This lab is meant to acquaint you with:

1. A simple C++ program that issues instructions intended for asynchronous execution
2. How to collect (and sum in this case) the “output” of multiple threads

In lecture we’ve discussed and in lab 2 you learned about the thread functionality available to you in C++. It is a simple way to designate that some chunk of code should be doled out to multiple processors and/or CPUs (diagram shown on right). Indeed that’s what you’ve been assigned to do (and to measure the performance of such a program) for the programming portion of homework 2.

However, the use of the thread functionality does not permit to (easily) collect the output(s) of a single or multiple threads. For example, assume that you have a square matrix, and that you want to sum the entries of the matrix – that’s easily done. But if the matrix is excessively large, and you have multiple cores and/or CPUs at your disposal, why not dole out to your set of cores and/or CPUs different “portions” of your matrix, and then sum each core’s summation. The diagram on the right for the entries for Matrix A specifies four colored regions, each of which can be sent to a separate core/CPU for summation.

In this lab, you’ll create a 3D array with a high-enough dimension (1000, for example, thus with 1 billion entries), and record the time that it takes to sum the entries if asynchronous threading is not used, and when you chunk up the 3D array into different regions and send each region to an asynchronous thread. The calculations (the summing) performed by each asynchronous thread will be collected and summed.

This lab is more-or-less open-ended. I provide you with all of the needed ingredients, and it is up to you assemble them in such a way so as to achieve the task that you are asked to do.

I. Create a lab3 directory

In your account on your N drive, create a lab3 directory.
II. Functions in C++ that return values, and 3D arrays of pointers.

Create a new C++ file, matrixSum3DAsync.cpp, in your lab3 directory. Before you start writing code, a few hints.

Although you can specify a 3D array in C++ by writing the following:

```cpp
int myArray[2][2][2] = {{ {1,2},{3,4}},{5,6},{7,8}};
```

this works only for “small” arrays. If your array is large (in our case), unless you are skilled at manual memory management, it is better to create a 3D array made up of pointers. That way, the OS will manage memory allocation for you. Here’s how you create a 3D array, called my3DArray, made up of 1 billion doubles (each having a value of 3.6):

```cpp
// array dimension
int dim = 1000;

// create a 3d array up of pointers pointer pointers to doubles
double ***my3DArray = new double**[dim];
for (int i=0; i<dim; i++){
    my3DArray[i] = new double*[dim];
    for (int j=0; j<dim; j++){
        my3DArray[i][j] = new double[dim];
        for (int k=0; k<dim; k++){
            my3DArray[i][j][k] = 3.6;
        }
    }
}
```

In our case, you want to write a function that returns the sum of a portion of the 3D array. The general structure of such a function is being provided to you in the following figure:

```cpp
// A function, myFunction, that receives two arguments,
// one of which is a pointer to a 3D array, the other
// an int, and which returns a long double
long double myFunction(double*** a3DArray, int anInt){
    long double aValue;
    // calculations magic done here
    return aValue;
}
```

**Note:** In this lab, you will write your “own” myFunction function. How many arguments you declare that function to have is up to you. Therefore the header of your myFunction might look far different than the one that is shown here for demonstration purposes.
III. Asynchronous thread invocations in C++

As was already mentioned in the introduction of this lab, the use of explicit threads in C++ does not provide functionality to easily retrieve the values of computations performed by threads once they have completed. Instead, the future library/include in C++ provides async for issuing threads asynchronously, and then retrieving their “outputs.” You need to include the correct library calls at the top of your cpp file:

```cpp
#include <iostream>
#include <future>
#include <sys/time.h>
#include <stdio.h>
```

The iostream, sys/time.h and stdio.h are all functionality that you’ve seen in previous labs.

To issue an asynchronous thread from which you can then collect a value, you’d do the following:

```cpp
auto thread = std::async(myFunction, my3DArray, 3432);
long double myOutput = thread.get();
```

The above example issues to a single thread the function myFunction with the arguments my3DArray (both shown above) with the second int argument as 3432. The get() method retrieves the myFunction’s return value as soon as it is available.

IV. Asynchronous launch

Just because you create a thread, does not mean that the scheduler will run that thread as soon as that portion of your code where the thread is declare is executed. A scheduler on an n processor computer might indeed even decide to run in series n threads on a single of the n processors if the load on all available processors is high.

There is a way for you to force the scheduler to run a thread as soon as you create it. Refer to the future library (http://www.cplusplus.com/reference/future/) for more details.

Hint: You’ll want to use something in addition to async to force your threads to run as soon as they are created. The cplusplus.com link provided above provides sample code which might be very helpful for completing this lab.
V. The pseudocode

With what you’ve learned in previous labs and in this lab’s writeup, you now have all of the ingredients necessary to:

A. Create a cube 3D array of a non-trivial size (1 billion entries)
B. Issue asynchronous threads to sum the 3D array’s entries
C. Collect values from the asynchronous threads
D. Calculate wall clock time (see lab 1)

The pseudocode for your program is being provided for you on the last page of this lab. Because a large chunk of time is needed to create the 3D array, you cannot rely on the `time` command-line functionality to assess time needed to perform just the summing portion of this program. Thus, use the wall clock function provided to you in the first lab.

You’ll need to determine HOW to break up the 3D cube array into different portions. Should you send ½ of the array to each of two threads? Or, should you send 1/3 of the array to each of three threads? Etc.

VI. Compiling and running

To compile the file `matrixSum3DAsync.cpp`, issue the following (refer to lab 2 for a refresher):

```
g++ matrixSum3DAsync.cpp -o matrixSum3DAsync -std=c++0x -pthread
```

To run the just-created executable `matrixSum3DAsync`, issue the following command:

```
./matrixSum3DAsync
```

The output should be something like the following:

```
Init time : 7.31048
SumThread time : 2.02844
The sum of the matrix's entries, threaded calculation : 3.6e+09
SumNonThread time : 3.93903
The sum of the matrix's entries, non-threaded calculation : 3.6e+09
```

The output above outputs the summation of the matrix when asynchronous threading is used and when asynchronous threading is not used as a check that the calculations are being done correctly.

**Task**: Experiment with different dimension values of the 3D array. Your goal is to use a “large” enough 3D array so that the asynchronous threaded version of the summation takes less time than the synchronous version. Note that depending on the architecture of your computer, a speedup using asynchronous execution may be difficult to achieve (we’ll discuss this in lecture).
Compose a document in which you:

1. Provide the dimension for your 3D array
2. Specify how many asynchronous threads you issued and explain WHY
3. Provide a screen shot of the output of your program (as for example above)
4. Discuss/explain why having two threads does not necessarily reduce the execution time (summation) by half. If the speedup of using threads is not as expected, speculate why that may be the case.

VII. Submission

Upload to Canvas the following:

- A document (.doc, .docx, .pdf) with your answers and brief discussion for the 4 items above.
- The program matrixSum3DAsync.cpp

VIII. Rubric

<table>
<thead>
<tr>
<th>Component of Lab</th>
<th>Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>Brief write-up</td>
<td>20 points</td>
</tr>
<tr>
<td>matrixSum3DAsync.cpp completed</td>
<td>10 points</td>
</tr>
<tr>
<td>matrixSum3DAsync.cpp uses future and async</td>
<td>20 points</td>
</tr>
<tr>
<td>matrixSum3DAsync.cpp compiles without errors</td>
<td>20 points</td>
</tr>
<tr>
<td>Asynchronous calculation is faster than synchronous calculation in matrixSum3DAsync.cpp</td>
<td>10 points</td>
</tr>
<tr>
<td>Program runs to completion</td>
<td>20 points</td>
</tr>
<tr>
<td>Total</td>
<td>100 points</td>
</tr>
</tbody>
</table>
IX. Pseudocode

```
#include <iostream>
#include <future>
#include <sys/time.h>
#include <stdio.h>

using namespace std;

// retrieve clock time (from lab 1)
double get_walltime(){
    // refer to lab 3 for this code
}

// a function, myFunction, that takes as input a pointer to a 3D array, 
// and three other integers arguments, and which returns a long double
long double myFunction(double*** array3D, int dimLower, 
                        int dimUpper, int dim){

    // Iterate over the portion of the 3D cube array that this function 
    // receives, and sum the entries. Return the sum.

}

// main program
int main(){

    // part I -- creating array
    // 1. get the clock time, t1
    // 2. declare the cube 3D array's dimension(s)
    // 3. create the cube 3D array
    // 4. get the clock time, t2
    // 5. calculate time needed to initialize the array using t1 and t2

    // part II -- asynchronous summing
    // 1. get the clock time, t3
    // 2. create asynchronous threads
    // 3. retrieve "sum" from each async thread and sum them
    // 4. get clock time, t4
    // 5. calculate time needed to sum the entries for async using t3 and t4
    // 6. output the sum of the array's entries

    // part III - synchronous summing
    // 1. get the clock time, t5
    // 2. sum the entries of the cube 3D array using a "standard" for loop.
    // 3. get the clock time, t6
    // 4. calculate time needed to sum the entries for sync using t3 and t4
    // 5. output the sum of the array's entries

}
```