# CS121@Fall2021 Lab 1
# parallel breadth-first-search on a shared memory architecture with OpenMP

Cheng Peng (彭程) 2020533068

November 27, 2021

# Contents

# 1 Introduction

## 1.1 backgrounds and related works

Breadth-first search (or BFS for short) is a ubiquitous graph algorithm which is used a building-block of various network analysis algorithms and state space search algorithms. The classic queue-based BFS implementation has a linear time and space complexity $\Theta(|V|+|E|)$, which is already optimal in a sequential setting, is not efficient enough for nowadays data mining tasks for example searching the social network and the world wide web. Great amount of research have been done to parallize the algorithm and a general framework of parallel BFS is developed, which is called level-synchronous is developed described in [BM14]. Further more, optimizations for the specific architecture of the machine or the characteristic of input graphs are made e.g. [YFG13] proposed a way to partition the adjcent matrix for better performance on a NUMA architecture, [LWH10] showed that BFS can be efficiently carried out on GPUs and [BAP12] revealed that a "bottom-up" approach out performs the tradition "top-down" one on graphs with low diameter.

## 1.2 lab task

In this lab, we designed and implemented a paralized BFS for undirected graphs on a shared memory architecture with OpenMP C++. We then run the program to inspect its performance and scalability.

# 2 Design and implementation of the algorithm

## 2.1 overall idea

BFS visit the vertices layer by layer, however, only one vertex is added into the frontier at one time. This can be parallized. Our algorithm finds all the vertices that belongs to the next layer concurrently.
The general idea can be showed by the following python-like pseudo code.

```
def BFS(source: Vertex, G: Graph) -> BFSTree:
   visited:set[Vertex] = {source} # mark for visted vertices
   parent:set[Vertex] = {} # parent in the BFS tree
   distance:map[Vertex,int] = {} # distance to the source
   frontier:set[Vertex] = {source} # the frontier, vertices in the same layer
   while not frontier.empty():
      next_layer:set[Vertex] = {} # vertices in the next layer
      parallel for u in frontier:
         for v in G.neighbour(u):
            if not visited:
               visited.add(v)
               parent[v] = u
               distance[v] = distance[u]+1
               next_layer.add(v)
      frontier = next_layer # vertices in the next layer now become the frontier
   return BFSTree(source, parent, distance)
```

## 2.2 implementation

- We use a bitset data structure for `vis` in order to reduce space cost and take advantage from the shared L3-cache on modern CPUs.

- Synchronization across all the working threads is needed after the for-loop. OpenMP do this automatically unless a `nowait` appears in the `omp for` directive.
- We might have race condition in the if statement, however it is harmless as long as the data written to `parent[v], distance[v]` does not corrupt.
- Efficient concurrent set e.g. binary serach tree or hash table with certain synchronization mechanism are hard to implement. What make things worse is that these data structures have bad memory access patterns which become potential bottleneck of our program due to the limited memory bandwidth and (or) latency.
  Therefore, an heap-allocated array e.g. `std::vector` is used, and we rely on the `vis` bitset to eliminate duplications.
- To allocate memory for `next_layer` and make sure that differnt worker threads write to differnt address, we use a "prefix-sum pre allocation". That is to say, we perform a prefix-sum $S(i) = S(i-1) + \mathrm{deg}(\mathrm{u_i})$ where $u_i$ is the $i$-th vertex in the frontier. The neighbours of $u_i$ that belongs to the next layer get writes to address between $S(i-1)+1$ to $S(i)$.
- To collect all the unvisted neighbours, the prefix-sum based parallel filter algorithm is employed.
- Since a vertex that belongs to the next layer might be connected to multiple vertex in the frontier, we might have duplicated elements in `next_layer`. If no duplication detection is applied, the number of duplications can grow exponentially in some cases e.g. on a 2D-grid. Therefore deduplication is necessary on each layer.
  We can use atomic test-and-set instruction to make sure that each vertex is added only once.

The implementation in cplusplus can be found in `src/parallel-bfs.cpp`.

# 3 Testing setup

## 3.1 hardware and software specification of the testing environment

The benchmarking is performed on a machine with dual-socket intel E5-2690 v4 CPUs (14 physics core each CPU, hyperthreading disabled) and 251GiB of memory. Ubuntu 18.04 is installed on that machine. The compiler used is gcc 7.5.0 and OpenMP version is 4.5.
For more detailed information, see `report/env.md`.

## 3.2 testcases

We run the program on 7 testcases. For each testcase, we select 20 random sources to perform BFS. The performance of our program is reported in MTEPS or millions of traveled edges per second that is to devide edges in the connected compoent containing the source by the running time in microseconds.

The first four testcases are from the Stanford network analysis project [LK14] and the latter three testcase are the R-MAT [KVG15] graphs of $10^8$ vertices and $10^9$ edges, with parameters $(a, b, c) = (0.3, 0.25, 0.25)$, $(a, b, c) = (0.45, 0.25, 0.15)$, and $(a, b, c) = (0.57, 0.19, 0.19)$ respectively.

The input file are in a matrix-market exchange coordinate format, see [ST] for detailed specification, where each undirected edge $(u, v)$ is presented twice i.e. both `u v 1` and `v u 1` should appear in the input.

# 4 Benchmark result

The performance is weird on ShanghaiTech AI cluster.
There might be too many tasks running on that node (ShanghaiTech AI cluster node13), while our program receives a quite low scheduling priority. Another possible reason is that, the memory on the node is saturated and our program's data structure is swaped to disk.
To verify that our program is correct and has consistent performance, we tested the program multiple times on a laptop with AMD ryzen 4800U processor and 16GB dual-channel RAM. The confusing performance drop down never appears.

The thread-MTESP plot is in the appendix section.
The best speed up we achieve is 3.539.
We use the linux kernel tool `perf`[Dev] to find the hotspot and bottleneck.
A great amount of frontend-stalled-cycles and backend-stalled-cycles is witnessed. We are convinced that this have caused the performance drop down.
The bottleneck of the whole program is the atomic test-and-set operation.

# 5 Further optimization

- Reduce atomic instructions.
- Implement a hybrid approach to explore the graph. This is described in [BAP12].
- Re-order the allocated memory to improve cache locality.

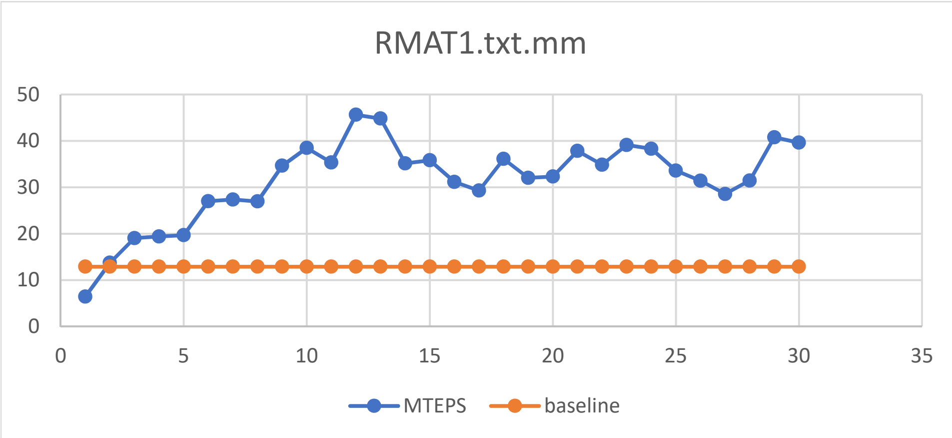
# A How to test the program

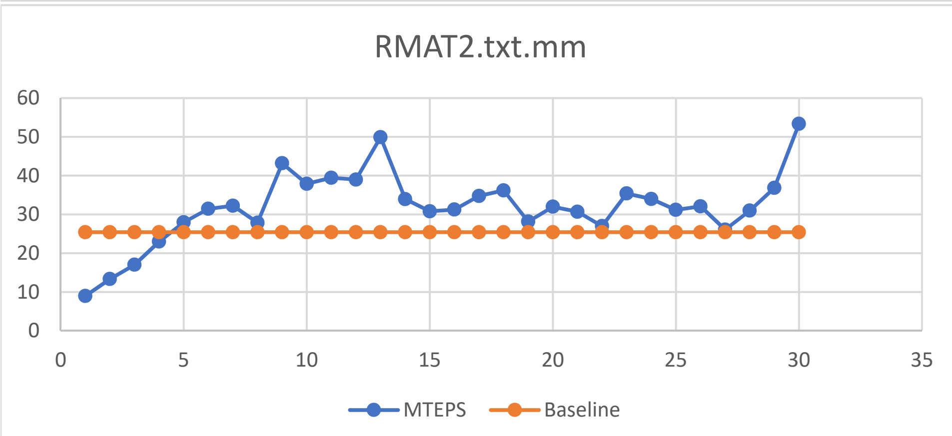
- To run all the testcases, use the all-in-one testing script `AIO.fish`.
- To run the program on a graph given by a matrix-market format, use command `bin/serial file.mm` or `bin/parallel file.mm`. This will run 20 BFS from randomly selected vertices and report the performance.
- Use `bin/serial file.mm source` and `bin/parallel file.mm source` to run only one BFS from the specified source.
  The source should be a integer ranging from 1 to $|V|$.

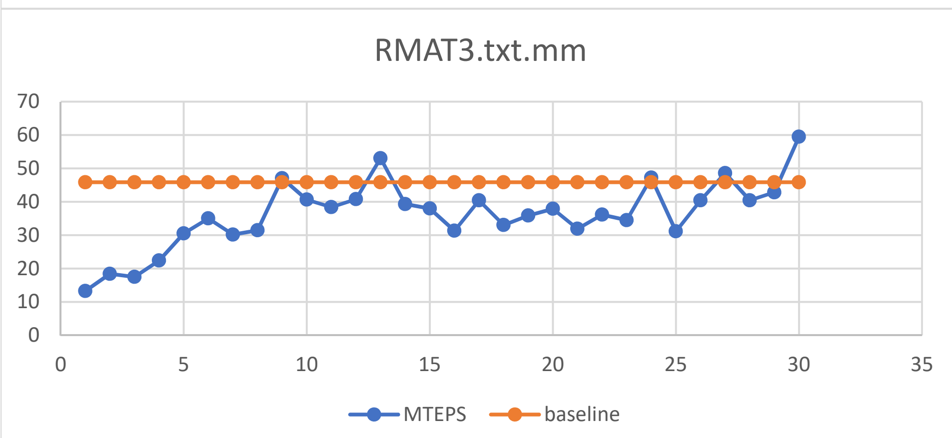
# B Typical benchmark results

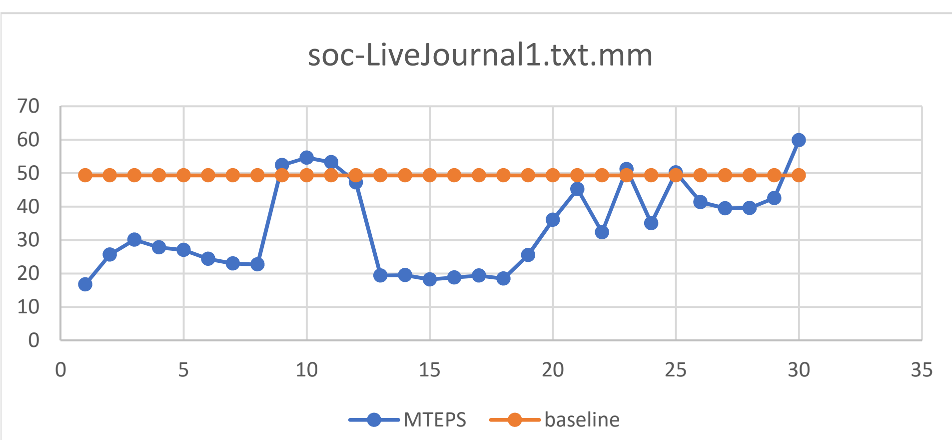
## B.1 on AI cluster

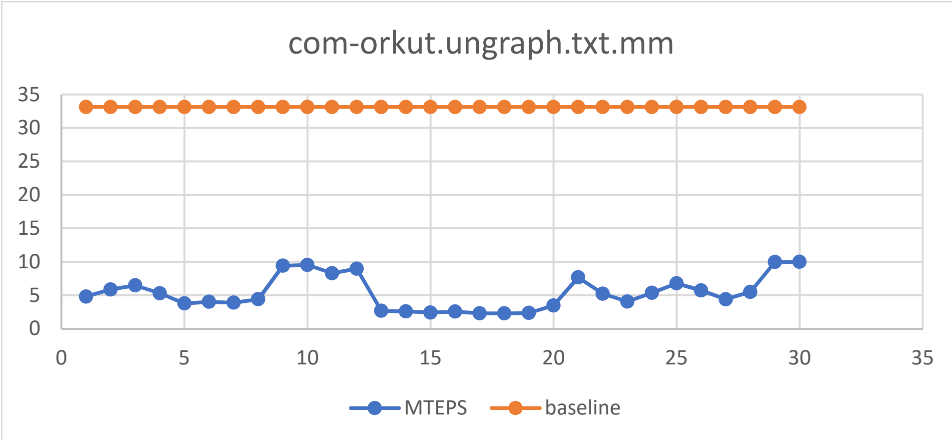
See the `report/record` directory.
We provide the performance data of the parallel algorithm when running on $1, 2 \ldots 30$ threads.

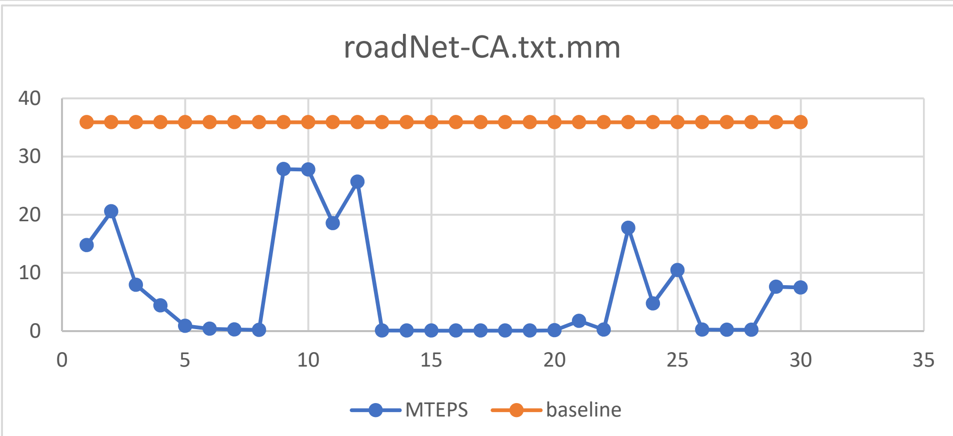

RMAT1.txt.mm
0
10
20
30
40
50
0
5
10
15
20
25
30
35
MTEPS
baseline

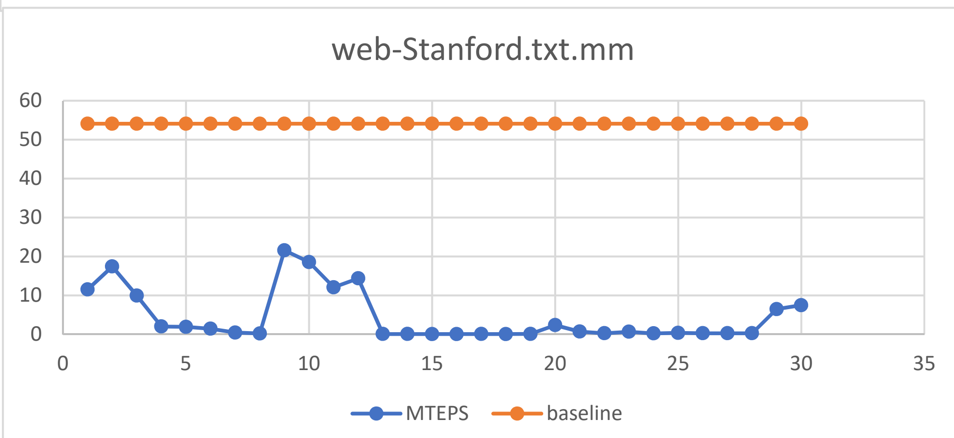

RMAT2.txt.mm
0
10
20
30
40
50
60
0
5
10
15
20
25
30
35
MTEPS
Baseline


RMAT3.txt.mm
0
10
20
30
40
50
60
70
0
5
10
15
20
25
30
35
MTEPS
baseline


soc-LiveJournal1.txt.mm
0
10
20
30
40
50
60
70
0
5
10
15
20
25
30
35
MTEPS
baseline



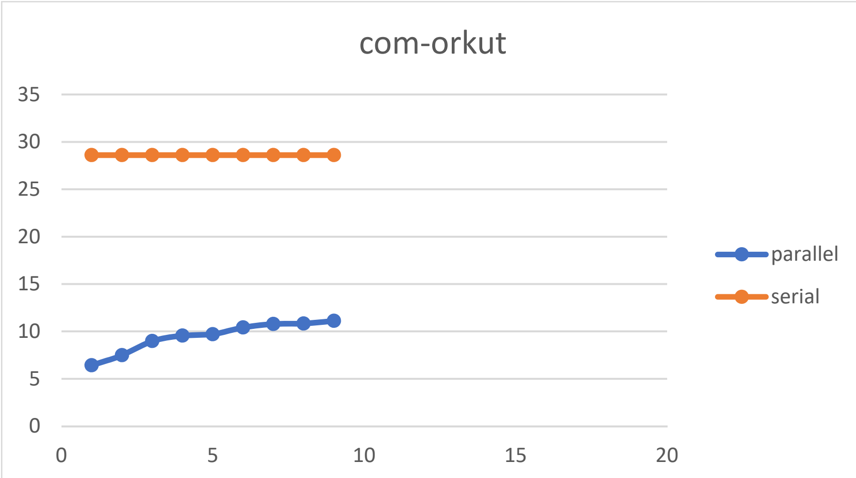



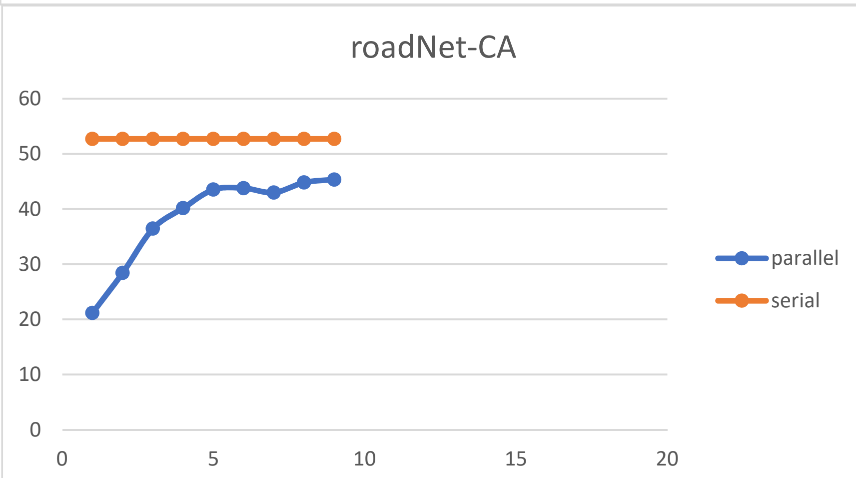

| | max | serial | speedup |
|---|---|---|---|
| RMAT1 | 45.627 | 12.892 | 3.539 |
| RMAT2 | 53.367 | 25.404 | 2.101 |
| RMAT3 | 59.505 | 45.851 | 1.298 |
| com-orkut.ungraph | 9.989 | 33.123 | 0.302 |
| roadNet-CA | 27.826 | 35.896 | 0.775 |
| soc-LiveJournal1 | 59.832 | 49.342 | 1.213 |
| web-Stanford | 21.574 | 54.094 | 0.399 |

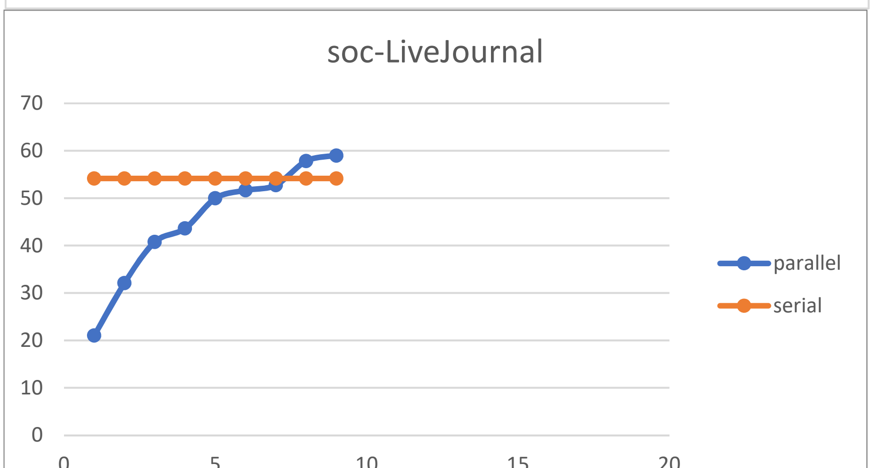
### B.2 on local machine

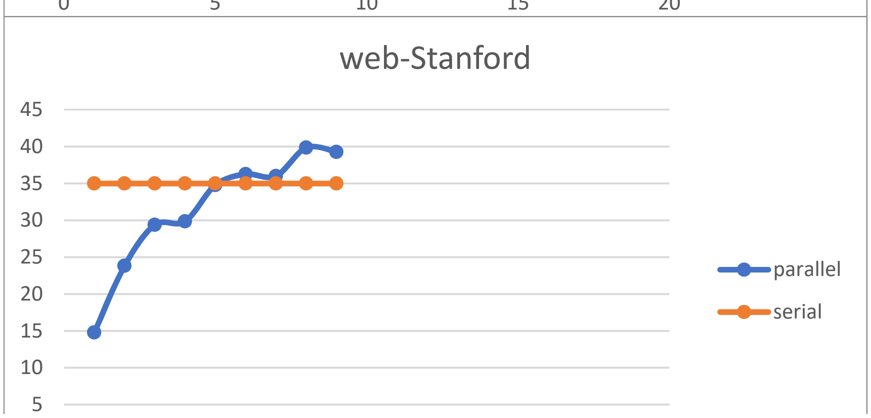
| com-orkut | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| MTEPS (parallel n) | 6.416519 | 7.494395 | 8.981975 | 9.551848 | 9.703494 | 10.40808 | 10.77909 | 10.83424 | 11.10628 |
| MTEPS (serial 1) | 28.59914 | 28.59914 | 28.59914 | 28.59914 | 28.59914 | 28.59914 | 28.59914 | 28.59914 | 28.59914 |
| R-MAT 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| MTEPS (parallel n) | 12.18286 | 17.20151 | 25.00471 | 30.00356 | 33.85331 | 36.35412 | 37.87842 | 40.69201 | 40.8063 |
| MTEPS (serial 1) | 21.04543 | 21.04543 | 21.04543 | 21.04543 | 21.04543 | 21.04543 | 21.04543 | 21.04543 | 21.04543 |
| R-MAT 2 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| MTEPS (parallel n) | 14.06986 | 20.28421 | 28.5422 | 32.63704 | 36.12809 | 38.99032 | 40.69752 | 43.48141 | 43.98102 |
| MTEPS (serial 1) | 27.46904 | 27.46904 | 27.46904 | 27.46904 | 27.46904 | 27.46904 | 27.46904 | 27.46904 | 27.46904 |
| R-MAT 3 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| MTEPS (parallel n) | 23.8466 | 28.12192 | 36.32763 | 40.3922 | 43.96515 | 46.01933 | 47.33822 | 49.61083 | 53.17687 |
| MTEPS (serial 1) | 59.89692 | 59.89692 | 59.89692 | 59.89692 | 59.89692 | 59.89692 | 59.89692 | 59.89692 | 59.89692 |
| roadNet-CA | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| MTEPS (parallel n) | 21.1615 | 28.44356 | 36.44944 | 40.1787 | 43.52251 | 43.78752 | 42.98373 | 44.81491 | 45.33251 |
| MTEPS (serial 1) | 52.71347 | 52.71347 | 52.71347 | 52.71347 | 52.71347 | 52.71347 | 52.71347 | 52.71347 | 52.71347 |
| soc-LiveJournal | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| MTEPS (parallel n) | 21.02555 | 32.09257 | 40.76807 | 43.6193 | 49.96813 | 51.65841 | 52.74756 | 57.83608 | 58.98332 |
| MTEPS (serial 1) | 54.15113 | 54.15113 | 54.15113 | 54.15113 | 54.15113 | 54.15113 | 54.15113 | 54.15113 | 54.15113 |
| web-Stanford | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| MTEPS (parallel n) | 14.77086 | 23.8107 | 29.36588 | 29.84305 | 34.76836 | 36.24929 | 35.99193 | 39.8431 | 39.25326 |
| MTEPS (serial 1) | 34.98102 | 34.98102 | 34.98102 | 34.98102 | 34.98102 | 34.98102 | 34.98102 | 34.98102 | 34.98102 |

| com-orkut | 10 | 11 | 12 | 13 | 14 | 15 | 16 | Max | SpeedUP |
|---|---|---|---|---|---|---|---|---|---|
| MTEPS (parallel n) | 11.2308 | 11.22263 | 11.27213 | 11.56447 | 11.49431 | 11.50793 | 11.57348 | 11.10628 | 0.388343 |
| MTEPS (serial 1) | 28.59914 | 28.59914 | 28.59914 | 28.59914 | 28.59914 | 28.59914 | 28.59914 | 28.59914 | |
| R-MAT 1 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | | |
| MTEPS (parallel n) | 41.34642 | 42.78034 | 43.25767 | 45.04466 | 45.57383 | 46.19662 | 46.58455 | 40.8063 | 1.938963 |
| MTEPS (serial 1) | 21.04543 | 21.04543 | 21.04543 | 21.04543 | 21.04543 | 21.04543 | 21.04543 | 21.04543 | |
| R-MAT 2 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | | |
| MTEPS (parallel n) | 44.40742 | 45.14268 | 46.12384 | 47.66229 | 47.9426 | 47.98571 | 48.4278 | 43.98102 | 1.601113 |
| MTEPS (serial 1) | 27.46904 | 27.46904 | 27.46904 | 27.46904 | 27.46904 | 27.46904 | 27.46904 | 27.46904 | |
| R-MAT 3 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | | |
| MTEPS (parallel n) | 53.15512 | 54.41378 | 55.13932 | 55.94138 | 57.5553 | 57.18925 | 56.87408 | 53.17687 | 0.887806 |
| MTEPS (serial 1) | 59.89692 | 59.89692 | 59.89692 | 59.89692 | 59.89692 | 59.89692 | 59.89692 | 59.89692 | |
| roadNet-CA | 10 | 11 | 12 | 13 | 14 | 15 | 16 | | |
| MTEPS (parallel n) | 47.00109 | 45.53705 | 47.23204 | 47.04309 | 47.54143 | 46.73739 | 46.87887 | 45.33251 | 0.85998 |
| MTEPS (serial 1) | 52.71347 | 52.71347 | 52.71347 | 52.71347 | 52.71347 | 52.71347 | 52.71347 | 52.71347 | |
| soc-LiveJournal | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 9 | |
| MTEPS (parallel n) | 59.48805 | 60.22419 | 61.85294 | 62.78445 | 62.82623 | 63.15883 | 63.51599 | 58.98332 | 1.089235 |
| MTEPS (serial 1) | 54.15113 | 54.15113 | 54.15113 | 54.15113 | 54.15113 | 54.15113 | 54.15113 | 54.15113 | |
| web-Stanford | 10 | 11 | 12 | 13 | 14 | 15 | 16 | | |
| MTEPS (parallel n) | 39.93772 | 40.64984 | 42.0481 | 42.57745 | 44.17612 | 44.70099 | 41.27974 | 39.8431 | 1.138992 |
| MTEPS (serial 1) | 34.98102 | 34.98102 | 34.98102 | 34.98102 | 34.98102 | 34.98102 | 34.98102 | 34.98102 | |


com-orkut
35
30
25
20
15
10
5
0
0
5
10
15
20
parallel
serial


roadNet-CA
60
50
40
30
20
10
0
0
5
10
15
20
parallel
serial


soc-LiveJournal
70
60
50
40
30
20
10
0
0
5
10
15
20
parallel
serial


web-Stanford
45
40
35
30
25
20
15
10
5
parallel
serial


0
0
5
10
15
20

## bibliography

[LWH10] Lijuan Luo, Martin Wong, and Wen-mei Hwu. "An effective GPU implementation of breadth-first search". In: *Design Automation Conference*. IEEE. 2010, pp. 52–55.

[BAP12] Scott Beamer, Krste Asanovic, and David Patterson. "Direction-optimizing breadth-first search". In: *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE. 2012, pp. 1–10.

[YFG13] Yuichiro Yasui, Katsuki Fujisawa, and Kazushige Goto. "NUMA-optimized parallel breadth-first search on multicore single-node system". In: *2013 IEEE International Conference on Big Data*. IEEE. 2013, pp. 394–402.

[BM14] Rudolf Berrendorf and Mathias Makulla. "Level-synchronous parallel breadth-first search algorithms for multicore and multiprocessor systems". In: (2014).

[LK14] Jure Leskovec and Andrej Krevl. *SNAP Datasets: Stanford Large Network Dataset Collection*. `http://snap.stanford.edu/data`. June 2014.

[KVG15] Farzad Khorasani, Keval Vora, and Rajiv Gupta. *Parmat: A parallel generator for large r-mat graphs*. 2015. URL: `https://github.com/farkhor/PaRMAT`.

[Dev] Linux Kernel Developers. *linux profiling tool*. URL: `https://perf.wiki.kernel.org/index.php/Main_Page`.

[ST] National Institute of Standards and Technology. *Matrix Market Exchange Formats*. URL: `https://math.nist.gov/MatrixMarket/formats.html`.