UNIVERSITY OF WASHINGTON
Department of Applied Mathematics

AMATH 483/583 **Assigned:** 5/5/2017
Problem Set #5 **Due:** 5/10/2017

1 Preliminaries

1.1 Roofline Parameters

As discussed in lecture, the roofline model depends on the memory and CPU characteristics of the system being modeled. Although one could look up various parameters about your system, it is more accurate to measure the actual system.

A docker image that will profile your computer and create a roofline plot has been put docker hub. When running this image will put its results into a subdirectory called "ERT". So, before running this image, first create a subdirectory named ERT in the directory you normally work in with docker. Then you can run the image.

```
$ cd <your working directory>
$ mkdir ERT
$ docker run -v <your working directory>:/home/amath583/work amath583/ert
```

The `<your working directory>` stands for the location on your host machine where you have deen doing your work under docker.

Note that this will run for quite some time – 20-30 minutes – during which time your computer should have as little load on it as possible. I.e., this docker image should be the only thing running on your machine. The traditional activity when waiting for a long-running computational task was to get a cup of coffee.

When you are back from coffee and the job is completed, it will have created a small trove of the data it generated. Most interesting will be in the subdirectories named Results. In that subdirectory are one or more subdirectories with the name Run.001. Directly in the Run.001 subdirectories is a file roofline.pdf, which contains a graph of the roofline model for your computer. **NB:** If you do multiple runs of the roofline profiler, delete the subdirectory Run.001 each time so that the profiler has a fresh directory to write to and there won't be old or incomplete results to confuse you or to confuse the profiler.

1.2 Bandwidth and Hierarchical Memory Parameters

A second (related) characterization of your machine is to measure the bandwidth for various sizes of data and with various types of data movement instructions.

A docker image that will profile your computer and create a bandwidth plot has been put docker hub. When running this image will put its results into a subdirectory called "BW. So, before running this image, first create a subdirectory named BW in the directory you normally work in with docker. Then you can run the image.

```
$ cd <your working directory>
$ mkdir BW
$ docker run -v <your working directory>:/home/amath583/work amath583/bandwidth
```

The `<your working directory>` stands for the location on your host machine where you have deen doing your work under docker. Note that as with the roofline program, the bandwidth program will run for quite some time – 20-30 minutes – during which time your computer should have as little load on it as possible.

When the program completes, there will be two files in the BW directory – bandwidth.png and bandwidth.csv. The file bandwidth.png is an image of a graph showing data transfter rates for various types of operations and various types of operations. The file bandwidth.csv is the raw data that is plotted in bandwidth.png, as comma separated values. You can use that data to do your investigations about your computer.

In looking at the bandwidth graph you will see one or more places where the transfer rate drops suddenly. For small amounts of data, the data can fit in cache closer to the CPU – and hence data transfer rates will be higher. The data transfer rates will fall off fairly quickly once that data no longer completely fits into cache. Based on the bandwidth plot obtained for your computer, how many levels of cache does it have? How large are each of the caches (at what sizes does the bandwidth fall)? If possible, check your estimates against published information for the microprocessor that your system is using.

2 Warm Up

2.1 Futures

C++ provides the `std::async()` mechanism to enable execution of a function call as an asynchronous task. Its semantics are important. Consider the example we have been using in lecture.

```
typedef unsigned long size_t;

size_t bank_balance = 300;

static std::mutex atm_mutex;
static std::mutex msg_mutex;

size_t withdraw(const std::string& msg, size_t amount) {
  std::lock(atm_mutex, msg_mutex);
  std::lock_guard<std::mutex> message_lock(msg_mutex, std::adopt_lock);

  std::cout << msg << " withdraws " << std::to_string(amount) << std::endl;

  std::lock_guard<std::mutex> account_lock(atm_mutex, std::adopt_lock);

  bank_balance -= amount;
  return bank_balance;
}

int main() {
  std::cout << "Starting balance is " << bank_balance << std::endl;

  std::future<size_t> bonnie = std::async(withdraw, "Bonnie", 100);
  std::future<size_t> clyde  = std::async(deposit, "Clyde", 100);

  std::cout << "Balance after Clyde's deposit is " << clyde.get() << std::endl;
  std::cout << "Balance after Bonnie's withdrawal is " << bonnie.get() << std::endl;

  std::cout << "Final bank balance is " << bank_balance << std::endl;

  return 0;
}
```

Here we just show the withdraw function. Refer to the full source code for complete implementation `bonnie_and_clyde.cpp`.

Note that we have modified the code from lecture so that the deposit and withdrawal functions return a value, rather than just modifying the global state.

Compile and run the `bonnie_and_clyde` program several times. Does it always return the same answer?

On my laptop I ran the example and obtained the following results (which were different every time):

```
[0]$ ./bonnie_and_clyde
Starting balance is 300
Bonnie withdraws 100
Clyde deposits 100
Balance after Clyde's deposit is 300
Balance after Bonnie's withdrawal is 200
Final bank balance is 300

[1]$ ./bonnie_and_clyde
Starting balance is 300
Balance after Clyde's deposit is Bonnie withdraws 100
Clyde deposits 100
300
Balance after Bonnie's withdrawal is 200
Final bank balance is 300

[2]$ ./bonnie_and_clyde
Starting balance is 300
Clyde deposits Balance after Clyde's deposit is 100
Bonnie withdraws 100
400
Balance after Bonnie's withdrawal is 300
Final bank balance is 300

[3]$ ./bonnie_and_clyde
Starting balance is 300
Balance after Clyde's deposit is Clyde deposits 100
400
Balance after Bonnie's withdrawal is Bonnie withdraws 100
300
Final bank balance is 300
```

See if you can trace out how the different tasks are running and being interleaved. Notice that everything is protected from races – we get the correct answer at the end every time. But, note the ordering of calls in the program:

```
  std::future<size_t> bonnie = std::async(withdraw, "Bonnie", 100);
  std::future<size_t> clyde  = std::async(deposit, "Clyde", 100);

  std::cout << "Balance after Clyde's deposit is " << clyde.get() << std::endl;
  std::cout << "Balance after Bonnie's withdrawal is " << bonnie.get() << std::endl;
```

That is, the ordering of the calls is Bonnie makes a withdrawal, Clyde makes a deposit, we get the value back from Clyde's deposit and then get the value back from Bonnie's withdrawal. In run 0 above, that is how things are printed out: Bonnie makes a withdrawal, Clyde makes a deposit, we get the value after Clyde's deposit, which is 300, and we get the value after Bonnie's withdrawal, which is 200. Even though we are querying the future values with Clyde first and Bonnie second, the values that are returned depend on the order of the original task execution, not the order of the queries.

But let's look at run 2. In this program, there are three threads: the main thread (which launches the tasks), the thread for the bonnie task and the thread for the clyde task. The first thing that is printed out after the tasks are launched is from Clyde's deposit. But, before that statement can finish printing in its entirety, the main thread interrupts and starts printing the balance after Clyde's deposit. The depositing function isn't finished yet, but the get() has already been called on its future. That thread cannot continue until the clyde task completes and returns its value. The clyde task finishes its cout statement with "100". But then the bonnie task starts running - and it runs to completion, printing "Bonnie withdraws 100" and

it is done. But, the main thread was still waiting for its value to come back from the clyde task – and it does – so the main thread completes the line "Balance after Clyde's deposit is" and prints "400". Then the next line prints ("Balance after Bonnie's withdrawal") and the program completes.

Try to work out the sequence of interleavings for run 1 and run 3. There is no race condition here – there isn't ever the case that the two tasks are running in their critical sections at the same time. However, they can be interrupted and another thread can run its own code – it is only the critical section code itself that can only have one thread executing it at a time. Also note that the values returned by bonnie and clyde can vary, depending on which order they are executed, and those values are returned by the `future`, regardless of which order the futures are evaluated.

std::future::get Modify your `bonnie_and_clyde` program in the following way. Add a line in the `main` function after the printout from Bonnie's withdrawal to double check the value:

```
std::cout << "Balance after Bonnie's withdrawal is " << bonnie.get() << std::endl;
std::cout << "Double checking Clyde's deposit " << clyde.get() << std::endl;
```

(The second line above is the one you add.) What happens when you compile and run this example? Why do you think the designers of C++ and its standard library made futures behave this way?

2.2 Function Objects

One of the most powerful features of C++ is function overloading. As we have seen, one can create new languages embedded within C++ by appropriately overloading functions and operators. Overloading `operator()` is particular interesting. Although we have overloaded it for matrix and vector types to be used as indexing, `operator()` can be defined to take any kind of arguments. In essence, overloaded `operator()` allows you to pass arbitrary parameters to a function that you define – in essence letting you build your own functions.

Now, of course, when you write a function down when you are programming, you are building functions. But when you build an object that has an overloaded `operator()`, your *program* can generate new functions. This is similar to the concept of lambda described below (and in fact was leveraged to develop the original Boost.Lambda library).

Objects that have an `operator()` defined are called function objects. What makes function objects so powerful is that they can be used just like a function in your programs. We have seen two cases already where we pass functions as parameters: `std::thread` and `std::async`. But what is important in those cases is not that an actual function is being passed to `std::thread` and `std::async`, but rather that the thing being passed is *callable* (i.e., that it can be passed parameters and a function run on those parameters).

Let's look at an example. In the running Bonnie and Clyde bank deposit and withdrawal program, we have had two functions `withdraw` and `deposit` that are invoked asynchronously by the main program. Instead of defining `withdraw` and `deposit` as functions, let's instead define them as function objects. Let's start with a transaction class.

```
int bank_balance = 300;

class transaction {
public:
  transaction(int _sign, int& _balance) :
    sign(_sign), balance(_balance) {}

  void operator()(const std::string& msg, int amount) {
    std::lock_guard<std::mutex> tr_lock(tr_mutex);
    std::cout << msg << ": " << sign*amount;
    std::cout << " -- Balance: " << balance << std::endl;
    balance += sign*amount;
  }
```

```
  void operator()(const std::string& msg, int amount) {
    std::lock_guard<std::mutex> tr_lock(tr_mutex);
    balance += sign*amount;
  }

private:
  int sign;  int &balance;
  static std::mutex tr_mutex;
};
std::mutex transaction::tr_mutex; // construct static member
```

The constructor for this class takes a sign (which should be plus or minus one and keeps a reference to a balance value. It also has its own mutex for safely modifying the balance.

Previously, the `withdraw` function was defined and invoked like this

```
void withdraw(int amount) {
  bank_balance -= amount;
}

int main() {

  withdraw(100);

  return 0;
}
```

We can create a `withdraw` function object that can be used exactly the same way:

```
int main() {
  transaction withdraw(-1, balance);

  withdraw(100);

  return 0;
}
```

The full Bonnie and Clyde example (with threads, say) would then be:

```
int bank_balance = 300;

int main() {
  transaction deposit ( 1, balance);  // define an object named deposit
  transaction withdraw(-1, balance);  // define an object named withdraw

  cout << "Starting balance is " << bank_balance << endl;

  thread bonnie(withdraw, "Bonnie", 100);
  thread clyde(deposit, "Clyde", 100);

  bonnie.join();
  clyde.join();

  cout << "Final bank balance is " << bank_balance << endl;

  return 0;
}
```

(For brevity we omit the messages that were printed by the original functions). This looks very much like the original program where `withdraw` and `deposit` were functions. But remember, in this case, they are objects. And they are created at run time when we construct them. In summary, function objects let you dynamically create and use stateful objects, just as if they were statically defined functions.

Read through the code for this example and run it (and/or modify it).

2.3 For Ninjas: Lambda

Most of us are accustomed to programming in an imperative fashion. Programs are an ordered set of commands that govern what the CPU should process and in what order. Even in a parallel program written in an imperative fashion, each of the concurrent executions are an ordered set of instructions. In contrast, functional programming consists of expressions that are evaluated to produce a final value. In pure functional languages, there is no state per se, there is no notion of assignment. In functional programming, the treatment of functions and data are unified. Expressions can evaluate to be functions, for example. That is, the value of an expression can be function, just as the value of an expression could also be a number or a string or a list. Using Scheme as an example, the following expression evaluates to be a function.

```
(lambda (x) (* x x))
```

So the value of this expression is a function – a lambda expression. A function can be applied – in this case the function takes one parameter and evaluates the product of that parameter with itself (i.e., it squares it). Note that this function has no name – it just *is* a function. The function itself is distinct from the body of the function. The expression `(* x x)` is the value obtained by applying the multiplication operator to x and x.

An unnamed function in Scheme can be used just as a named function:

```
=> ((lambda (x) (* x x)) 7)
49
```

The expression above applies the function `(lambda x)(* x x)` to the value 7 and returns 49.

C++11 (and the Boost.Lambda library before it) provides support for lambdas – for unnamed functions – in C++. Besides the expressiveness in being able generate new functions from other functions, lambdas are convenient when one wants to pass a function to another function, such as when we use `std::async`.

In lecture we have been using matrix-vector product as a running example. The sequential function looks like this:

```
void matvec(const Matrix& A, const Vector& x, Vector& y) {
  double sum = 0.0;
  for (int i = 0; i < A.numRows(); ++i) {
    for (unsigned long j = 0; j < A.numCols(); ++j) {
      sum += A(i, j) * x(j);
    }
  }
}
```

We applied a simple parallelization to this nested loop by performing the inner loop as parallel tasks. However, to realize that we had to create a helper function and put the inner loop computation there.

```
double matvec_helper(const Matrix& A, const Vector& x, int i, double init) {
  for (unsigned long j = 0; j < A.numCols(); ++j) {
    init += A(i, j) * x(j);
  }
  return init;
}

void matvec(const Matrix& A, const Vector& x, Vector& y) {
  std::vector<std::future <double> > futs(A.numRows());
  for (int i = 0; i < A.numRows(); ++i) {
```

```
    futs[i] = std::async(matvec_helper, A, x, i, 0.0);
  }
  for (int i = 0; i < A.numRows(); ++i) {
    y(i) = futs[i].get();
  }
}
```

Now the logic of the original single algorithm is spread across two functions.

With a lambda on the other hand, we don't need to create a separate, named, helper function. We just need to create an anonymous function.

```
void matvec(const Matrix& A, const Vector& x, Vector& y) {
  std::vector<std::future <double> > futs(A.numRows());
  for (int i = 0; i < A.numRows(); ++i) {
    futs[i] = std::async(   [A, x](int row) -> double {
        double sum = 0.0;
        for (unsigned long j = 0; j < A.numCols(); ++j) {
          sum += A(row, j) * x(j);
        }
        return sum;
      }, i);
  }
  for (int i = 0; i < A.numRows(); ++i) {
    y(i) += futs[i].get();
  }
}
```

In C++, lambdas begin with `[]`, which can optionally have "captured" variables passed (that is, variables outside of the scope of the lambda that will be accessed – in this case `A` and `x`). The value that the function is really parameterized on is the row we are performing the inner product with, so we make `row` a parameter to the lambda. The function returns a double, which we indicate with `->`. Finally we have the body of the function, which is just like the helper function body – and just like the inner loop we had before. Finally, we indicate that the argument to be passed to the function when it is invoked is `i`.

There are a variety of on line references to learn more about C++ lambdas and I encourage you to learn more about – and to use – this powerful programming feature.

3 Exercises

3.1 The Return of Norm: Partition Edition

In problem set 2 we wrote some functions for computing the norm of a vector. Now we are going to add 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 partitionedTwoNorm(const Vector& x, size_t partitions);
```

The function takes `x` as an argument, computes its two norm (Euclidean norm) and returns that value. For your implementation, you are to use `std::thread` to execute worker tasks. The number of worker tasks (the number of partitions to make) is given by the argument `partitions`. Each worker task should have this prototype:

```
void ptn_worker(const Vector& x, size_t begin, size_t end, double& partial);
```

The function should *safely* accumulate the sum of squares of the elements in `x` between `begin` and `end` (the half-open interval). The type you use for begin and end is up to you and you can use explicit types rather than the typedef for `size_t`. However it is recommended that you use a 64-bit quantity so that vectors longer than 2^{32} can be tested. You may use synchronization to safely do the accumulation into a single variable or you may accumulate into separate variables that are later summed together. It is important that the workers just compute and accumulate sums of squares and then the sum of all of those sums is computed – and then the square root taken on that final value. The pi example we worked in lecture should be helpful guidance for this exercise. These functions should go into your `Vector.cpp` file.

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 threads (partitions) to create. The vector should be created with the indicated size and then randomized. The function should first time and compute the norm using the sequential function we developed earlier. It should then time and compute the partitioned norm using the specified number of threads. For smaller vectors 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 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 `partitionedTwoNorm` and `ptn_worker` functions
- `Vector.cpp` containing the implementation for the above interface
- The driver `ptn2_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`)
- A `Makefile` that enables `make pt2n_driver` to create an (optimized) 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 The Return of Norm: Recursion Version

Another approach (common in the functional programming community) for decomposing a problem into smaller parts is `recursion`. Recursion is an interesting issue in functional programs because the programs are simply functions. Whereas it isn't too difficult in the imperative world to think of a function calling itself, in the functional world, recursion means that a function is defined in terms of itself. For example, we could define a function of $f(x)$ in the following way

$$f(x) = x \times f(x-1)$$

So what is f? It's defined in terms of itself. And, what do we get when we plug in a value for x, say, 3?

$$f(3) = 3 \times f(2) = 3 \times 2 \times f(1) = 3 \times 2 \times 1 \times (0) = \ldots$$

To make a self-referential definition sensible we have to define a specific value for the function with some specific argument. This is usually called the "base case". In this example, we could define $f(0) = 1$. Then the function is defined as:

$$f(x) = \begin{cases} 1 \text{ if } x = 0 \\ x \times f(x-1) \text{ otherwise} \end{cases}$$

One interesting way to think about recursion is to think of it in terms of abstraction. In the above example, we don't define what $f(x)$ is. Rather we define it as x times $f(x-1)$ – that is, we are abstracting

away the computation – we are pretending that we know the value of $f(x-1)$ when we compute the value of $f(x)$.

This interpretation is also useful for decomposing a problem into smaller pieces that we assume we can compute. For instance, suppose we want to take the sum of squares of the elements of a vector (as a precursor for computing the Euclidean norm, say). Well, we can abstract that away by pretending we can compute the sum of squares of the first half of the vector and of the second half of the vector. The sum of squares of the entire vector is the sum of the values from the two halves. For example:

```
double sum_of_squares(const Vector& x, size_t begin, size_t end) {
  return sum_of_squares(x, begin,                begin+(end-begin)/2)
       + sum_of_squares(x, begin+(end-begin)/2, end) ;
}
```

To use an over-used phrase – see what we did there? We've defined the function as the result of that same function on two smaller pieces. This general approach, particularly when dividing a problem in half to solve it, is also known as "divide and conquer."

Now, in this code example, as with the mathematical example, we need some non-self-referential value. In this case there are several options. We could stop when the difference between `begin` and `end` is sufficiently small. Or, we can stop after we have descended a specified number of levels. In the latter case:

```
double sum_of_squares(const Vector& x, size_t begin, size_t end, size_t level) {
  if (level == 0) {
    return twoNorm(x);
  } else
    return sum_of_squares(x, begin,                begin+(end-begin)/2, level-1)
         + sum_of_squares(x, begin+(end-begin)/2, end                 , level-1) ;
  }
}
```

Now, we can parallelize this divide and conquer approach by launching asynchronous tasks for the recursive calls and getting the values from the futures. Consider for a second what will happen though (particularly if we use `std::launch::deferred`. We won't actually do any computation until the entire tree of tasks is built. The first call will create two futures, when get is called on them, each of them will create two more futures that don't compute, and so on until the base case is reached – at which point the tree collapses back upwards.

Write a function with the following prototype:

```
double recursiveTwoNorm(const Vector& x, size_t levels);
```

The function takes `x` as an argument, computes its two norm (Euclidean norm) and returns that value. For your implementation, you are to use `std::thread` to execute worker tasks. The number of levels to recurse is given by the argument `levels`. Each worker task should have this prototype:

```
double rtn_worker(const Vector& x, size_t begin, size_t end, size_t level);
```

The function should *safely* accumulate the sum of squares of the elements in `x` between `begin` and `end` (the half-open interval) by asynchronously invoking itself with intervals of half the size (as shown above). These functions should go into your `Vector.cpp` file.

As a driver program for this exercise, create a file `rt2n_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 recursive levels to descend. The vector should be created with the indicated size and then randomized. The function should first time and compute the norm using the sequential function we developed earlier. It should then time and compute the recursive norm using the specified number of levels. For smaller vectors 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 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 `recrusiveTwoNorm` and `rtn_worker` functions
- `Vector.cpp` containing the implementation for the above interface
- The driver `rtn2_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`)
- A `Makefile` that enables `make rt2n_driver` to create an (optimized) executable for the driver program

3.3 Matrix Vector Product (AMATH 583 only)

In lecture we discussed some approaches for parallelizing dense matrix-vector product. Create a file `tmv.cpp` that contains a parallelized version (using threads or tasks). The matvec function should have the following prototype:

```
void task_matvec(const Matrix& A, const Vector& x, Vector& y, size_t partitions);
```

As with previous matrix-vector product implementations, the vector y should contain the product of A and x when the function returns. The function should divide the overall problem into the given number of partitions.

Your helper function should be prototyped thusly:

```
double matvec_helper(const Matrix& A, const Vector& x, Vector& y, size_t begin, size_t end);
```

As a driver program for this exercise, create a file `tmv_driver.cpp`. That file should contain a `main` function that reads two arguments from the command line. The first is the size of the problem. We assume the matrix is square and the vectors are the same length (as specified by this argument). The second is the number of threads (partitions) to create. The matrix and vectors should be constructed with the given sizes and then randomized. 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 partitioned version using the specified number of threads. 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 size of the problem, the second the sequential execution time, the third the parallel execution time, the fourth the speedup, and finally the difference between the two norms of y computed sequential and y computed in parallel.

Deliverables

- `Vector.hpp` containing the interface for the `Vector` type, including the `task_matvec` function
- `Vector.cpp` containing the implementation for the above interface
- `Matrix.hpp` containing the interface for the `Matrix` type
- `Matrix.cpp` containing the implementation for the above interface
- The driver `tmv_driver.cpp` as described above
- A `Makefile` that enables `make Matrix.o` to create an object file for the `Matrix` type (i.e. `Matrix.cpp`)
- A `Makefile` that enables `make Vector.o` to create an object file for the `Vector` type (i.e. `Vector.cpp`)
- A `Makefile` that enables `make tmv_driver` to create an (optimized) executable for the driver program

3.4 Written Questions

1. You measured bandwidth explicitly in the bandwidth program and implicitly with the roofline program. How well do the bandwidths from these two profiles match? Include the two generated pdf files (do not rename them) with your other materials for this assignment.
2. Calculate the numeric intensity of sparse matrix-vector product with compressed sparse row (CSR) format, for the two cases of index types being integer (32 bit) and unsigned long (64 bit). Use your roofline graph to estimate the performance of CSR sparse matrix vector product on your computer.
3. **Extra Credit** Implement sparse matrix-vector product with CSR format and compare the results you obtain experimentally with what was predicted by your roofline model.
4. **(AMATH 583 only)** In lecture 12 the instructor speculated about the reason for only limited speedup in the dense matrix-vector product example and hypothesized an explanation. What kind of experiment might you conduct to prove (or disprove) that hypothesis?

Place the above prompts/questions and your responses into a text file `ex5.txt`.

4 Turning in The Exercises

All your cpp and hpp files, your `Makefile`, your pdf files, and your txt file should go in the tarball `ps5.tgz`. If necessary, include a text file ref5.txt that includes a list of references (electronic, written, human) if any were used for this assignment.

Before you upload the `ps5.tgz` file, it is **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_ps5.py by using the command `python test_ps5.py` in your ps4 directory. Once you are convinced your code is exhibiting the correct behavior, upload ps5.tgz to Collect It.

5 Learning Outcomes

At the conclusion of week 5 students will be able to

1. Characterize the differences between a process and a thread
2. Use `std::thread` to spawn a thread
3. Describe the arguments passed to `std::thread`
4. Describe what `join()` does
5. Write a simple program that launches four concurrent threads and that demonstrate concurrency in some fashion
6. Explain the difference between concurrency and parallelism
7. Define data race and describe a scenario where a data race would occur
8. Write a simple program that exhibits a data race
9. Name two hardware supported atomic operations and show how to use them to protect a critical region
10. Describe a spin lock and its advantages and disadvantages
11. Identify potential data races in a concurrent program
12. Correctly use `std::mutex()` to protect critical regions

13. Explain the difference between `std::mutex()` and a spin lock
14. Define RAII and why it is used
15. Explain `std::lock_guard`
16. Correctly use `std::lock_guard` to protect a critical section
17. Explain what deadlock is
18. Describe a scenario where deadlock might occur
19. Use `std::thread_guard` and `std::lock` to correctly protect a critical section having multiple mutexes – in a deadlock-free fashion

At the conclusion of week 6 students will be able to

1. Describe the four steps of creating a parallel program (as presented by Mattson et al).
2. Describe the difference between `std::thread` and `std::async`
3. Describe the two different modes for launching `std::async` and what the difference is between them
4. Write a simple program that launches four concurrent tasks with `std::async` and that demonstrate concurrency in some fashion
5. Describe what `std::future` is
6. Define atomicity as it relates to concurrent programs
7. Use `std::atomic` to protect a shared variable
8. Characterize the types that can be used with `std::atomic`
9. Describe a parallelization approach for parallelizing inner product, dense matrix-vector product, sparse matrix-vector product, and dense matrix-matrix product
10. Describe the roofline performance model and an experimental approach that you could use to obtain its parameters
11. Use the roofline performance model to estimate the performance of dense matrix-vector product and sparse matrix-vector product with COO and CSR matrix formats