

Marshall Lanning

CECS 625 Parallel Programming Homework Assignment #3
September 23, 2019 (100 points)

Due: October 2 (Wed) midnight
(Submit your project report and the required VS 2015 project to the Blackboard.)

Assignment Description

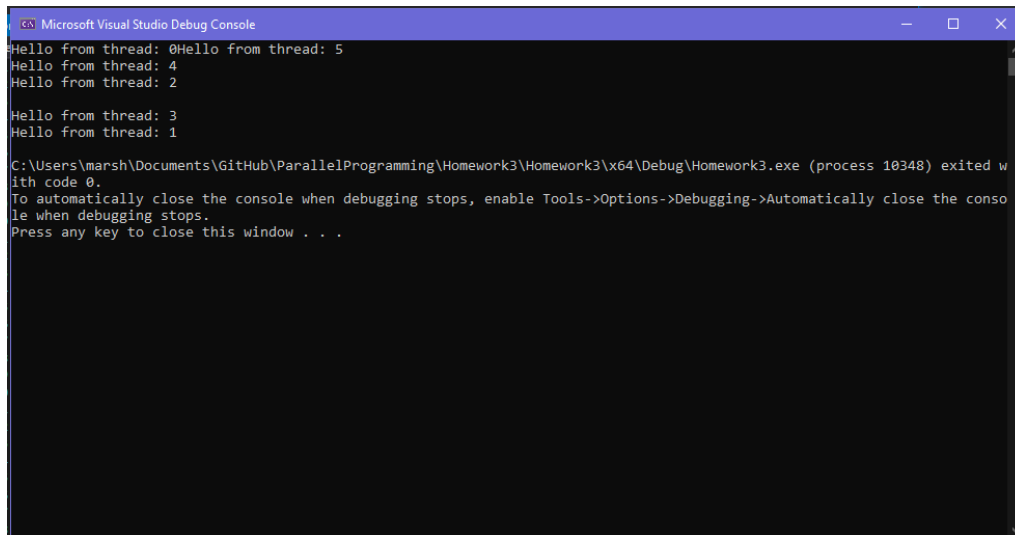
For this assignment, you need to study sections 4.1 to 4.5 (pp. 77-121) of the textbook.

Create a VS 2015 C++ console project and copy the folders data and include (which can be downloaded from the Blackboard to the project home folder as demonstrated in class. Use this project to do the following problems and all the required cpp files mentioned in the problems are available from the textbook's source code website:

<https://github.com/JGU-HPC/parallelprogrammingbook/tree/master/>.

1(10 points) Section 4.1

Run the first multithreaded program, `hello_world.cpp`, using 6 threads. In the project report, show the screenshot of the output and explain the output



```
Microsoft Visual Studio Debug Console
Hello from thread: 0Hello from thread: 5
Hello from thread: 4
Hello from thread: 2


Hello from thread: 3
Hello from thread: 1

C:\Users\marsh\Documents\GitHub\ParallelProgramming\Homework3\Homework3\x64\Debug\Homework3.exe (process 10348) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

The output shows the method `say_hello` being ran on 6 simultaneous threads. The output looks jumbled up and out of order due to all the threads competing to access the output buffer of the compiler.

2 (20 points) Section 4.2

Test run `traditional.cpp` and `promise_future.cpp`. In the report, explain the differences of these two approaches of handling return value.



Microsoft Visual Studio Debug Console

```

-----
Traditional Output
0
1
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
10946
17711
28657
46368
75025
121393
196418
317811
514229
832040
1346269
-----

```

```

-----
Promise_Future Output
0
1
1
2
3
5
8
13
21
34
55
89
144
233
377
610
987
1597
2584
4181
6765
10946
17711
28657
46368
75025
121393
196418
317811
514229
832040
1346269
-----

```

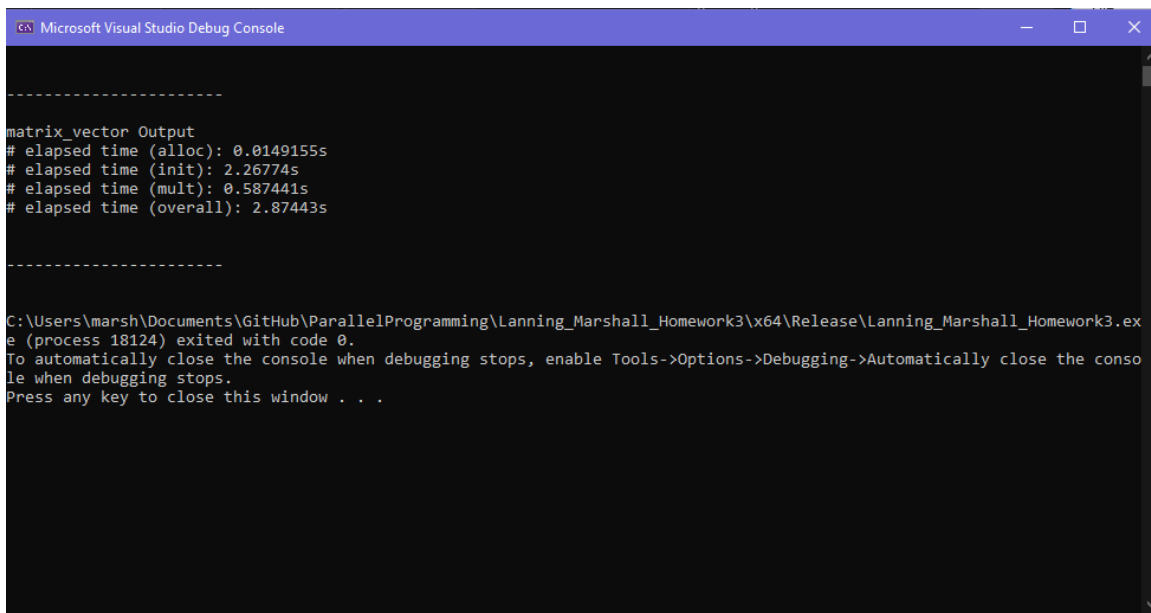
The difference between these two approaches is that the traditional way reserves the return value of a function for the error code making other compound quantities passed via pointers in the argument list which are subsequently manipulated inside the functions body. The Promise and Future method is designed for the return value to be passed asynchronously where the programmer may define so-called promises that are fulfilled in the future. This establishes a casual dependency between the promise p and the future f that can be used as a synchronization mechanism between a spawned thread and calling the master thread

3 (35 points) Section 4.3

- (a) Test run `matrix_vector.cpp` and compare sequential time and parallel (using `cyclic_parallel_mult` function) time. In the project report, show the timing results and explain why using lambda expressions in the implementation.

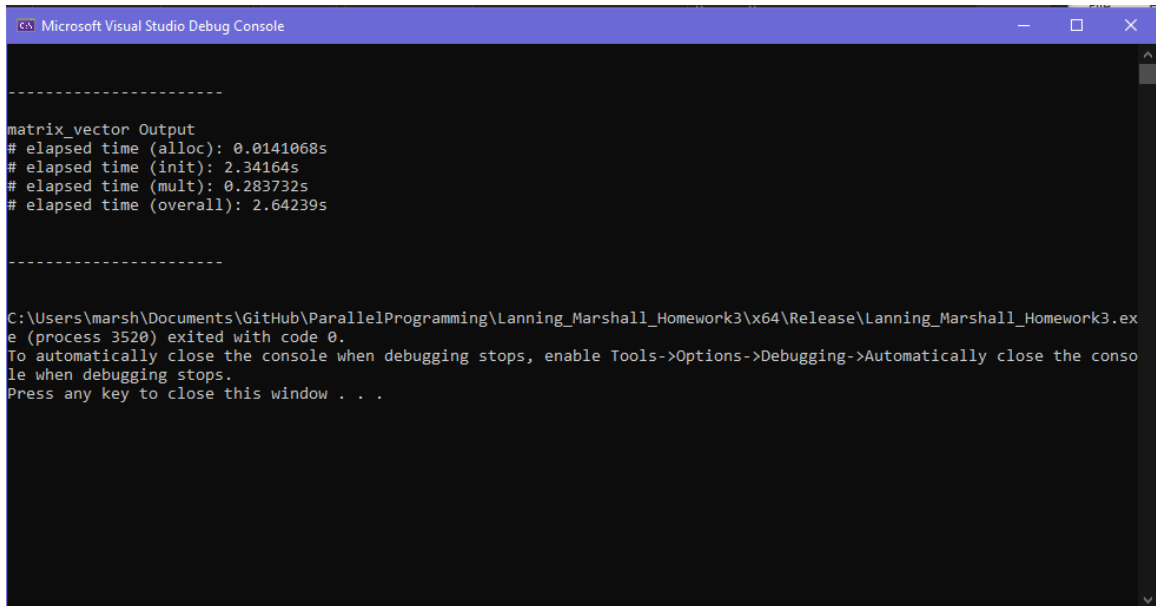
The use of lambda expression helps to pass a a large amount of references to threads in an elegant way.

Sequential



```
-----  
matrix_vector Output  
# elapsed time (alloc): 0.0149155s  
# elapsed time (init): 2.26774s  
# elapsed time (mult): 0.587441s  
# elapsed time (overall): 2.87443s  
-----  
  
C:\Users\marsh\Documents\GitHub\ParallelProgramming\Lanning_Marshall_Homework3\x64\Release\Lanning_Marshall_Homework3.exe (process 18124) exited with code 0.  
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.  
Press any key to close this window . . .
```

Parallel



```
-----  
matrix_vector Output  
# elapsed time (alloc): 0.0141068s  
# elapsed time (init): 2.34164s  
# elapsed time (mult): 0.283732s  
# elapsed time (overall): 2.64239s  
-----  
  
C:\Users\marsh\Documents\GitHub\ParallelProgramming\Lanning_Marshall_Homework3\x64\Release\Lanning_Marshall_Homework3.exe (process 3520) exited with code 0.  
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.  
Press any key to close this window . . .
```

- (b) In the parallel implementation, `cyclic_parallel_mult`, on pages 100-101 of the textbook), replace Lines 13-22 by the following code

```
auto cyclic = [&] (const index_t& id) -> void  
{  
    for (index_t row = id; row < n; row += num_threads)  
    {  
        // initialize result vector to zero  
        b[row] = 0;  
        // directly accumulate in b[row]  
        for (index_t col = 0; col < n; col++)  
            b[row] += A[row*n+col]*x[col];  
    }  
}
```

Run the modified version and measure the running time and in the report, show the result and explain why the time performance is worse than the original one.

```
Microsoft Visual Studio Debug Console

-----
matrix_vector Output
# elapsed time (alloc): 0.0154285s
# elapsed time (init): 2.35941s
# elapsed time (mult): 0.561773s
# elapsed time (overall): 2.9396s
-----

C:\Users\marsh\Documents\GitHub\ParallelProgramming\Lanning_Marshall_Homework3\x64\Release\Lanning_Marshall_Homework3.exe (process 11316) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

The new way has us accumulate the contributions of the scalar product directly in the result vector b and do not declare a dedicated register $accum$ in order to cache the intermediate results. This ends up causing an excessive amount of invalidation of shared cache lines and is called false sharing. We should avoid excessive updates of entries stored in the same cache line when using more than one thread in parallel and try to cache intermediate results in registers in order to reduce the update frequency to cached entities.

4 (35 points) Section 4.4

- (a) Run `all_pair.cpp` to collect timing results of `sequential_all_pairs` and `dynamic_all_pairs` and in the report, show the timing results.

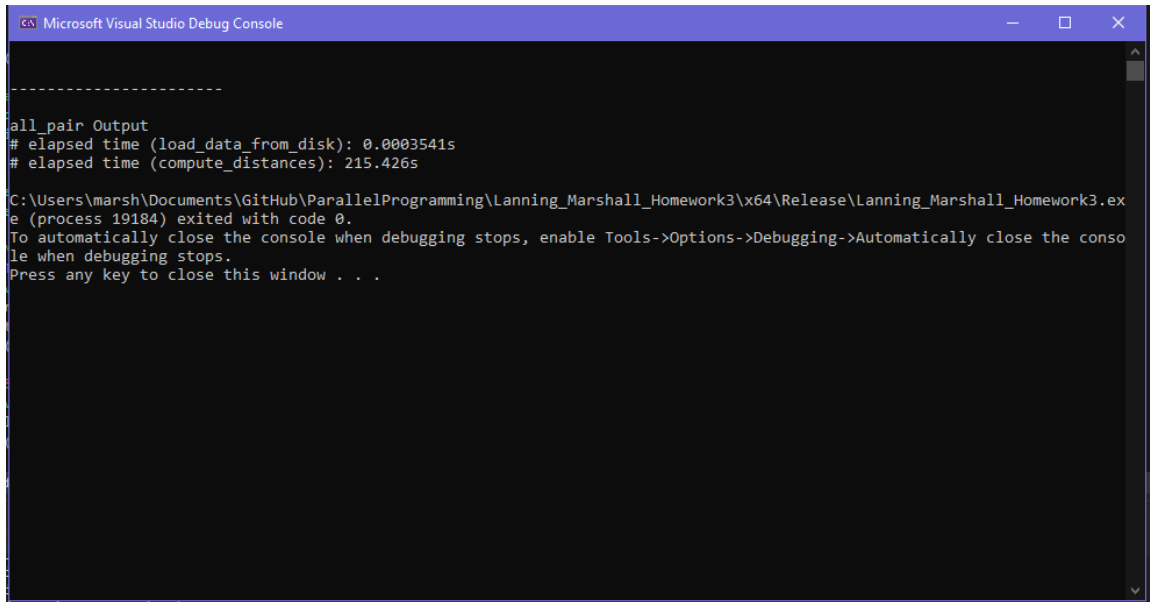
Sequential

```
Microsoft Visual Studio Debug Console

-----
all_pair Output
# elapsed time (load_data_from_disk): 0.187496s
# elapsed time (compute_distances): 2335.52s
-----

C:\Users\marsh\Documents\GitHub\ParallelProgramming\Lanning_Marshall_Homework3\x64\Release\Lanning_Marshall_Homework3.exe (process 5272) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

Dynamic



```
-----  
all_pair Output  
# elapsed time (load_data_from_disk): 0.0003541s  
# elapsed time (compute_distances): 215.426s  
C:\Users\marsh\Documents\GitHub\ParallelProgramming\Lanning_Marshall_Homework3\x64\Release\Lanning_Marshall_Homework3.exe (process 19184) exited with code 0.  
To automatically close the console when debugging stops, enable Tools->Options->Debugging->Automatically close the console when debugging stops.  
Press any key to close this window . . .
```

- (b) In the report, explain why dynamic scheduling is better than static scheduling for the all-pair distances problem using the formula (4.6) given on page 110 of the textbook.

Static schedules with a small chunk size are useful to approximately balance skewed work distributions. Dynamic scheduling works the same way but is better because we eliminate excess runtime at program start by dynamically building the skewed work distributions. Smaller chunk sizes are favorable due as shown in the formula:

$$(\text{Alpha}(c + 2i(0)*c + c^2))/2$$

The static distribution sets the chunk size before runtime while the dynamic distribution sets the chunk size when threads run out of work.

- (c) In the report, explain the key ideas of the `dynamic_all_pairs` function implementation and how and why the `std::mutex` object is used.

The `dynamic_all_pairs` function dynamically sets chunk sizes for work distributions by utilizing a globally accessible variable `global_lower` which denotes the first row of the currently processed chunk. Whenever a thread runs out of work, it reads the value of that variable, subsequently increments it by the chunk size `c`, and finally processes the corresponding rows of that chunk. All threads terminate if `global_lower` is greater than or equal to the number of to be processed

rows m. In order to be sure that access to `global_lower` are mutually exclusive to guarantee correct results we utilize the C++ mechanism called mutex. A mutex can be locked by a specific thread causing implicit synchronization of threads. Basically, this mechanism serializes certain portions of code in a parallel context to ensure the safe manipulation of shared information.