_locksmith
   ```

3. Start emacs (always on this node)
   ```bash
   emacs &
   ```

4. Log onto your node using the command:
   ```bash
   ssh n5
   ```

Make a directory for today’s code: `mkdir examples`. 

Now, let’s write “hello, world”. In Emacs, open a file: press (C-x,C-f), then type `~/examples/hello.c`

Examples for today:

- Hello World: main program, printf with just a string
- simpleSum: input 2 floats, return a float. printf with a float.

I am mailing you the Makefile. Download it onto your Mac. The copy it to nscc using li!scp! (secure copy). In the Terminal, navigate to the directory that contains the Makefile, then copy it via

   ```bash
   sep Makefile nscc:~/examples/
   ```

You will be prompted for your password. Type it in, and the copy should work.

FYI, here are the contents of the Makefile
CC=gcc
CFLAGS=
LFLAGS= -lm

hello : hello.c
   $(CC) $(CFLAGS) $(LFLAGS) -o hello hello.c

addTest : addTest.c
   $(CC) $(CFLAGS) $(LFLAGS) -o addTest addTest.c

# Inference rules
% : %.c
   $(CC) $(CFLAGS) $(LFLAGS) -o $@ $^
Here are the contents of the C code files:

/* hello.c
 * You know what hello must do, don’t you?
 * Stephanie Taylor
 */

#include <stdio.h>

/* This is a major comment
 * It can go on for many lines.
 */
int main ( int argc , char * argv[] ) {
    printf("Hello, World!\n");
} // end main

/* addTest.c
 * Add two numbers and print them out.
 * Stephanie Taylor
 */
#include <stdio.h>

float add ( float n1 , float n2) {
    return n1+n2;
} // end add

int main(int argc, char *argv[]) {
    float answer;
    answer = add(5,2);
    printf("5+2 = %f\n",answer);
} // end main

Notice the file structure – .h files, .c files, and Makefiles. Walk through the includes and the norm function and tester.

To compile and run the program (on nscc), navigate to the examples directory and type:

make hello
./hello