Parallel Breadth First Search with OpenMP

Zhanrui Zhang
Shanghaitech University
zhangzhr2@shanghaitech.edu.cn
Id: 2019533227

ABSTRACT

In this lab, we implement a parallel breadth first search algorithm with OpenMP on a shared memory system. First we implement a sequencial algorithm, then we adapt it into a parallel version. Several techniques are used to synchronize between different threads. We benchmark these algorithms on some different graphs and analyze the result.

1. INTRODUCTION

The classic implementation of sequential bfs is to use a FIFO queue. Each time we pop the first node in the queue and push all its unvisited neighbours into the queue, until the queue is empty. A quite simple idea to parallelize this process is to use two list representing current layer and next layer. By expanding a frontier layer by layer from the source, and put all unvisited neighbour of nodes in current layer into the next layer, we can visit all nodes in the connected part the source nodes belongs to.

2. ALGORITHMS

The general idea is to use two list, containing current frontier and next layer. All threads pick nodes from current layer concurrently and add those unvisited neighbours into next layer.

2.1 Local List

Since each queue is gathering neighbours and try to push them into the next layer list concurrently, there exits some race condition on the next layer list. Using lock is a straight forward solution, but it is extremely inefficient. Here we create a private list for every thread, and temporary store the nodes into it.

2.2 Atomic Operation

After all threads have gathered their neighbour, we should combine all local list into a shared list as the next frontier. Each threads need to write to a same list, at different location. There are several ways to achieve this. The first approach is to sequentially loop through all threads by there id, and each threads just copy their local data to shared list. There is some unnecessary waiting time here, since we just need to know from which index each thread should start to write their local list and then they can copy data simultaneously. So we use an atomic operation fetch and add to calculate a writing index atomically. It is basically equals to a lock with much smaller grain size. Also, we can use prefix sum to calculate the writing index, but after experiment, we found that prefix sum is not significantly faster than atomic operation given relative small number of threads on CPU. So here we just use atomic operation.

When we are copying local list into shared list, we have already gathered all neighbours of current layer. So we do not need two global list for current layer and next layer, we just insert local list into current layer list and start next loop. In this way, we can save memory and improve performance.

We use distance array to keep track which nodes are visited. Unvisited nodes' distance are set to -1. When updating them, we DO NOT need an atomic CAS operation, because CAS is slow. When multiple threads try to update the same node's distance, we just let them race and keep the last value. Race condition only happens in the same layer, so the distance will be correct anyway.

2.3 Schedule

We do not know in advance how many neighbours to gather for each node picked from current layer list, so it is very likely that even if we assign same number of nodes to each threads, the load will still be quite imbalanced. And we also cannot have any information in advance about which nodes have a lot of neighbours to check and which nodes do not. This makes all static schedule same inefficient. They basically rely on the randomness of a number of neighbours of a graph. So here we use dynamic schedule. But meanwhile, we do not need that small grain size, when each thread just fetch one node from waiting list and gather its neighbour. After testing, we found that dynamic schedule with grain size 4 works well for our test case. So we use schedule (dynamic, 4).

3. BENCHMARK

3.1 Test Results

We test our algorithm on 7 different graphs, the result are as shown in figure 1. In each round we pick same 20 random nodes as source node. The result is averaged over 3 rounds.

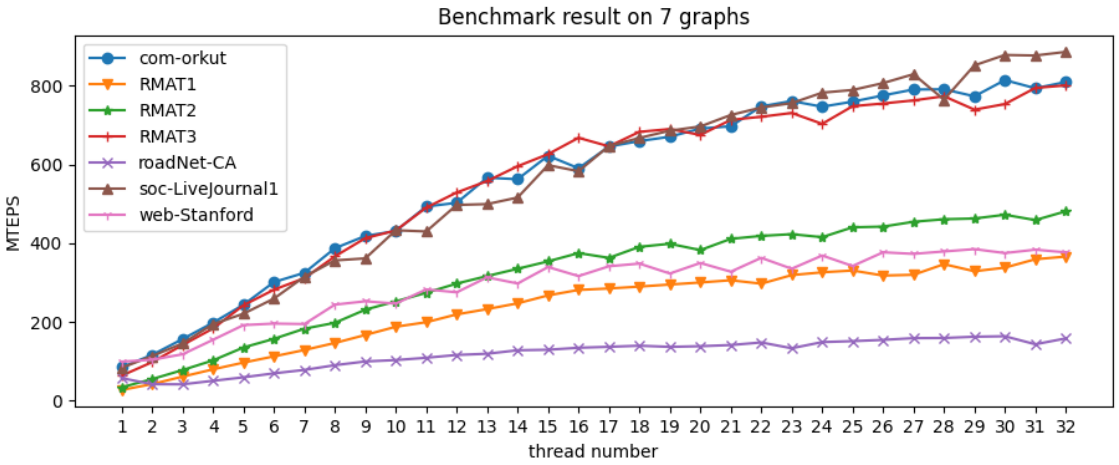

Figure 1: Benchmark Result

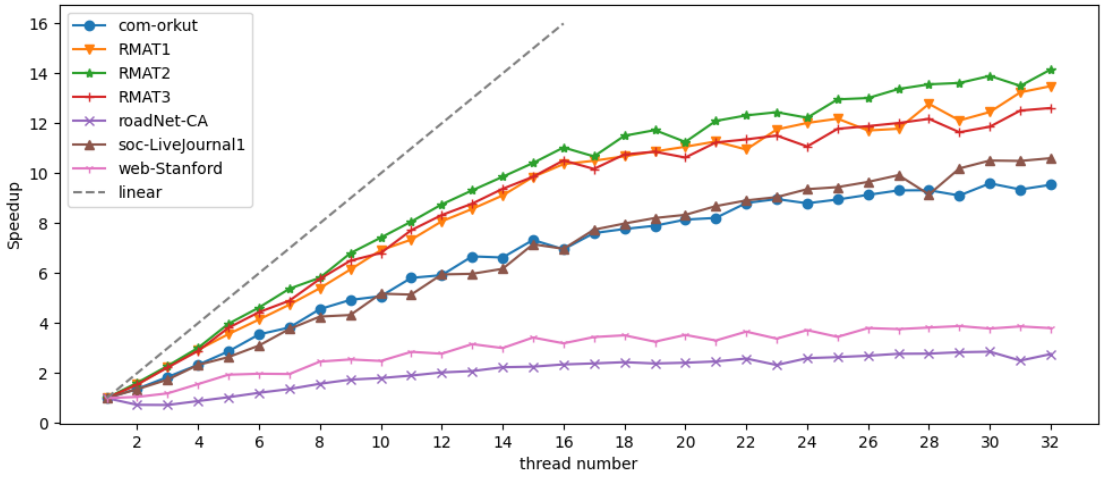

Figure 2: Speed Up

Graph name	MTEPS	MTEPS	Speed up
web-Stanford	99.14	376.85	3.80
roadNet-CA	57.26	158.19	2.76
com-orkut	84.89	809.91	9.54
RMAT1	27.14	366.15	13.49
RMAT2	33.99	481.3	14.16
RMAT3	63.52	801.34	12.61

Table 1: The benchmark is performed on AMD Epic ROME platform(32 cores@3.3GHz), with 64GB memory.

Then we calculate the average speed up given each thread number. As can be seen from the figure, we achieve highest performance at maximum thread number.

3.2 Analysis

For web-Stanford and roadNet-CA, they have relative large diameter, which means there is not many neighbours in each layer, but we need many loops to iterate through all layers. This is not friendly to parallizing the task. soc-LiveJournal1 and three rmat graph is very large, and thus the schedule overhead will be smaller in comparison. So they have the best speed up.

In future, we can implement some efficient array-like data stucture, support fast and flexible memory allocation to speed up our algorithm. For example, we can allocate chunks of memory and use to chunked list instead of continuous list.

4. REFERENCE

Evaluating Parallel Breadth-First Search Algorithms for Multiprocessor Systems *Matthias Makulla and Rudolf Berrendorf*

A Work-Efficient Parallel Breadth-First Search Algorithm *Charles E. Leiserson, Tao B. Schardl*

Table 2: Frequency of Special Characters (MTEPS/std deviation)

Thread Num	web-Stanford	roadNet-CA	com-orkut	soc-LiveJournal1	RMAT1	RMAT2	RMAT3
1	99.14/36.51	57.26/0.86	84.89/41.39	83.58/32.37	27.14/0.05	33.99/0.08	63.52/51.22
2	103.84/12.14	41.83/0.34	116.3/42.11	113.24/0.55	42.07/0.1	54.78/0.19	97.93/78.95
3	117.93/15.41	41.5/0.54	156.61/58.16	145.48/0.82	61.07/0.15	77.76/0.07	141.36/0.26
4	154.7/23.31	50.46/0.2	198.13/65.26	194.55/1.11	79.58/0.25	102.36/0.14	183.47/0.33
5	192.07/7.83	59.32/0.26	243.82/50.51	220.95/1.33	96.73/0.22	135.22/0.23	242.45/0.47
6	195.85/8.47	69.41/0.46	301.32/37.31	259.4/1.16	112.52/0.3	157.18/0.46	282.03/0.67
7	194.72/8.35	78.06/0.51	324.95/43.38	314.99/1.83	128.72/0.36	182.86/0.14	311.32/1.21
8	244.03/11.09	90.12/0.58	387.37/50.22	356.42/2.08	146.58/0.68	197.8/0.14	366.64/1.09
9	252.56/11.59	99.61/0.91	418.55/35.86	361.58/1.77	166.88/0.98	231.19/0.17	412.98/0.59
10	246.6/11.35	103.15/1.17	431.17/41.87	433.03/2.44	187.85/0.21	252.35/0.15	432.55/1.66
11	282.74/14.38	108.96/1.38	493.43/43.39	430.0/2.58	199.32/0.21	274.17/0.27	490.56/0.98
12	275.34/13.58	116.12/1.82	502.77/47.03	497.44/3.38	219.23/0.44	297.57/0.27	529.03/1.23
13	313.41/15.4	119.16/1.62	566.65/52.78	499.55/3.17	232.7/0.56	316.69/0.31	558.08/1.17
14	297.78/14.83	128.06/1.93	562.76/59.26	516.21/3.2	247.34/0.51	335.16/0.39	595.76/1.0
15	339.6/17.44	129.35/1.76	622.44/55.37	598.5/4.36	267.48/0.42	353.99/0.61	626.45/1.2
16	316.58/16.43	134.44/1.92	590.64/59.21	583.14/4.17	281.4/0.4	375.06/0.51	668.04/1.92
17	341.56/19.64	136.79/3.15	645.76/54.75	647.26/5.56	285.07/0.25	362.9/0.33	646.33/1.91
18	348.33/19.58	139.63/2.87	659.74/56.5	667.77/4.74	289.9/0.33	391.02/0.32	683.24/1.82
19	323.1/17.9	136.65/2.8	670.77/59.15	686.18/5.43	295.14/0.3	398.73/0.41	690.21/1.78
20	349.9/20.07	138.37/3.03	691.55/59.81	696.88/4.55	300.29/0.26	382.61/0.31	675.36/1.98
21	327.75/18.28	141.32/3.71	697.4/57.13	726.02/5.88	306.03/0.3	411.22/0.42	713.64/2.96
22	362.94/20.96	147.91/3.59	747.97/51.78	745.08/6.88	297.27/0.24	418.72/0.35	721.51/2.34
23	335.23/18.61	133.13/3.1	761.49/54.18	756.19/6.4	319.08/0.3	423.09/0.33	731.2/2.12
24	368.84/21.04	148.82/3.8	747.06/62.13	782.73/6.38	326.15/0.46	415.52/0.37	703.49/1.88
25	342.18/20.24	151.22/3.9	759.98/62.38	789.63/5.66	330.77/0.35	440.55/0.52	748.4/2.06
26	377.04/21.64	154.43/5.52	775.73/61.01	807.08/7.16	317.95/0.26	442.36/0.62	755.31/2.91
27	373.08/22.37	158.89/4.68	790.87/56.52	829.75/7.19	319.88/0.3	454.64/0.49	763.09/2.83
28	379.17/22.73	159.22/4.94	791.4/56.59	764.99/5.55	346.89/0.43	461.02/0.51	773.71/2.61
29	385.09/22.41	162.46/5.19	773.54/54.27	852.23/8.08	328.89/0.51	462.91/0.85	739.66/3.02
30	375.11/22.21	163.66/5.28	814.76/61.65	878.6/7.01	338.05/0.41	472.48/0.74	753.83/2.82
31	383.7/24.29	143.27/6.56	794.0/40.09	877.26/7.58	359.43/0.54	458.92/0.61	794.8/4.04
32	376.85/22.17	158.19/6.48	809.91/57.29	886.49/22.34	366.15/0.4	481.3/1.85	801.34/3.08