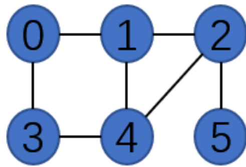
Final Project

100 points (20% of the entire grade)
Due: 6:15pm Monday, May 11, 2020

The goal of this project is to understand some benefits and overheads of thread level parallelism through two multithreaded Breadth-First Search (BFS) implementations, that is, frontier queue and status array-based options.

Particularly, classical BFS algorithm [1] starts at the root of the graph and inspects the status of all of its adjacent (or neighboring) vertices. If any adjacent vertex is unvisited, the algorithm will identify it as a frontier, and subsequently mark it as visited in the status array. In the next iteration, we will inspect the adjacent vertices of frontiers from the preceding level and generate frontiers for the next level. This process continues until all the vertices are visited.

Figures (c) and (d) explain how frontier queue and status array based BFS works on the toy graph in Figure (a) that is represented in the Compressed Sparse Row (CSR) format in Figure (b). Let us use this toy example to explain how two threads can a BFS traversal for this toy graph. Note the root of the BFS is 0.



Vertex	0	1	2	3	4	5
Begin pos	0	2	5	8	10	13
Adjacency	1,3	0,2,4	1,4,5	0,4	1,2,3	2

(a) Sample graph (b) CSR format

At level 2 of the BFS, frontier-queue based approach will schedule these two threads to work on vertex 1 and 3 in FQ2 as shown in Figure (c). Assuming threads 1 and 2 identified 2 and 4 as unvisited neighbors, respectively, these two threads will try to put 2 and 4 into FQ3 concurrently. In this case, atomic operation (i.e., __sync_fetch_and_add) [3] is needed to ensure that both 2 and 4 in FQ3 are recorded in FQ3 without overwritten.

FQ1	0	
FQ2	1	3
FQ3	2	4
FQ4	5	

(c) Frontier queue based BFS

A second approach does not generate any frontier queue, it directly uses the status array to indicate active vertices in order to avoid atomic operation. Particularly, the status array is a byte array indexed by the

vertex ID. The status of a vertex is represented by its BFS level. At every level, each vertex will be checked, whereas only those that are visited in the preceding level are treated as frontier. In this case, the thread will perform expansion and inspection. From table (d) we know that 2 threads check the status of all vertices at level2, but only vertices 1 and 3 will be the frontiers. Compared with the first approach, atomic operation is no longer needed.

Vertex	0	1	2	3	4	5
SA1	F	U	U	U	U	U
SA2	0	F	U	F	U	U
SA3	0	1	F	1	F	U
SA4	0	1	2	1	2	F
SA5	0	1	2	1	2	3

(d) top-down bfs using status array

The algorithms of both methods work as follows

//Algorithm 1: Single-threaded BFS frontier queue

```
char* BFSTraverse(graph<long, long, int, long, long, char>* ginst,int source)
{
        int j,frontier =0;

        char* statusArray = new char[ginst->vert_count];
        for (int i = 0; i < ginst->vert_count; i++)
                statusArray[i] = -1; //-1 means unvisited;

        int* currFrontierQueue = new int[ginst->vert_count];
        int* nextFrontierQueue = new int[ginst->vert_count];
        int currFrontierSize = 0;
        int nextFrontierSize = 0;
        currFrontierQueue[currFrontierSize] = source;
        currFrontierSize++;
        statusArray[source] = 0;
        int myFrontierIndex = 0;
        int currLevel = 1;
        while (true)
        {
                while (myFrontierIndex < currFrontierSize) {
                        frontier = currFrontierQueue[myFrontierIndex];

                        long int beg = ginst->beg_pos[frontier];
                        long int end = ginst->beg_pos[frontier + 1];

                        for (j = beg; j < end; j++) {
                                if (statusArray[ginst->csr[j]] == -1) {
                                        statusArray[ginst->csr[j]] = currLevel;
                                        nextFrontierQueue[nextFrontierSize] = ginst->csr[j];
                                        nextFrontierSize++;
                                }
                        }
                        myFrontierIndex++;
                }
                if (nextFrontierSize == 0) { return statusArray; }

                //Swap current and next frontier queue;
                currFrontierSize = nextFrontierSize;
                myFrontierIndex = 0;
                nextFrontierSize = 0;

                int* temp = currFrontierQueue;
                currFrontierQueue = nextFrontierQueue;
                nextFrontierQueue = temp;
```

```
			currLevel++;
		}

}
```

//Algorithm 2: Single-threaded BFS status array

```
char* BFSTraverse2(graph<long, long, int, long, long, char>* ginst,int source)
{
		int ptr;
		int j;

		char* statusArray = new char[ginst->vert_count];
		for (int i = 0; i < ginst->vert_count; i++)
			statusArray[i] = -1; //-1 means unvisited;

		statusArray[source] = 0;
		int myFrontierCount = 0;

		int currLevel = 0;

		while (true)
		{
			ptr = 0;
			while (ptr < ginst->vert_count) {

				if (statusArray[ptr] == currLevel) {
					int beg = ginst->beg_pos[ptr];
					int end = ginst->beg_pos[ptr + 1];

					for (j = beg; j < end; j++) {
						if (statusArray[ginst->csr[j]] == -1) {
							statusArray[ginst->csr[j]] = currLevel+1;
						}
					}
				}
				else if (statusArray[ptr] != currLevel) {
					myFrontierCount++;
				}
				ptr++;
			}
			currLevel++;

			if (myFrontierCount == ginst->vert_count) {
				return statusArray;
			}
			myFrontierCount = 0;

		}

}
```

You need to use OpenMP to implement the multi-threaded BFS. Particularly, OpenMP [2] consists of a set of compiler primitives - #pragmas - that control how the program works. The pragmas are designed so that even if the compiler does not support them, the program will still yield correct behavior, but without any parallelism. Below we explain how single-threaded and multithreaded vector-vector addition works with OpenMP.

```
//Algorithm 3: Single-threaded vector reduction
int vect_add (int *a, int num)
{
        int res = 0;
        for (i = 0; i < num; i ++)
        {
                res += a[i];
        }
        return res;
}
```

```
//Algorithm 4: Multithreaded vector reduction
int vect_add (int *a, int num, int thread_count)
{
        int res = 0;
        #pragma omp num_threads (thread_count)
        {
                int work_per_thread = num/thread_count;
                int my_thread_id = omp_get_thread_num();
                int my_beg = my_thread_id * work_per_thread;
                int my_end = my_beg + work_per_thread;

                int my_res = 0;
                //In case num cannot be evenly divided by thread_count
                if (my_thread_id == thread_count - 1)
                {
                        my_end = num;
                }
                while (my_beg < my_end)
                {
                        my_res += a[my_beg];
                        my_beg ++;
                }
                __sync_fetch_and_add(&res, my_res)
        }
        return res;
}
```

Assignments: In this project, you need to turn in a final project report, along with the source code to fulfill the following tasks:

1. [20 points] Comparing the performance difference of sing-threaded frontier queue and single-threaded status array-based implementations on the following four datasets:
    a. Orkut: https://snap.stanford.edu/data/com-Orkut.html
    b. Pokec: https://snap.stanford.edu/data/soc-Pokec.html
    c. Livejournal: https://snap.stanford.edu/data/com-LiveJournal.html
    d. Youtube: https://snap.stanford.edu/data/com-Youtube.html

    Please treat all of these graphs undirected when using tuple_text_to_binary_csr_mmap from https://github.com/asherliu/graph_project_start Further, you will use graph_reader from the same github repository to read the graph and get the ginst object. Finally, you will use this object to implement the desired BFS. In this homework, the single-threaded BFS implementations (both frontier queue and status array-based options) are provided to you.
2. [20 points] Making status array based BFS multithreaded with OpenMP by following the syntax in Algorithm 4.
3. [20 points] Making frontier queue based BFS multithreaded, again, with OpenMP.
4. [20 points] Comparing the performance of both methods on four datasets. The number of threads should increase from 1 to 2, 4, 8 and 16.
5. [20 points] Understanding the performance differences

References:

[1] Liu, Hang, and H. Howie Huang. "Enterprise: breadth-first graph traversal on GPUs." In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-12. 2015.
[2] Guide into OpenMP: Easy multithreading programming for C++. https://bisqwit.iki.fi/story/howto/openmp/
[3] Built-in functions for atomic memory access https://gcc.gnu.org/onlinedocs/gcc-4.1.1/gcc/Atomic-Builtins.html