CMPSC 240A HW-4 Report

Tanmoy Sanyal

November 9, 2016

Why I didn't use cilk

I tried my best to get a working code with cilk, but gave up eventually and turned to openMP. I will list a quick overview of what I learned from my failures with cilk and then sketch the essential details of the openMP implementation.

The assignment description hinted at maintain $O(np)$ space complexity where p is an upper bound on the number of threads. This was the single most important guiding principle that I tried to implement throughout. There are two ways to do this in any shared-memory-parallelization model. Either (a) declare all your thread-private variables and initialize them inside the parallel for construct, so that their scope is limited to the thread and they can be updated without race condition, or (b) declare a master array of size $O(np)$ for potential thread-private data of size n and use the thread id to point to the correct position of this master structure, so that again updates to the data happen at non-contending locations. Strategy (b) can also be used to implement a array of reducers.The key feature of both (a) and (b) are that they ameliorate the use of mutex locks which do prevent data races but also cause slow-down by arbitrary amounts.

Both schemes (a) and (b) are easy to apply with cilk. However, for scheme (a), I had difficulty convincing myself that the number of re-declarations is really equal to the number of cilk strands (p) and not equal to the number of iterations (n). For scheme (b), it was essential to obtain the cilk worker id. This is possible using `__cilkrts_get_worker_number`, and is akin to a thread-id in a less-than-strict-sense of the term. However, some recent literature on cilk as well as a chance discussion with Barry Tenenbaum from Intel over a forum, convinced me that thread-id like approaches are a bad idea in cilk. The "cilk-ian" philosophy is to create new hyperobjects to suite your purpose, since hyperobjects are thread-private by definition and guarantee race-free and serial-like behavor. Reducer hyperobjects are common and we have seen them in HW3. My first thought was to simply create arrays of reducers for the update variables like σ, δ, d and `BC`, and then use their `set_value()` and `get_value()` methods to update them. Turns out that these methods are specifically for initialization before the parallel region and extracting output after that region. Using them to obtain thread-local values, can and will cause indeterminate behavior. The solution to this problem are so called "holder-reducers" which can be realized by creating a new reducer object and then leaving its reduce functionality blank. Thus, these are dummy reducers whose updates and assignments are thread-safe. I found this easy to grasp, but very difficult to code, given my less than proficient background in C/C++.

At this point, I switched over to openMP. Using thread-id s are common in openMP and parallelizing the outer loop was easy.

Brief overview of the openMP parallelization

I followed strategy (b) discussed in the previous section. So all necessary thread-private variables: the BFS queue `S`, predecessor lists `P`, BFS queue iteration managers `start`, `end`, centrality calculation variables σ, δ, d, `BC` were given n * `MAX_THREADS` long storage in shared memory. Inside the for loop, the thread id was extracted and use to point to the n-sized chunk of these master arrays, appropriate to the thread. Finally, outside the parallel region, the values of `BC` for each vertex were added from all the threads, to construct the output. The number of BFS traversals `num_traversals` was also shared, and its update was done using a #pragma `omp atomic` region, which, albeit similar to a lock, is considerably faster than using explicit locking like #pragma `omp critical`.

However, even after careful consideration of the parallelization the code is still buggy because I still have data races and incorrect final answers for large to moderately large sized graphs. I moved ahead with the scaling experiments anyway.

Scaling with size of graph

For this experiment,I employed all 48 cores on a Comet node. The number of BFS traversals were limited to 100.

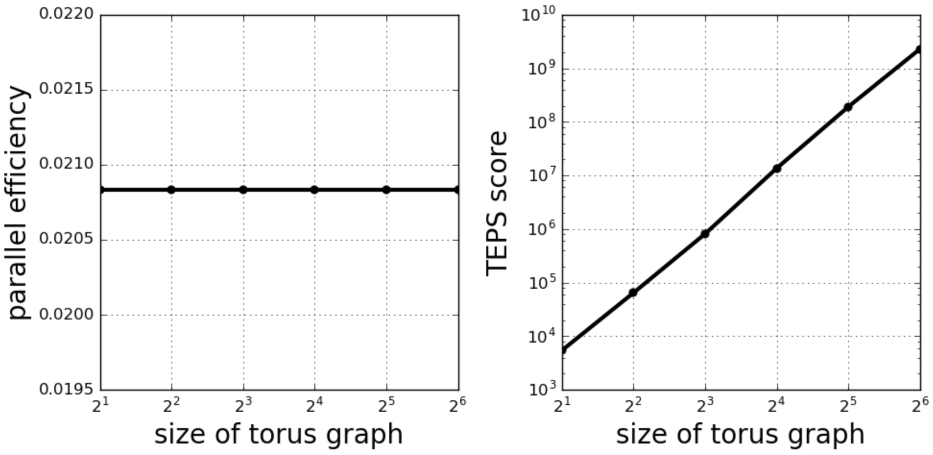

Figure 1: Parallel efficiency and TEPS score with varying graph size

Scaling with number of cores

Here, the (torus) graph size was kept fixed at 64x64. The number of BFS traversals were again limited to 100.

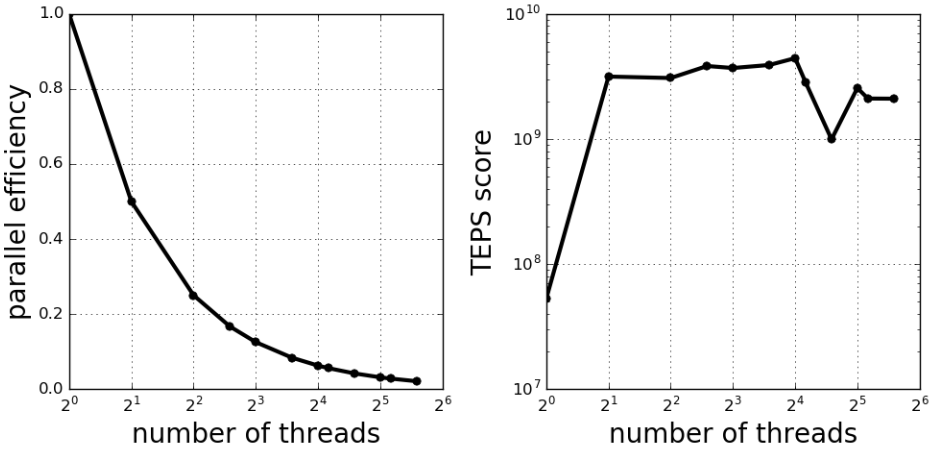

Figure 2: Parallel efficiency and TEPS score with varying thread pool

Discussion

For the scaling with number of threads in Fig.2 , I find it surprising that the speedup continues while the TEPS score starts decreasing. The scaling of parallel efficiency with graph size in Fig 1 is bizarre and most probably incorrect, because of all the data racing that I was unable to get rid of. So, while I learnt a lot from this assignment, it is unlikely that either of my scaling results are correct.