COMP90025 Parallel and Multicore Programming
HONGFEI YANG
hongfeiy1

The serial FW algorithm has a pessimistic |V|^3 time-complexity, where V is the number of vertices. The outer most loop k, the 'pitstop' in the path, cannot be parallelised since i and j depends on k, only the inner loops can be parallelised. One may simply put a #pragma omp parallel for before the second loop to achieve a very simple parallelism. However, the approach I took is to divide the input distance matrix into equally sized tiles in Figure 1, then update each tile by strictly adhering to the algorithm's dependency to parallelise FW algorithm on multiple tiles. Below are the details of this algorithm:

```
for (k=0; k<|V|; k+=tile_size)
      FW(centre tile)

      For other tiles in the same row and column do in parallel:
            FW(tiles)
      wait for all threads to finish

      for the rest of the tiles do in parallel:
            FW(tiles)
      wait for all threads to finish
```

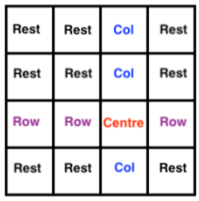
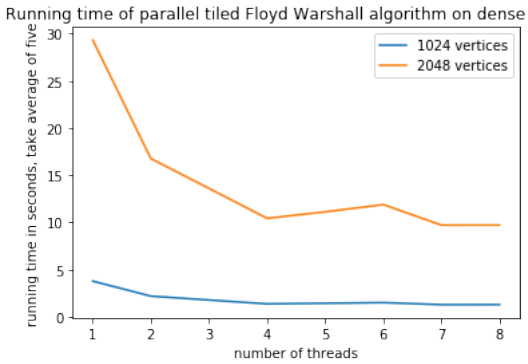


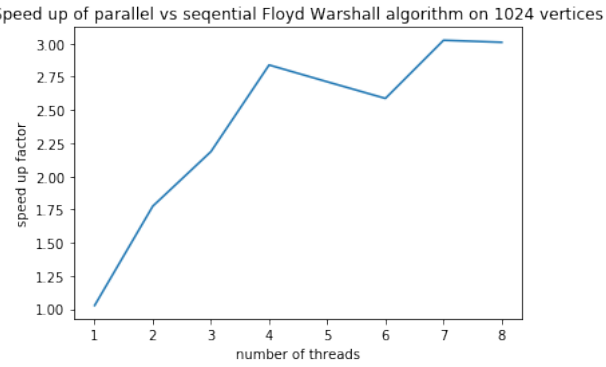
Figure 1: Layout of tiles

At the k-th iteration, the independent centre tile is updated first by running FW algorithm. After the centre tile is updated, the data-dependent tiles in the same column or row can be updated simultaneously, since the tile they depend on has already been updated, i.e. the rest of the tiles has been updated to the k-1 th iteration while the central tile is already updated before. After all row and column tiles are updated, the rest of the tiles may be updated simultaneously since the tiles they depend on has already been updated. By updating matrix in this order, parallelism could be enabled.

Suppose the matrix is divided into tiles of size B, which makes (|V|/B)^2 tiles in total. There are |V|/B iterations in total by the algorithm. In each iteration, all tiles are accessed and updated. Updating a tile requires holding 3 tiles in memory, a total of 3*B^2 vertices in the matrix. So the total amount of memory access is calculated by (|V|/B) which is the total number of iterations, multiply by (|V|/B)^2 which is the total amount of tiles accessed in each iteration, multiply by 3*B^2 which is the number of memory access in each tile update. A total of 3*|V|^3/B number of memory access, a function of O(|V|^3/B). Thus the total amount of memory access is reduced by a factor of B.

Suppose there are p processors and each processor can update a number of B^2 vertices in O(1) constant time, the sequential algorithm would require O ((|V|^3)/(|B|^2)) time to process. The parallel algorithm has |V|/B number of iterations, in each iteration, updating central takes B^2 vertices updates, updating row and columns takes 2*(|V|-B)*B vertices updates and updating the rest takes |V|^2 - B^2 - 2*(|V|-B)*B vertices updates. Since updating row and column as well as updating the rest tiles can be done in parallel, so the run time is O(B^2/(p * B^2) + 2*(|V|-B)*B / (p * B^2) + (|V|^2 - B^2 - 2*(|V|-B)*B) / (p *B^2)), which is O(|V|^2/(p * B^2)) in each iteration, so the algorithm's time complexity is O(|V|^2/(p * B^2)) * |V|/B number of iterations, which is O(|V|^3 / (p * B^3)). The sequential algorithm using only one processor will give a time complexity of O(|V|^3/B^2). The speed up of the parallel algorithm is p*B. If keeping tile size B fixed, theoretically it can achieve a linear speed up. Below are the diagrams:





* Note that all experiments results are based on algorithms running on a machine with an Intel Core i7-7920HQ and 16GB of RAM, tile size is set to 128 vertices

The speed up between 1 to 4 threads are near linear, however after that it tends to struggle to achieve linear speed up. This may be caused by the physical limit of the 4-core CPU.

However, one must careful choose the tile size in order for this parallel algorithm to work well. If the tile size is larger than the cache of the machine, the machine may not store an entire tile in its cache and have to fetch the rest of this tile while updating a tile, thus wasting an unnecessary amount of operations in doing so.

Dijkstra's algorithm has a worst case complexity of O(|E| + |V|log|V|), running repeatedly on |V| vertices gives a worst case complexity of O(|E||V| + |V|^2*log|V|). If the matrix is very sparse, i.e. |E| is much smaller than |V|^2, then it is better to use Dijkstra's algorithm than FW algorithm, otherwise FW algorithm would be efficient.