UNIVERSITY OF MICHIGAN

EECS 587 Homework 3

Yisen Wang 77072987
yisenw@umich.edu

October 26, 2021

1 Parallel Algorithm for Calculation

We combined BFS with DFS in order to achieve a good performance. Since the largest possible slope is s, and the tolerance is ϵ, the minimal length of an interval is ϵ/s, meaning if the divided intervals have a length less than ϵ/s, we will not divide it further and stop at this point. Our algorithm runs as follows:

1. Construct a vector which contains the whole interval.

2. Run BFS several times on the interval on a single processor to get several small intervals. We set the BFS to stop until the vector has more elements than the number of processors.

3. Assign the elements in the vectors evenly to every processor. Assume we have p processors and we have n elements in the vector, every processor would get n/p elements. At this time, the vector would be empty after all the assignments.

4. Each processor run DFS 100 times on its assigned intervals parallelly. After runs 100 times (or the process stop before 100 times), the process will push its remaining intervals (if any) back to the vector in step 3.

5. Go the step 3 and repeat the process until there are no intervals left in the vector defined in step3.

2 Explanations for the Algorithm

In step 4, we push the intervals in each process back to the global vector because we want to balance the work in each process. After running 100 times, some processors may still have a lot of intervals to process, while others no longer have any intervals left. If we do not balance the workload, some processors will be wasted, causing a longer runtime. Step 4 is also a successful attempt of mine to combine BFS and DFS to optimize the performance.

Another critical point is that we find the function g takes a really long time to calculate. As a result, we should minimize the calculation of $g(x)$. So in our code, for each x, we only calculate $g(x)$ once and store its value for future use.

3 Performances

3.1 The Experiments

number of processor	time(s)	speed up	efficiency
$p = 1$ (serial)	227	NA	NA
$p = 9$	27	8.41	0.934
$p = 18$	14	16.21	0.901
$p = 36$	8	28.4	0.788

Table 1: Performances

3.2 Analysis

We find that we implement the parallel algorithm pretty well. For $p = 9$, the efficiency is pretty high. For $p = 18$, the efficiency gets lower, but still have 0.901. For $p = 36$, the deficiency has an obvious drop to 0.788 (which is still not bad). Viewing from the data, we find that our speed-up and efficiency has a reasonable decreasing with the increasing of the number of processors.

4 Running our Codes

In the zip file named "yisenw_hw3.zip", the most important file is "hw3_faster.cpp" and "g.h". If you'd like to test on other function g, feel free to directly replace "g.h". If you'd like to change the value of a, b, s or ϵ, you can change it directly in line 32-35 of "hw3_faster.cpp". If you'd like to run our codes, first you need to compile it using the following command

g++ -std=c++11 -O0 -fopenmp -o s hw3_faster.cpp

and generate an executable file "s". Then you may run the executable file directly by the following command:

OMP_NUM_THREADS=$< p >$ ./s

Where you could replace $< p >$ to be the number of processors (e.g. 9, 18, 36).