UNIVERSITY OF WASHINGTON
Department of Applied Mathematics

AMATH 483/583 **Assigned:** 5/11/2017
Problem Set #6 **Due:** 5/16/2017

---

1 Preliminaries

For command-line programs (like the ones we have been writing in this course), the user of a program can pass parameters to it on the command line and they are "automagically" put into the `argv` array. This is obviously a very useful way for a user of a program to state their intentions to the program. Often however, the program itself needs to find out information about the environment in which it is executing – and this may be information that the user might not know or care about or that would be inconvenient to pass on the command line every time a program is run.

To enable a program to obtain information about its environment, most operating systems have a simple look up mechanism that associates variable names with values (though these values are also just strings). One environment variable that we have discussed previously in the class was the status variable (i.e., the return value from a `main()` function).

From the `bash` command line, if we want to retrieve the value associated with an environment variable (if we want to evaluate it) we prepend a `$` sign to it. In `bash` the status variable was retrieved with `$?`, though in the `csh` shell, it is retrieved with `$status`. As with program variables, evaluating a variable is not the same as printing it out. If you just type

```
$ $?
```

at the command line, you will get the cryptic response

```
0: Command not found
```

because the shell evaluated `\$?`, which evaluated to `0`. This in turn was evaluate by bash just as if we had typed in

```
$ 0
```

and since `0` is (probably) not an executable command, the shell prints its error message. If you want to examine the value of an environment variable in the shell, you need to pass that value to a program that will just print its argument back out. In most Unix like environments, this command is `echo`:

```
$ echo $?
0

$ echo $SHELL
/bin/bash
```

Now, there are actually two overlapping namespaces in most command-line shells: the environment variables and the shell variables. These variables are accessed in the same way, with the `$` prefix – but only the environment variables are available to running programs. The shell variables are used only by the shell program itself.

To see a list of all of the environment variables, issue the command:

```
$ env
```

Note that this is an executable program, not a built-in shell command, so it will retrieve and print the list of environment variables that are accessable by all programs. Try this yourself. You should see a couple dozen lines that look like this (I am just showing a few):

```
HOSTNAME=259e82e56952
SHELL=/bin/bash
TERM=xterm
```

```
HOME=/home/amath583
PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin
PWD=/home/amath583
SHLVL=1
```

These variables in the environment are not special in any particular way. Again, the environment just maintains an association between variable names (e.g., `HOME`) and a value (e.g., `/home/amath583`). There are conventions established about certain environment variables are used by different programs. And, some programs will update the environment. For instance if you query the value of `PWD`:

```
$ cd /
$ echo $PWD
/
$ cd
$ echo $PWD
/home/amath583
```

so the name of the current working directory is updated whenever you change directories.

Another important environment variable is the path (`PATH`). This contains a colon separated list of directories. When you enter a command on the command line, the shell will search for a program that matches the name you typed in each of those directories. If it finds a match, it will execute that program (using `fork()` and `exec()` as we have discussed in lecture). If it does not find a match it will return an error message (and set the status to a non-zero value).

But how does the bash shell – which is just a program – get the value of the environment variables? The C++ standard library provides a function for looking up environment variables.

Disclaimer. As with `main()` when passed arguments, the function for querying the environment deals with character pointers. I will first explain the function itself and then give you a wrapper function that you can use so that you don't have to actually deal with pointers.

The function for querying the is `std::getenv()`, prototyped as follows:

```
char* std::getenv(char *str);
```

The function will look for a variable whose name matches the character string passed in as `str`. If there is a match, it will return a pointer to the string value in the environment. If a match is not found – and this one of the appalling things about pointers, it will return an error, but will indicate that error by setting the value of a pointer to a special value, namely `NULL` (or zero). NULL pointers are no end of trouble and have probably set the software community back a generation. And they are a problem (this a general problem with pointers – but NULL is the most common) because when a programmer has a pointer they want to de-reference it. If for some reason the pointer is `NULL` or some other invalid value, the program will throw an exception – a segmentation violation – and it will abort. Since pointers can be manipulated just like any other variable in C, it is easy to give one an invalid value. Moreover, as with `getenv`, the special (and dangerous) value of NULL is often used to indicate an error condition for function calls that return pointers. If that value is not checked, and then later used, the rare occasion that there is a failure in that call will result in a segmentation violation.

If you are curious, try this program

```
int main() {
  char* ptr = 0;
  return *ptr;
}
```

So that you can avoid dealing with pointers (and you should), I provide the following wrapper.

```
#include <iostream>
#include <cstdlib>
#include <string>
```

```
std::string getenv_to_string(const char *in) {
  char *gotten = std::getenv(in);
  if (NULL == gotten) {
    return std::string("");
  } else {
    return std::string(gotten);
  }
}

std::string getenv(const std::string& in) {
  return getenv_to_string(in.c_str());
}
```

Just pass in a C string literal (pointer to char) or a `std::string` and it will return a string.

Experiment with executing `env` and examining some of the variables individually. Create some new variables and examine their values. In a fresh docker container, you might try the following (as an example of what to be careful about):

- What do you get when you issue the command `echo $PATH`?
- What happens when you issue the command `export PATH=""`?
- What happens when you issue the command `echo $PATH` following the command above?
- `^D` may come in handy after the above.

2 Warm Up

As we have seen in lecture, OpenMP provides a compiler directive based mechanism for parallelizing a job for shared-memory execution. One of the principles upon which OpenMP is based is that the program source code does not need to be changed – the compiler, as controlled by *`#pragma omp`* statements, manages all concurrency. At the same time, there are parameters about the running parallel program that we would like the parallel program to be aware of. For example, we might want to specify the number of threads that a program uses to execute.

Now, with the OpenMP philosophy, the program itself can be both sequential and parallel – and it should largely be "unaware" of its parallelization. (This should be true for any parallel program – concerns should be well-separated.) Rather than using command-line arguments as a way to pass information to OpenMP, OpenMP uses environment variables. Environment variables also have the advantage (compared to command-line arguments) that they can be accessed from any arbitrary part of the program – including from a library. You can find a list of environment variables that OpenMP uses on the OpenMP web site or the OpenMP quick reference (`http://bit.ly/2pCSigX`). The most important environment variable is `OMP_NUM_THREADS`, which sets the (maximum) number of threads that OpenMP will use at run-time. **NB:** It is important to remember that environment variables don't actually do anything. Setting the value of `OMP_NUM_THREADS` to some value doesn't automatically parallelize your program or *cause* OpenMP to have that many threads. Rather it is a variable whose value any program can read – it is up to the program what action to take (if any) in response to the value it reads.

2.1 Compiling and Running OpenMP

IMPORTANT! The clang compiler we have been using for the course in docker is compatible with OpenMP. However, a few additional libraries are also necessary. To be able to use OpenMP, you need to run the docker image named `amath583/openmp`.

Compiling with OpenMP is fairly straightforward – simply add a flag `-fopenmp` to the compilation and linking invocations of the compiler. E.g.,

```
c++ -fopenmp -c hello.cpp -o hello.o
c++ -fopenmp hello.o hello
```

Note that in the version of clang we are using, there is some incompatibility between clang and the cmath include file – this problem will only show up if you compile with the `-Ofast` compilation flag in addition to `-fopenmp`. (The conflict has to do with the recently introduced SIMD commands in OpenMP 4.0.) If you get a page full of errors for what should be a correct program, change `-Ofast` to `-O3`.

When a program is compiled with OpenMP, it can be run just as with any other program – execute it on the shell command line.

2.2 Hello OMP World

When running a parallel program, there are several pieces of information we would like to have. For example, in the last assignment, we partitioned our work into pieces and let a thread or task execute each piece (in parallel). For that assignment, we specified the number of threads / tasks to use so that we could experiment with different numbers of threads. In general, however, we don't want the user to just arbitrarily set the number of threads that a job runs with. We might not want to run with more threads than there are cores available, for example. Or, we might want to set some resource limit on how many threads are used (which might be more or less than the number of cores). That is, we want to let the runtime system have some control over how our parallel program runs (and doing this automatically is in the spirit of OpenMP).

Consider the following program:

```
int main() {

  std::string envName = "OMP_NUM_THREADS";
  std::cout << envName << "        = " << getenv(envName) << std::endl;

  std::cout << "hardware_concurrency() = " <<
        std::thread::hardware_concurrency() << std::endl;
  std::cout << "omp_get_max_threads()  = " <<
        omp_get_max_threads() << std::endl;
  std::cout << "omp_get_num_threads()  = " <<
        omp_get_num_threads() << std::endl;

  return 0;
}
```

Note that we are using the `getenv()` wrapper we presented earlier.

When I run this in my docker container I get the following output:

```
./a.out
OMP_NUM_THREADS        =
hardware_concurrency() = 4
omp_get_max_threads()  = 4
omp_get_num_threads()  = 1
```

In this case, `OMP_NUM_THREADS` is not set so OpenMP uses default values. However, we have to be careful how we interpret this. The first line shows that `OMP_NUM_THREADS` is not set. The second line is obtained with a call to the C++ standard library (outside of OpenMP) and shows that the available hardware concurrency (number of cores) is equal to four. The next two lines were obtained with calls to the OpenMP library. This shows that the maximum available number of OpenMP threads is equal to four. The final line, however, may seem curious – it says that the number of threads is equal to one.

If we take a step back, though, number of threads equal to one makes sense in the context where it is called. Recall from lecture that OpenMP is based on the fork-join model. It runs sequentially between OpenMP directives, the code blocks following which are run in parallel. In `main()` where the call to

omp_get_num_threads, there is only one running thread – there are no other threads running concurrently with it. But, how many threads actually run at one time with OpenMP? And how can we tell?

Let's add a simple function to the above:

```
int omp_thread_count() {
  int n = 0;
#pragma omp parallel reduction(+:n)
  n += 1;
  return n;
}
```

This function will create the default number of threads and each one of those will increment the variable n by one – the final value of n will be the number of threads that executed in parallel. If we add a call to omp_thread_count() to the above program we obtain as output:

```
./a.out
OMP_NUM_THREADS         =
hardware_concurrency()  = 4
omp_get_max_threads()   = 4
omp_get_num_threads()   = 1
omp_thread_count()      = 4
```

And, indeed, the number of threads that were run in parallel is four.

The code for this example is available as hello_omp_thread.cpp.

2.3 OpenMP is Still Threads

One advantage as programmer to using OpenMP is that parallelization can be done in line. When we parallelized programs explicitly using threads and tasks in C++, we had to bundle the work we wanted to be done into a separate function and launch a thread or task to execute that function. With OpenMP we can parallelize directly in line. For example, we can create a parallel "Hello World" as follows:

```
int main() {
  std::cout << "omp_get_max_threads()  = " <<
        omp_get_max_threads() << std::endl;
  std::cout << "omp_get_num_threads()  = " <<
        omp_get_num_threads() << std::endl;

 #pragma omp parallel
  {
   std::cout << "Hello! I am thread " << omp_get_thread_num() <<
        " of " << omp_get_num_threads() << std::endl;
    std::cout << "My C++ std::thread id is " << std::this_thread::get_id() <<
          std::endl;
   }

  return 0;
}
```

There is no helper function needed.

But, OpenMP is not a panacea. Underneath the covers, OpenMP runs threads. This is the output from one run of the above program:

```
$ ./a.out
omp_get_max_threads()  = 4
omp_get_num_threads()  = 1
Hello! I am thread Hello! I am thread 0 of 4
```

```
Hello! I am thread 2 of 4
My C++ std::thread id is 140325565663104
My C++ std::thread id is 140325592336192
3 of 4
My C++ std::thread id is 140325561464768
Hello! I am thread 1 of 4
My C++ std::thread id is 140325569861440
```

The first thing to notice is that we can get the thread id for the current thread with C++ standard library mechanisms. Problematically, note that the strings that are being printed are interleaved with each other. We have seen this before in multithreading – when we had race conditions.

Just as with the explicit threading case, OpenMP can have race conditions when shared data is being written. However, as with the mechanisms for parallelization, the OpenMP mechanisms for protecting data are also expressed as `#pragma omp` directives. We can fix this race by adding a `#pragma omp critical`. Note that this is more general than the mutex approach we had before. We are just telling OpenMP that there is a race – not how to protect it.

```
int main() {
  std::cout << "omp_get_max_threads()  = " << omp_get_max_threads() << std::endl;
  std::cout << "omp_get_num_threads()  = " << omp_get_num_threads() << std::endl;

 #pragma omp parallel
 #pragma omp critical
  {
   std::cout << "Hello! I am thread " << omp_get_thread_num() << " of " <<
          omp_get_num_threads() << std::endl;
    std::cout << "My C++ std::thread id is " << std::this_thread::get_id() <<
          std::endl;
   }

  return 0;
}
```

The code for this example is contained in the file `hello_omp.cpp`. In this example, there are three commented lines with `#pragma omp` lines.

```
  // #pragma omp parallel
  // #pragma omp parallel num_threads(2)
  // #pragma omp critical
  {
 std::cout << "Hello! I am thread " << omp_get_thread_num() << " of " <<
        omp_get_num_threads() << std::endl;
  std::cout << "My C++ std::thread id is " << std::this_thread::get_id() <<
        std::endl;
   }
```

Experiment with this program a little bit until you get a feel for how OpenMP is managing threads and a feeling for how the OpenMP concepts of threads etc map to what you already know about threads. You might try the following:

- Uncomment the first line. Try different values of `OMP_NUM_THREADS`. You should see different numbers of "Hello" messages being printed. What happens when you have more threads than hardware concurrency? Are the thread IDs unique?
- With the first line uncommented, see what the difference is between having the second line uncommented or not.
- Comment out the first line and uncomment the second. How many threads get run? What happens when you have different values of `OMP_NUM_THREADS`?

3 Exercises

3.1 Norm

In problem sets 2 and 5 we wrote some functions for computing the norm of a vector. Now we are going to use OpenMP for concurrency and (hopefully) parallelism to speed up computation of the norm. For the implementation of the vector and for the sequential norm calculation you may use your previous implementation(s) or the instructors implementation. As has been the practice so far in the course, the interface for the vector type should be in a file `Vector.hpp` and the implementation in `Vector.cpp`.

Write a function with the following prototype:

```
double ompTwoNorm(const Vector& x);
```

The function takes `x` as an argument, computes its two norm (Euclidean norm) and returns that value. For your implementation, you should start with your sequential function for computing Euclidean norm and add `#pragma omp` directive(s) to *safely* parallelize your computation.

As a driver program for this exercise, create a file `pt2n_driver.cpp`. That file should contain a `main` function that reads two arguments from the command line. The first is the size of the vector to take the norm of. The second is the number of times to run the norm computation inside of the timer calls. That is, if you pass in 10 as the number of times, you should time ten runs of `ompTwoNorm()` and print the total elapsed time **divided by 10**.

A before, your program should print a line of tab (or space) separated numbers. The first element in the line should be the size of the vector, the second the sequential execution time, the third the parallel execution time, the fourth the speedup, and finally the difference between the computed norms.

Deliverables

- `Vector.hpp` containing the interface for the `Vector` type, including the `ompTwoNorm` function prototype
- `Vector.cpp` containing the implementation for the above interface
- The driver `pt2n_driver.cpp` as described above
- A `Makefile` that enables `make Vector.o` to create an object file for the `Vector` type (i.e. `Vector.cpp`) – parallelized with OpenMP
- A `Makefile` that enables `make pt2n_driver` to create an (optimized) and parallelized executable for the driver program

Recall that any warnings from `-Wall` or the use of `using namespace std;` in `Vector.hpp` puts you at risk to lose points.

3.2 Sparse Matrix Computation with OpenMP

You've been provided with the `COOMatrix` class in the `COO.cpp` and `COO.hpp` files. Create a free function with the following prototype:

```
void ompMatvec(const COOMatrix& A, const Vector& x, Vector& y);
```

Note that the prototype should go in `COO.hpp` and implementation in `COO.cpp`. As with the sequential version, this function should immediately dispatch to a member function of the `COOMatrix` class, which you are to write. The core of it is as follows:

```
void ompMatvec(const Vector& x, Vector& y) const {
  for (size_type k = 0; k < arrayData.size(); ++k) {
    y(rowIndices[k]) += arrayData[k] * x(rowIndices[k]);
  }
}
```

You are asked to parallelize this function using `#pragma omp` directive(s) to *safely* parallelize your computation.

As a driver program for this exercise, create a file `coo_driver.cpp`. That file should contain a `main` function that reads one argument from the command line: the size of the problem. We assume the matrix is square and the vectors are the same length (as specified by this argument). The matrix and vectors should be constructed with the given sizes. The vectors should be randomized or zeroized, accordingly, and the matrix should be piscetized. The function should first time and compute the matrix vector product using the sequential function we developed earlier. It should then time and compute the omp version. For smaller problems that can be computed in less than a millisecond, it is suggested that you run the computation multiple times (as we have done in previous assignments). Finally, your program should print a line of tab (or space) separated numbers. The first element in the line should be the number of cores available, the second should be the number of threads your program used when executing its parallel region, the third should be the size of the problem, the fourth the sequential execution time, the fifth the parallel execution time, the sixth the speedup, and finally the difference between the two norms of y computed sequentially and y computed in parallel.

For testing we will run your driver program with varying number of threads as given by the environment variable `OMP_NUM_THREADS`. Your program should run with the maximum number available.

Deliverables

- `COO.hpp` containing the implementation of the member function `ompMatvec` and the interface declaration for the free function `ompMatvec`
- `COO.cpp` containing the implementation of free function `ompMatvec`
- The driver `coo_driver.cpp` as described above
- A `Makefile` that enables `make COO.o` to create an object file for the `COOMatrix` class (i.e. `COO.cpp`)
- A `Makefile` that enables `make Vector.o` to create an object file for the `Vector` class (i.e. `Vector.cpp`)
- A `Makefile` that enables `make coo_driver` to create an (optimized) executable for the driver program

3.3 CSR Sparse Matrix (AMATH583 Only)

Repeat the exercise above in Section 3.2 except for `CSRMatrix` instead of `COOMatrix`. Note that you've been provided `CSR.hpp` and `CSR.cpp` to start with.

3.4 Written Questions

There are no written questions for this homework assignment.

4 Turning in The Exercises

All your cpp files, hpp files, and your `Makefile` should go in the tarball `ps6.tgz`. If necessary, include a text file ref6.txt that includes a list of references (electronic, written, human) if any were used for this assignment.

Before you upload the `ps6.tgz` file, it is **still very important** that you have confidence in your code passing the automated grading scripts. After testing your code with your own drivers, make use of the python script test_ps6.py by using the command `python test_ps6.py` in your ps6 directory. Once you are convinced your code is exhibiting the correct behavior, upload ps6.tgz to Collect It.

5 Learning Outcomes

At the conclusion of week 7 students will be able to

1. Explain the basic parallelization strategy of OpenMP
2. Use `#pragma omp parallel` to parallelize a small block of code.
3. Explain the environment variable `OMP_NUM_THREADS` and what it does
4. Use `#pragma omp parallel for` to parallelize a loop.
5. Translate a short program using `#pragma omp parallel` into the equivalent C++. `std::thread` implementation.
6. Use `#pragma omp parallel for reduction` to implement a parallel accumulation algorithm, for addition and multiplication.
7. Specify the number of threads to be used in parallelization by a `#pragma omp` directive.
8. Use OpenMP functions to get and set the number of threads used by a program.
9. Explain the `#pragma` directive.
10. Use `#pragma omp task` for a simple program.
11. Explain how and where to use `#pragma omp critical`
12. Explain how and where to use `#pragma omp atomic`