Programming for performance - Assignment 4

Abhay Kumar Dwivedi
22111001

October 28, 2022

System description :

Model name : Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz
Architecture : x86_64
Processor Frequency : 3.20 GHz
Core : 6
L1d cache size (Data cache) : 192 KiB
L1i cache size (Instruction cache) : 192 KiB
L2 cache size : 1.5 MB
L3 cache size : 12 MB

g++ version : 9.4.0
ubuntu version : 20.04.1

Solution - 1 (Fibonacci)

For N = 40	Serial Version	Fib Version 1	Fib Version 2	Fib Blocking	Fib Continuation
Time taken(1)	515050 micro seconds	417054 micro seconds	8 micro seconds	89746 micro seconds	81254 micro seconds
Time taken(2)	516347 micro seconds	447056 micro seconds	6 micro seconds	89256 micro seconds	79835 micro seconds
Time taken(3)	518165 micro seconds	397251 micro seconds	2 micro seconds	89455 micro seconds	81683 micro seconds
Time taken(4)	513266 micro seconds	401271 micro seconds	5 micro seconds	89213 micro seconds	81131 micro seconds

Compilation command : g++ -std=c++11 -fopenmp 22111001-prob1.cpp -o fibonacci -ltbb

Execution command : ./fibonacci

Avg. Speedup (Fib Version 1) : 1.23
Avg. Speedup (Fib Version 2) : 128043.68
Avg. Speedup (Fib Blocking) : 5.76
Avg. Speedup (Fib Continuation) : 6.36

Fib Version 1 : In this version I have tried to parallelize the recursion call, though In fibonacci we can't parallelize more than 2 calls as there are dependencies in it. Also it is very slow for small numbers so I have used an offset value of 18 which means if a number is less than or equal to 18 then we will call the serial version, if not then we will call the parallel version.

I have used taskwait because the ans at any point depends upon last two values. master clause is used to make only single task. The average speedup I got in parallel version compared to serial version is **1.23**.

Fib Version 2 : In this version I have tried to parallelize the optimized version of fibonacci. Fibonacci have been optimized by using space optimized version of dp. In this version unlike the plain dp I have not used an array, only three variables are used as at any point of time the current number depends upon only the previous two numbers.

TaskWait is used so that dependency is maintained and any number is processed only when the past two numbers are already processed. The average speedup I got in parallel version compared to serial version is **128043.68**.

Fib Blocking : In this version I have used task based programming with blocking style parallelism. I have made a class FibBlocking which inherits data and functions from class task.

There are two main methods in the class, one is constructor method and another is overridden execute method. The constructor initializes the data members of the class and the execute method contains the logic of Fibonacci.

In the execute method, first there is a cutoff so as to compute Fibonacci serially if the number is small. If the number is large enough then we have made two child task using allocate_task() method which is inherited from the task class.

Then we set reference_count to 3, 2 for child tasks and 1 for method spawn_and_wait_for_all(). Then we spawned the second task and the parent does not wait for it to get complete till this point. Then we spawn first task using method spawn_and_wait_for_all() which forces parent to wait till both its child task gets completed.

After the child tasks get completed, *sum is updated by the addition of result of both child tasks. To run all this code first we made an object of this class using allocate_root() method which makes space for root task and spawned that task using spawn_root_and_wait() method which

waits till the root task is over. This method calls the execute function.The average speedup I got in parallel version compared to serial version is **5.76**.

Fib Continuation : This version is similar to the blocking version except the parent does not wait for the child tasks to get completed. In this version there are total of two classes - FibCHelperClass and FibContinuation. Both the class extends task class. FibCHelperClass is basically a helper class where simple we are overriding execute method of task to add x and y and store it to sum.

In FibContinuation class we have two functions, constructor and execute. Execute function is basically overriding the existing execute function in the task class and it has the overall logic in it of creating tasks. Firstly it checks whether the number is less than cutoff or not. If yes then find fibonacci serially, if no then use task based programming.

Firstly we create an object of our helperclass using allocate_continuation() method, then we create two objects for two child tasks using allocate_child() method, this time we did not use local variable, instead data members of our helper class is used as the parent may return before child completes. The reference count has been set to 2 (for both task).

Then task b is spawned and using scheduler bypass which helps to directly specify the next task we are going to run, we did not spawn the task a, instead we returned it and it will be executed by the thread. To run these code an object of FibContinuation class has been made using method allocate_root() and spawned using function spawn_root_and_wait() then we return sum. The average speedup I got in parallel version compared to serial version is **6.36**.

Solution - 2 (Quick Sort)

For N = 20	Serial Version	Parallel version	SpeedUp
Time taken(1)	2209090 micro sec	1420257 micro sec	1.55
Time taken(2)	2218234 micro sec	1411143 micro sec	1.57
Time taken(3)	2223248 micro sec	1416197 micro sec	1.56
Time taken(4)	2215033 micro sec	1407599 micro sec	1.57

Compilation command : g++ -std=c++11 -fopenmp 22111001-prob2.cpp -o quicksort

Execution command : ./quicksort

Avg. Speedup : 1.56

Parallel version : In the parallel version of quick sort function I have used tasks in Open MP

to parallelize the two recursive calls. **Also these calls are independent as they work on different portion of array we do not need to use taskwait here.**

Task based programming has an overhead of creating and destroying tasks, that's why for small numbers it is slow, as the overhead is more than the computation required for those tasks. To tackle this issue I have used an offset of 2^{10} which means if the array elements are less than or equal to 2^{10}, then we will use serial version, else we will use parallel version and create task.

Also, #pragma omp single is used so that to avoid duplicates tasks creation which will affect the result. The average speedup I got in parallel version compared to serial version is **1.56**.

Solution - 3 (Find Pi)

NUM_INTERVALS = 1000000000	Serial Version	Open MP Version	SpeedUp (OMP)	TBB Version	SpeedUp (TBB)
Time taken(1)	2653052 micro sec	545827 micro sec	4.86	664228 micro sec	3.99
Time taken(2)	2689528 micro sec	538961 micro sec	4.99	664424 micro sec	4.04
Time taken(3)	2690844 micro sec	539780 micro sec	4.98	664879 micro sec	4.04
Time taken(4)	2656552 micro sec	555131 micro sec	4.78	668429 micro sec	3.97

Compilation command : g++ -std=c++11 -fopenmp 22111001-prob3.cpp -o pi -ltbb

Execution command : ./pi

Avg. Speedup(Open MP) : 4.90
Avg. Speedup(TBB) : 4.01

Open MP Version : In Open MP version of the code, we have run a for loop in parallel, where each thread is going to run iterations of index multiple of their identity number, that is why the index variable has been initialised with thread_id and is increased by num_of_threads .

There is a variable sum define inside the parallel block which means it is private to all the thread. In the for loop body, each thread update their private sum variable, hence there is no need to use synchronization constructs inside the loop body.

After calculating their local sum they go on to update the shared variable p, which is why there I have used atomic synchronization construct so that only one thread will be able to add to sum

at any point of time. The average speedup which I got in this version compared to serial version is **4.90**.

TBB Version : In TBB version of the code I have used functor to calculate value of PI efficiently. There is a class ComputePI which contains two data variables sum and step which are public so that we can access them directly by the object.

There are four functions in the class. First one is ComputePI(double s) which is a constructor that initializes sum by s. Second one is ComputePI(obj, split) which is also a constructor that initializes sum when there is splitting of work between threads. Third method is join which basically joins the result of the thread. And the last one is overloaded operator()(range) method which basically performs the computation of PI parallely by dividing iteration among the available threads.

In operator method every thread has their local sum in localSum variable and after calculating localSum they add it into the sum variable.

Then we made an object of this class by giving parameter 0.0 (initializing sum by 0.0). After that I called tbb::parallel_reduce() function to basically perform reduction operation parallely. At last we multiplied the sum by 4 and the number of steps and returned it. The average speedup which I got in this version compared to serial version is **4.01**.

Solution - 4 (Find Max)

For N = 2^{26}	Serial Version	Parallel Version	SpeedUp
Time taken(1)	129366 micro sec	34949 micro sec	3.70
Time taken(2)	129831 micro sec	34789 micro sec	3.73
Time taken(3)	129767 micro sec	35129 micro sec	3.69
Time taken(4)	127714 micro sec	35045 micro sec	3.64

Compilation command : g++ -std=c++11 22111001-prob4.cpp -o find-max -ltbb

Execution command : ./find-max

Avg. Speedup : 3.69 times

Parallel Version : In TBB version of the code I have used reduction to find index of maximum number present in the array. I have used Functor in this program to perform reduction. In class FindMax we have three variables, arr which points to our array, maxEl which gives us the maximum element and index which gives us index of the maximum element.

There are four functions in the class as well. First one is the constructor to initialize data variables. Second one is also constructor which initializes the data variable when work splitting happens among threads. Third one is join function which basically joins the result of threads that is compare the max element of each thread and update data variables accordingly.

Fourth function is operator()() which we have overloaded so as to run this method when this class object is used with tbb parallel algorithms. This method has the logic of computing max element and its index.

We have made object of this class which initializes arr pointer by address of our array. After that we have called parallel_reduce function to perform reduction operation and then returned obj.index to return the index of maximum element. The average speedup which I got in this version compared to serial version is **3.69**.