Homework 4

Hai Zhu

due date: October 30, 2017

Programming Assignment

Compute $\max\{g(x) : a \leq x \leq b\}$ for a given real-valued function g and endpoints $a < b$. An $\epsilon > 0$ tolerance will be given. The problem is further simplified by the fact that g is differentiable and s on the absolute value of the derivative will be given.

Procedure: Using OpenMP, run your program on the PSC Bridges computer for $p = 7, 14, 28,$ and 56:

1. Start with interval $[a, b]$, and set $M = \max\{g(a), g(b)\}$.
2. Examine intervals by determine if they could contain a value larger than one that you already have found.
3. If not then discard the interval, and otherwise look at it more closely.
4. Divide interval until no further interval might have a large value.
5. Print out the elapsed time and the max function value within tolerance.

Write your program to work with arbitrary values of $a < b$, $\epsilon > 0$, $s > 0$, and arbitrary function g. All calculations are in double precision. You are not allowed to use the OpenMP TASK construct. Write a short report reporting the max value, and briefly explain how you parallelized the problem, and analyze the timings obtained.

Search and Examine Interval:

- With a little algebra one can show that in the interval $[c, d]$, g can be at most $(g(c) + g(d) + s(d - c)/2)/2$.
- There are two main search algorithms for such a problem: depth-first and breadth-first.

C++ OpenMP Implementation

1. **Examine and Parallel Scheme**

   In this problem, we want to compute the maximum of a differentiable function with given bound on the absolute value of the derivative. And we can only generate candidate intervals by examining the current available intervals and divide it into halves. The search algorithm I choose to use is breadth-first search(BFS).

   For one reason, the function provided looks like a periodic function with a slightly increasing local maximum, that tells us if we use depth-first search from the left end point, we might end up doing too much work and find out it's in fact not the maximum within tolerance. DFS might helps us getting rid of a few intervals when you don't require ϵ to be so small, and the function itself is not this well made up. And if the function has a different behavior, the performance if going to change unless we took a reasonable guess.

The second reason is because for BFS, you can see where the parallel happens intuitively at each level. Once we are on an fine grid, we need to loop over all current smaller intervals to search for possible maximum. And this for loop can be parallelized by using OpenMP.

$$
\begin{array}{l}
[0,1] \\
[0,\frac{1}{2}],\ [\frac{1}{2},1] \\
[0,\frac{1}{4}],\ [\frac{1}{4},\frac{1}{2}],\ [\frac{1}{2},1] \\
[0,\frac{1}{4}],\ [\frac{1}{4},\frac{1}{2}],\ [\frac{1}{2},\frac{3}{4}],\ [\frac{3}{4},1]
\end{array}
$$

And here is a short description of my algorithm. The procedure is very similar to the levels of intervals shown in the above picture. And the idea is to parallelize each level instead of a while loop. First, I start from one interval. And based on the dividing criteria, I divide the interval in two, and update the current maximum value M. From here, we continually do a while loop as long as number of intervals need to be checked aren't 0. And when we loop over all current intervals in level n (at most 2^n intervals), we use OpenMP to parallelize this for loop. It may seems to be slowing things down, because each time we call `progma omp parallel`, there is a barrier. But this way, it is easy to do trouble shooting, and keep track of how many intervals get divided at each level. And inside each parallel for loop, we need to use `#pragam omp critical` to update `tailIdx` and `M`, so that these global variables can only be touched by each thread at a time.

Another option would be to parallelize the while loop as long as there is still interval in the priority queue. But the issue with that is I don't have management over which thread gets which interval. This way, I am getting control over assigning the job myself, and easier to debug. It is more convenient to work with parallelizing for loop when you know what should be the result at each step.

```
//initialize first interval
double eps, gprime; leftNode[0]; rightNode[0]; gvalL[0]; gvalR[0];
M = max(M,max(gvalL[0],gvalR[0]));
int headIdx, tailIdx, numOfInterval, numIncrement;
bool nonEmpty = true;

double seconds = omp_get_wtime ( );   //start timing
//local private
double left, right, mid, leftVal, rightVal, midVal;

//continually apply the max possible checking procedure until no ...
    further refinement required
while( nonEmpty ){

  numIncrement = 0;   //num of intervals increased to update tail index

  //start parallelize each level of intervals (each interval length ...
      gets divided by half compare to last while loop)
  #pragma omp parallel for ...
      private(left,right,mid,leftVal,rightVal,midVal) ...
      reduction(+:numIncrement)
  for(int i=0; i < numOfInterval; i++){   //loop over intervals from ...
      head to tail
    int localHead = (headIdx + i) % numel;  //head index for each ...
        thread, manually assign job
    left = leftNode[localHead]; right = rightNode[localHead]; ...
        leftVal = gvalL[localHead]; rightVal = gvalR[localHead];
    if((leftVal + rightVal + gprime*(right-left))/2 ≥ M+eps){  ...
        //dividing criteria
      mid = (left + right)/2;
      midVal = g(mid);
      int localTail(0);
      numIncrement += 2;
      #pragma omp critical  //update tail index and function max one ...
          thread at a time
      {
        localTail = tailIdx;
        tailIdx = (tailIdx + 2) % numel;
        M = max(M,midVal);
      }
      leftNode[localTail] = left; rightNode[localTail] = mid; ...
          gvalL[localTail] = leftVal; gvalR[localTail] = midVal;
      leftNode[localTail+1] = mid; rightNode[localTail+1] = right; ...
          gvalL[localTail+1] = midVal; gvalR[localTail+1] = rightVal;
    }
  }
  if(numIncrement == 0){
    nonEmpty = false;
  }
  headIdx = (headIdx + numOfInterval) % numel;  //update new head index
  numOfInterval = numIncrement;
}
```

2. **Performance and Speedup Analysis**
The results this time are also changing quite a bit over time. And mostly because of trying to use different compiler and options. Initially, both on my personal computer and on the PSC bridges, I was use GNU compiler to compile the code

```
$ g++ -fopenmp hw4_hszhu.cpp -o hw4_hszhu
```

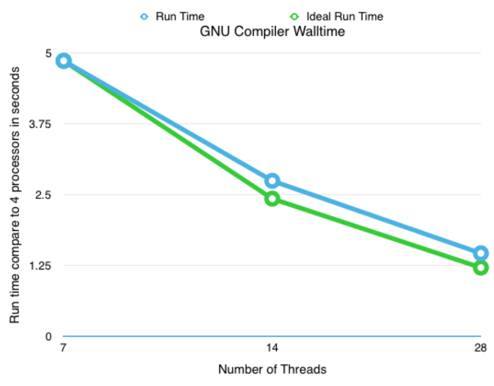
And the result is alright.

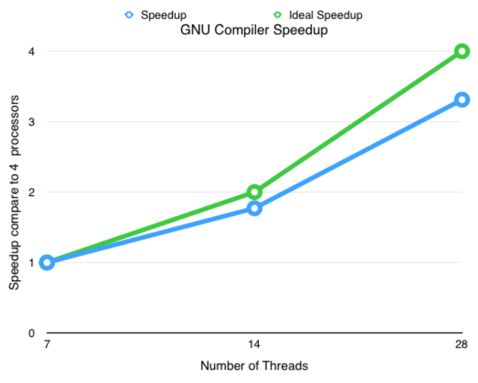
```
+ export OMP_NUM_THREADS=28
+ OMP_NUM_THREADS=28
+ ./hw4_2 1 100 0.00001 12
================================
max value is 4.9999999396063046
```

```
wall time is 1.4688526438549161

+ export OMP_NUM_THREADS=14
+ OMP_NUM_THREADS=14
+ ./hw4_2 1 100 0.00001 12
================================
max value is 4.9999999396063046
wall time is 2.7473644036799669

+ export OMP_NUM_THREADS=7
+ OMP_NUM_THREADS=7
+ ./hw4_2 1 100 0.00001 12
================================
max value is 4.9999999396063046
wall time is 4.8649620683863759
```





Then I tried to use intel's compiler, which seems to speed things up about 3 times.

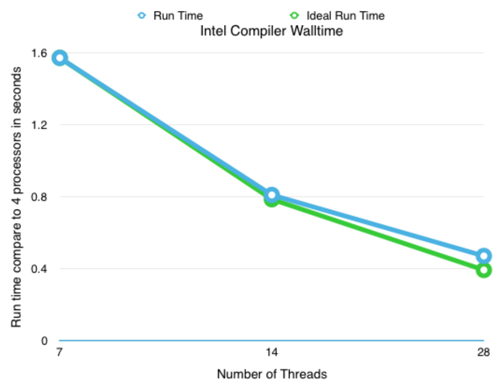
```
$ icpc -qopenmp hw4_hszhu.cpp -o hw4_hszhu
```

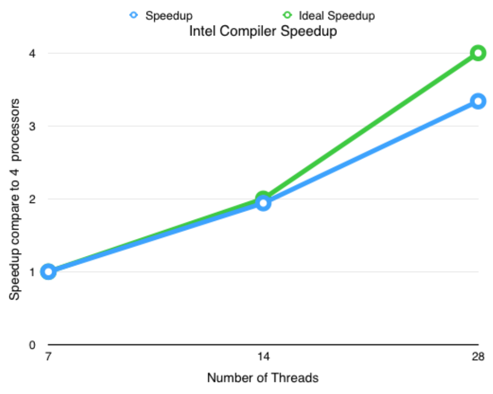
The result is much better. Here is the run time result with intel's compiler.

```
+ export OMP_NUM_THREADS=28
+ OMP_NUM_THREADS=28
+ ./hw4_2 1 100 0.00001 12
================================
max value is 4.9999999396063046
wall time is 0.47083902359008789

+ export OMP_NUM_THREADS=14
+ OMP_NUM_THREADS=14
+ ./hw4_2 1 100 0.00001 12
================================
max value is 4.9999999396063046
wall time is 0.80996108055114746

+ export OMP_NUM_THREADS=7
+ OMP_NUM_THREADS=7
+ ./hw4_2 1 100 0.00001 12
================================
max value is 4.9999999396063046
wall time is 1.5718410015106201
```





And further more, I turned on compiler optimization option. It runs about twice faster than even before, which surprises me.

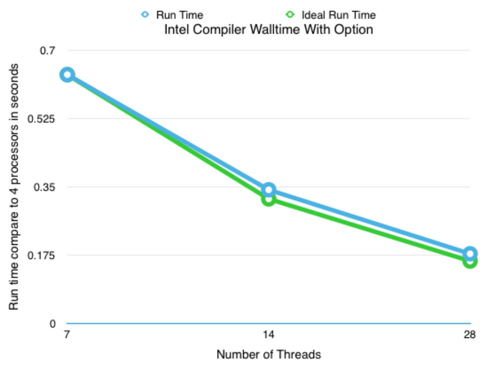
```
$ icpc -qopenmp -xHost -O2 hw4_hszhu.cpp -o hw4_hszhu
```

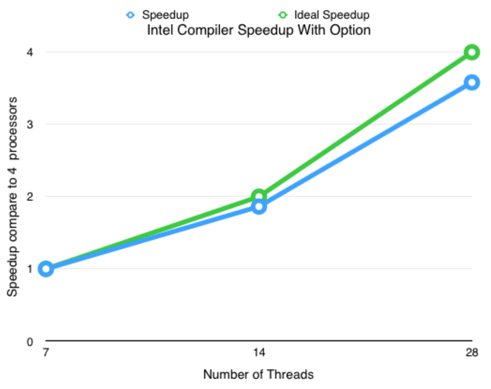
And here is the result for intel's compiler with optimization option

```
+ export OMP_NUM_THREADS=28
+ OMP_NUM_THREADS=28
+ ./hw4_2 1 100 0.00001 12
================================
max value is 4.9999999396063046
wall time is 0.17768597602844238

+ export OMP_NUM_THREADS=14
+ OMP_NUM_THREADS=14
+ ./hw4_2 1 100 0.00001 12
================================
max value is 4.9999999396063046
wall time is 0.3415379524230957

+ export OMP_NUM_THREADS=7
+ OMP_NUM_THREADS=7
+ ./hw4_2 1 100 0.00001 12
================================
max value is 4.9999999396063046
wall time is 0.63643693923950195
```





And there is one other interesting thing I observed while trying to run the code with 56 threads. Because of the architecture of PSC bridges, in the `RM` allocation, we are not supposed to use more than 28 cores, otherwise it has to run across different nodes, which are no longer a shared memory architecture. And the thing I found out is that with GNU compiler, when you try to run 58 threads, things won't get speed up, but it won't slow down either. But with intel's compiler, the code runs way slower when you export 56 threads, and there are really only 28.

We could see consistent behavior from all the above walltime, and speedup graphs. As we parallelize the problem from $p = 7$, 14 to 28 threads, we should expect twice and four times speedup. And that is basically what we see from the real runtime. We are not getting the perfect speedup. And there are two main reasons from my implementation. The first one is caused by initialization and barrier of OpenMP, where I called OpenMP inside a while loop. A serial of parallelized for loop with barrier would cause some extra time to initialize OpenMP, and to synchronize after OpenMP was done. Another reason is because of the use of `critical` to ensure no thread is messing with current maximum value of the function, and the tail index. And these lines are operated once at a time by the threads, which prevent us from getting perfect speedup.

Initially, I thought there might be a chance to get perfect, or even super-linear speedup. Because compared to serial, parallel updating maximum would probably benefit us in a way that less intervals are going to be examined. But in this case, updating `M` happens after the evaluation step of function $g(x)$, which is a really complicated function, and causes a lot of time to compute. Therefore, it is highly unlikely that one thread is going to be able to update its `M` before some other thread to get the point of `if` (checking criteria) in order for that other thread to benefit.