Parallel BFS using POSIX Threads

1. Improvement to the Algorithm – Properly Load Balanced

Algorithm thread_pool(*int id)*

```
1.  while(true) repeat
2.      if(id == choice) then
3.          wait_for_signal()
4.          if(Q.empty())
5.              exit(0)
6.          while(Q is not empty) repeat
7.              write_to_file(Q.front)
8.              TQ[k%NUM_THREADS].push_back(Q.front)
9.              Q.deque()
10.             k++
11.         endWhile
12.         choice++
13.         wake_up_threads()
14.         for all vertices in its' private queue TQ[k]
15.             node ← TQ[k].front()
16.             for u in adjacencies of node do
17.                 if(V[u] == false) then
18.                     V[u] = true
19.                     TQout[id].push(u)
20.                 endIf
21.             endFor
22.         endFor
23.     endIf
24.
25.     else
26.         for all vertices in its' private queue TQ[k]
27.             node ← TQ[k].front()
28.             for u in adjacencies of node do
29.                 if(V[u] == false) then
30.                     V[u] = true
31.                     TQout[id].push(u)
32.                 endIf
33.             endFor
34.         endFor
35.     endElse
36.
37.     if(id != choice)
38.         wait_for_signal();
39.     endIf
40. endWhile
```

2. Explanation

In the previous algorithm there was a main thread that executed the function of a master thread. It allocated works to the worker threads via individual thread queues and then waited for the thread to complete their working and repeated this process until all vertices were traversed. The problem with this algorithm was that the master thread was remaining idle for the time period when the other threads worked. This affected the performance and also the fact that work was not properly load balanced among the threads.

To overcome this problem, the proposed algorithm is used. This algorithm does not create any master thread. Instead a number of thread are created as a team of threads. This team of threads does the work of both the master and slave threads in the previous algorithm. Among the threads in the team, one thread is chosen to do the allocation work for the rest of the threads. After allocation, that thread along with the others starts working itself. This working reduces the waiting time. Earlier a thread was waiting during this computation but now it also performs some work. In the next iteration a new thread is chosen to do the allocation work. This way of choosing different threads ensures that the work load is uniformly divided among the set of threads.

This algorithm thus takes care of both load balancing and waiting tie reduction. The work allocation step that is performed by one thread takes a minimal amount of time in comparison to the time taken to perform vertices adjacencies calculations as seen by the profiling results. Thus we can safely assume that the impact of that time will be insignificant.