Multi-core programming with Intel's Manycore Testing Lab

Intel Corporation has set up a special remote system that allows faculty and students to work with computers with lots of cores, called the *Manycore Testing Lab (MTL)*. In this lab, we will create a program that intentionally uses multi-core parallelism, upload and run it on the MTL, and explore the issues in parallelism and concurrency that arise.

Extra: Comments in this format indicate possible avenues of exploration for people seeking more challenge in this lab.

1 Requirements

It is recommended that you work on a non-lab machine (preferably a personal laptop) for this lab; you can still use the lab computers in order to develop our multi-core program, but we need to connect to the Intel MTL system using a non-lab computer. This is because the Cisco VPN software for connecting to the MTL blocks all other network access, so we can't use it on the lab computers or it would interfere with all other uses of those computers.

If you choose to use your own computer for this lab, you will need the following materials:

- A C++ compiler, if you do not have one on your computer already. GNU's gcc compiler is one such example, and is available on Windows, Mac OS, and Linux.
- A text editor or program to write and save C++ programs
- Cisco VPN client, which we will use to access the MTL. Instructions for installing Cisco VPN client for your machine is detailed in the next section.
- A terminal. Linux and Mac OS have UNIX-based terminals by default. For Windows, you also need `ssh` and `scp` capabilities. Putty and Cygwin are two ways to get these capabilities. With Putty, you'll need both `putty.exe` and `pscp.exe`. Ask for help if you need it.

2 Preparing your machine for the lab

Carry the steps in this section before the lab, if possible, on a laptop you can bring with you to the lab session.

The Cisco VPN client software is free, easy to install, easy to use, and does not interfere with your computer except while you are connected to the MTL. Here are download links (install before the lab if possible):

- Windows (2000,XP,Vista,7)

  Download the from this link and run the executable to install the Cisco VPN client.
- Mac OS 10.4 and later

  Download from this link, which creates a pseudodisk on your desktop, then open that pseudodisk and follow the instructions. (You can delete the pseudodisk when you're done.)

You can use a UNIX terminal window to access `ssh` and `scp`.

- For Linux and other versions of Windows and Macintosh, you can see the University of Ghent page that these downloads were obtained from. Choose among the "Cisco VPN clients without config file".

Once you have installed the Cisco VPN client (and have `ssh` and `scp` installed), configure the Cisco VPN client to access the multicore testing lab, as follows.

1. Start up the Cisco VPN client.

- *Note:* This is installed as a software application. *Don't* look for it under system network connection options, like other VPN systems.

2. Create a new connection, with the following connection information:
    - *Connection entry:* Choose a name, perhaps "Intel MTL VPN"
    - *Description:* Optional
    - *Host:* **192.55.51.80**
    - Select *Group Authentication* (probably the default)
    - *Name* is **VPN2**
    - *Password* is sent to you separately.

   Save this connection to finish creating it.

3. Now try connecting on your new VPN connection entry.
    - If this succeeds, you will find that *none of your usual network services work.* For example, your browser won't be able to find any pages (thus, you'll have to use a different machine while you're connected to the MTL if you need to access network services).
    - If your new VPN connection fails, recheck the settings you entered, or seek help.

4. Finally, disconnect from your new VPN connection entry. This will give you your usual network capabilities back, etc.

3 Intel's Threading Building Blocks (TBB)

OpenMP works well for adding parallelism to loops in working sequential code, and it's available for C, C++, and Fortran languages on many platforms (including Linux, Windows, and Macintosh OS X). Older versions of OpenMP did not readily support non-loop parallelism or programming with concurrent data structures, but OpenMP version 3.0 (released May 2008) provides a *task* feature for programming such computations.

Intel's Threading Building Blocks (TBB) provides an object-oriented approach to implementing parallel algorithms, for the C++ language (and any of the three platforms). Adding parallelism to existing code in TBB is somewhat more involved than in OpenMP, but is considerably less complicated than programming in a native threads package for a particular operating system. The forthcoming new standard for the C++ language is likely to include parallelism similar to TBB.

1. Enter the following TBB program into a file `trap-tbb.cpp`.

```
#include <iostream>
#include <cmath>
using namespace std;

#include "tbb/tbb.h"
using namespace tbb;

/* Demo program for TBB: computes trapezoidal approximation to an
integral*/

const double pi = 3.14159265358979323846264338327 9;

double f(double x);

class SumHeights {
  double const my_a;
  double const my_h;
  double &my_int;

public:
  void operator() (const blocked_range<size_t>& r) const {
    for(size_t i = r.begin(); i != r.end(); i++) {
      my_int += f(my_a+i*my_h);
    }
  }

  SumHeights(const double a, const double h, double &integral) :
    my_a(a), my_h(h), my_int(integral)
  {}
};

int main(int argc, char** argv) {
   /* Variables */
   double a = 0.0, b = pi;  /* limits of integration */;
   int n = 1048576; /* number of subdivisions = 2^20 */

   double h = (b - a) / n; /* width of subdivision */
   double integral; /* accumulates answer */

   integral = (f(a) + f(b))/2.0;

   parallel_for(blocked_range<size_t>(1, n), SumHeights(a, h,
integral));

   integral = integral * h;
   cout << "With n = " << n << " trapezoids, our estimate of the
integral" <<
     " from " << a << " to " << b << " is " << integral << endl;
}

double f(double x) {

   return sin(x);
}
```

Comments on this code:

- This program does *not* use a command-line argument (and the `cstdlib` library is not needed). Unlike OpenMP, TBB does not provide a simple way to request a particular number of threads. Instead, the TBB system chooses a number of threads to use automatically. (OpenMP will also make such a selection for you if you do not specify the number of threads to use.)
- The following lines prepare for using TBB.

```
#include "tbb/tbb.h"
using namespace tbb;
```

- Recall that in the OpenMP code, we parallelized the loop below by adding a pragma just before that `for` loop.

```
for(i = 1; i < n; i++) {
  integral += f(a+i*h);
}
```

- In order to program a comparable computation in TBB, we create a class `SumHeights` whose method `operator()` contains the following loop:

```
for(size_t i = r.begin(); i != r.end(); i++) {
  my_int += f(my_a+i*my_h);
}
```

  then passes an instance of that class `SumHeights` to a call of `parallel_for()`. Observe that the forms of the two loops indicate the same iterative computation, if one matches `1` to `r.begin()`, `n` to `r.end()`, variables and `integral`, `a`, and h to `SumHeights` state variables `my_int`, `my_a`, and `my_h`.

  One way to describe this relationship is to say that the class `SumHeights` is a "wrapper" around its loop.

  i. The class `SumHeights` defines `operator()`, which means that an object of type `SumHeights` can be called using function-call syntax. Since `operator()` is defined here with one argument, this means we can cause the `for` loop to execute using a call `sh(range)`, where `sh` is an object of type `SumHeights` and `range` is an appropriate argument.
  ii. Note that `operator()` is a `const` method (indicated by the `const` after `)` and before `{`), which means that it is permitted to call `sh(range)` with a `const` object `sh`.
  iii. The argument `r` of `operator()` indicates the *range* of the loop, i.e., the starting and ending values for the loop control variable.
  iv. The loop control variable `i` has type `int` in the OpenMP implementation, but type `size_t` in the TBB implementation. `size_t` is an integer type, which may be equivalent to `int`, `long`, or another integer type depending on implementation.
  v. The constructor `SumHeights()` makes local copies `my_a`, etc., of variables `a`, etc., in `main()`, enabling values in `main()` to be used within the class `SumHeights`.

- The range `r` has the type `blocked_range<size_t>`. This is a *templated type* built over the `size_t` type. There could be `blocked_range` types built over other types, as well, e.g., `int` or `long`.
- The call to `parallel_for` in `main()` automatically subdivides (or *chunks*) the range `r` for multi-threaded parallel computation. `parallel_for` expects a range in its first argument, and an object with a method `operator()` having one range argument in its second argument.
- The variable `integral` is passed by reference in the constructor `SumHeights()` in an effort to use that memory location `integral` as an accumulator during the parallelized computation.
- The constructor initializes the state variables `my_a`, `my_h`, and `my_int` using *colon initializers*. In the constructor definition

  ```
  SumHeights(const double a, const double h, double &integral) :
      my_a(a), my_h(h), my_int(integral)
    {}
  ```

  the expression `my_a(a)` located after the colon `:` and before the curly bracket `{` has the same effect as an assignment

  ```
  my_a = a;
  ```

  would if it occurred *between* the curly brackets. Colon initialization is optional for the state variables `my_a` and `my_h`, but it is required for the state variable `my_int`, because that state variable was defined using a reference type.

Note: Can you detect any problems in this code?

2. For this lab, we will run this TBB program on the MTL.

   - First, if necessary, copy the program file you created to a local machine for connecting to MTL, e.g., your laptop.
   - If your code is on a lab machine, use the following command from the terminal on your local machine (where "username" is your username, "labmachinename" is the name of the machine on which you are working, and "file_location" is the location of `trap-tbb.cpp` within your directory):

     ```
     laptop% scp username@labmachinename:file_location/trap-tbb.cpp .
     ```

     (Don't forget the final dot!)
   - Now connect to the MTL's network using the Cisco VPN client software on that local machine. Login to the MTL computer, as follows:

     ```
     laptop% ssh accountname@36.81.203.1
     ```

     Use one of the student account usernames provided to you, together with the password distributed to the class.
   - Create a subdirectory for yourself in the home directory for that student account name:

     ```
     36.81.203.1% mkdir username
     ```

Here, *username* can be your username.

- Use `scp` to copy your file to your subdirectory in the MTL system:

  ```
  laptop% scp trap-omp.C accountname@36.81.203.1:username
  ```

  After making this copy, login into the MTL machine `36.81.203.1` again.

- On the remote MTL system, execute the following command, which sets up environment variables for compiling with TBB:

  ```
    36.81.203.1% source
  /opt/intel/Compiler/11.1/056/tbb/bin/tbbvars.sh intel64
  ```

  The `intel64` command-line argument prepares for 64-bit compilation.

- To compile your program that was copied in a prior step, issue this command:

  ```
  36.81.203.1% g++ -o trap-tbb trap-tbb.cpp -ltbb_debug
  ```

  Note: You can use `-ltbb` instead of `-ltbb_debug` for a production version of the library instead of one with debugging hooks.

- Now run your program with the following command:

  ```
  36.81.203.1% ./trap-tbb
  ```

  The result is significantly less than 2! Can you think of an explanation for the answer being so far off?

  Also run several time tests of your program

  ```
  36.81.203.1% time ./trap-tbb
  ```

  What do you observe in these time tests? How do the times compare to timed runs of `trap-omp` for various thread sizes?

4 TBB, multiple threads, and reduction

The code above for a TBB trapezoidal computation produces an incorrect answer if there are multiple threads, because each thread attempts to update the shared variable `integral` without any mechanism to avoid one thread from overwriting the results produced by another thread. We will solve this issue using a *reduction,* in which results will be computed in *local* variables for each thread, then those local results added together at the end.

1. To do the reduction in TBB, we will use the `parallel_reduce` call instead of the `parallel_for` call, and will use a modified class `SumHeights2`.

   ```
   #include <iostream>
   #include <cmath>
   using namespace std;

   #include "tbb/tbb.h"
   using namespace tbb;

   /* Demo program for TBB: computes trapezoidal approximation to an
   integral*/
   ```

```
const double pi = 3.141592653589793238462643383079;

double f(double x);

class SumHeights2 {
  double const my_a;
  double const my_h;

public:
  double my_int;

  void operator() (const blocked_range<size_t>& r) {
    double a2 = my_a;
    double h2 = my_h;
    double int2 = my_int;
    size_t end = r.end();
    for(size_t i = r.begin(); i != end; i++) {
      int2 += f(a2+i*h2);
    }
    my_int = int2;
  }

  SumHeights2(const double a, const double h, const double integral) :
    my_a(a), my_h(h), my_int(integral)
  {}

  SumHeights2(SumHeights2 &x, split) :
    my_a(x.my_a), my_h(x.my_h), my_int(0)
  {}

  void join( const SumHeights2 &y) { my_int += y.my_int; }
};

int main(int argc, char** argv) {
   /* Variables */
   double a = 0.0, b = pi;  /* limits of integration */;
   int n = 1048576; /* number of subdivisions = 2^20 */

   double h = (b - a) / n; /* width of subdivision */
   double integral; /* accumulates answer */

   integral = (f(a) + f(b))/2.0;

   SumHeights2 sh2(a, h, integral);
   parallel_reduce(blocked_range<size_t>(1, n), sh2);
   integral += sh2.my_int;

   integral = integral * h;
   cout << "With n = " << n << " trapezoids, our estimate of the
integral" <<
     " from " << a << " to " << b << " is " << integral << endl;
}

double f(double x) {

   return sin(x);
}
```

Comments:

- The class `SumHeights2` handles the variable `my_int` differently than the class `SumHeights` does. Instead of `SumHeights`'s misguided attempt to share `main()`'s memory location `integral` through reference types, the new class `SumHeights2` allocates a new separate (state) variable location `my_int` for each object of type `SumHeights2` (by avoiding reference types).
- Also, `my_int` is a `public` state variable in `SumHeights2`, instead of the default `private` visibility in the prior class `SumHeights`. This makes it possible for a method of a `SumHeights2` object to compute a value and store that value in `my_int`, then for another part of the code to access that computed value through that public state variable `my_int`. (Alternatively, we could have made `my_int` private like the other state variables, and added a "getter" method `get_my_int()` to retrieve that computed value.)
- The `operator()` definitions in the two classes differ in several ways.
  i. The code for the new class's operator `SumHeights2::operator()` begins my making local copies `a2`, `h2`, and `int2` of the variables `my_a`, `my_h`, and `my_int`, and also storing the (unchanging) value of `r.end()` in another local variable. These local variable assignments are *not* necessary for the logical correctness of the code. Instead, they make it possible for the compiler to produce a more efficient computations. With this help, the compiler can realize that it's safe to use *registers* to implement those variables *instead of memory locations*, which would lead to faster access to those values.
  ii. The loop is rewritten to use these local variables, but otherwise represents the same computation as in the previous `SumHeights::operator()`.
  iii. *After* the loop, the local variable `int2` is assigned to the state variable `my_int`, in order to deliver the sum for this thread's subdivision (chunk) of the summation range.
  iv. `SumHeights2::operator()` is *not a* `const` *method*. This means it's not safe for `const` objects to call this method -- they will be changed. In this case, the change is that `my_int` is changed when `operator()` is called.
- The three-argument constructor for `SumHeights` is the same as the three-argument constructor for `SumHeights2`, except for the handling of the third argument `integral` (discussed above).
- However, the class `SumHeights2` has an additional constructor and an additional method `join()`.
  i. The second constructor is called a *split constructor*. This constructor will be used to construct new instances of `SumHeights2` for additional threads brought into the summation computation.
  ii. The method `join()` is used to add the partial sum from one thread's computation to a running sum -- i.e., to perform the reduction operation.
- Here is an overview description of the parallel computation for this program.
  i. An object `sh2` is allocated, using the three-argument constructor for `SumHeights2`.

ii. The call to `parallel_reduce()` in `main()` performs `sh2`'s `operator()` over the range 1 to `n` by subdividing (i.e., chunking) that range and assigning a thread to perform the trapezoidal sum for each chunk.

iii. Each of those threads creates its own `SumHeights2` object using the splitting constructor.

- The thread first calls that splitting object's `operator()` with that thread's range chunk to compute a partial sum.
- Then, the thread calls `sh2.join()` with that splitting object as the argument, in order to add its partial sum to `sh2`'s accumulator `sh2.my_int`.

iv. After all range chunks have been processed, `parallel_reduce()` finishes, leaving the final answer in the `public` state variable `sh2.my_int`.

- The splitting constructor for `SumHeights2` has a dummy argument of type `split` (defined by the TBB library), because without that extra argument, there would be no way for a compiler to tell the difference between a call to that splitting constructor and a call to `SumHeight2`'s copy constructor.

2. Enter the program above, using the filename `trap-tbb2.cpp`.

   You can enter it on a lab machine, but then you'd have to disconnect Cisco VPN on your local machine (e.g., your laptop), `scp` the new file to your local machine, reconnect Cisco VPN, and `scp` to the MTL machine in order to transfer it to the MTL system.

   Alternatively, you could enter the program on your local machine and `scp` to the MTL machine. Another possibility is to enter it on the remote MTL system directly, using an editor such as `emacs` or `vi`.

3. Compile and test your `trap-tbb2` program on the MTL. Does it now produce the correct answer of 2 for the trapezoidal approximation?
4. Also time the performance of runs of this revised program, and compare to the time performance of runs of the prior program `trap-tbb`.