Abhijjith Venkkateshraj (av36677)

Parallel Computing – Final Project Report

Parallel Implementation of Breadth First Search (BFS) using OpenMP and comparison across multiple large graph workloads

Contents

Abstract

Data-intensive, graph-based computations are pervasive in several scientific applications and are known to be quite challenging to implement on shared memory and distributed memory systems.

In this project, I will explore the design space of parallel algorithms for ***Breadth-First Search (BFS)***, a key subroutine in several graph algorithms. I will present a highly tuned parallel approach for BFS on large parallel systems: based on shared memory using OpenMP. This approach will be compared with the Sequential BFS approach across multiple large graph workloads and the necessary results are analyzed.

Specifically the workloads that are targeted are taken from https://snap.stanford.edu/data/

Introduction

The use of graph abstractions to analyze and understand social interaction data, complex engineered systems such as the power grid and the Internet, communication data such as email and phone networks, biological systems, and in general, various forms of relational data, has been gaining ever-increasing importance. Common graph-theoretic problems arising in these application areas include identifying and ranking important entities, detecting anomalous patterns or sudden changes in networks, finding tightly interconnected clusters of entities, and so on. The solutions to these problems typically involve classical algorithms for problems such as finding spanning trees, shortest paths, biconnected components, matchings, flow-based computations, in these graphs. To cater to the graph-theoretic analyses demands of emerging “big data” applications, it is essential that we speed up the underlying graph problems on current parallel systems.

A traversal refers to a systematic method of exploring all the vertices and edges in a graph. The ordering of vertices using a “breadth-first” search (BFS) is of particular interest in many graph problems. Theoretical analysis in the random-access machine (RAM) model of computation indicates that the computational work performed by an efficient BFS algorithm would scale linearly with the number of vertices and edges. However, efficient RAM algorithms do not easily translate into “good performance” on current computing platforms. This mismatch arises since current architectures lean towards efficient execution of regular computations with low memory footprints, and heavily penalize memory-intensive codes with irregular memory accesses. Graph traversal problems such as BFS are predominantly memory access-bound, and these accesses are further dependent on the structure of the input graph, thereby making the algorithms “irregular”.

BFS on distributed-memory systems involves explicit communication between processors, and the distribution (or partitioning) of the graph among processors also impacts performance. We utilize a testbed of large-scale graphs (https://snap.stanford.edu/data/) with thousands of vertices and edges to empirically evaluate the performance of our BFS algorithms. These graphs are all sparse, i.e., the number of edges m is just a constant factor times the number of vertices n.

Implementation

Sequential BFS - Frontier based algorithm

We have a C++ based implementation of the Sequential BFS.

The graph class is defined as the following

```
class Graph {
  long int numVertices;
  long int numEdges;
  list<long int>* adjLists;
  bool* visited;

   public:
  Graph();
  void addEdge(long int src, long int dest);
  void BFS(long int startVertex);
};
```

The constructor of the graph is as follows.

- It takes a large graph dataset as input ‘data.txt’ which contains the information related to the number of vertices, number of edges and the connection between the vertices.
- An adjacency list is created and maintained for each vertex and is updated accordingly.

```
// Create a graph with given vertices,
// and maintain an adjacency list
Graph::Graph()
{
  ifstream infile("data.txt");
  if (infile >> numVertices >> numEdges)
  {
    printf("Num vertices = %ld, Num edges =  %ld\n", numVertices, numEdges);
    adjLists = new list<long int>[numVertices];
    // populate the graph with edges
    long int src_vertex, dst_vertex;
    while (infile >> src_vertex >> dst_vertex)
    {
       addEdge(src_vertex, dst_vertex);
    }
  }
}
```

The addEdge function is as follows

```
// Add edges to the graph
void Graph::addEdge(long int src, long int dest) {
  adjLists[src].push_back(dest);
  adjLists[dest].push_back(src);
}
```

The sequential BFS algorithm works as the following

- We have two lists (frontier and next_frontier) to keep track of vertices at the current and the next level
- We allocate an array ‘visited’ to keep track of all visited vertices.
- Mark the visited of all vertices as false
- Make the startVertex as visited
- Until the frontier list is not empty, do the following
    - Iterate through all the elements of the frontier
        - Get an element from the front of the frontier list
        - Look at all the neighbours of the vertex obtained from the frontier list
            - If that neighbour has not been visited before, make that neighbour visited and add that element to the ‘next frontier’ list.
    - Assign ‘next_frontier’ to ‘frontier’
    - Reset the next frontier list
    - Increment the level

```
// BFS algorithm
void Graph::BFS(long int startVertex)
{
  /* To keep track of vertices at the current level */
  list<long int> frontier;
  /* To keep track of vertices at the next level */
  list<long int> next_frontier;

  /* Allocate an array to keep track of visited vertices */
  visited = new bool[numVertices];

  /* Mark the visited of all vertices as false */
  for (long int i = 0; i < numVertices; i++)
  {
    visited[i] = false;
  }

  /* Make the startVertex as visited */
  visited[startVertex] = true;
  frontier.push_back(startVertex);

  /* Until the frontier list is not empty */
```

```
  while (!frontier.empty())
  {
    for(long int i = 0; i < frontier.size(); i++)
    {
        /* Get an element from the front of the fronteir list */
        list<long int>::iterator it = frontier.begin();
        advance(it, i);
        long int currVertex = *it;
        //cout << "Visited " << currVertex << " ";

        /* Look at all the neighbours of the vertex obtained from the frontier
list */
        for (list<long int>::iterator iter = adjLists[currVertex].begin(); \
                      iter != adjLists[currVertex].end(); iter++)
        {
          long int adjVertex = *iter;
          /* If that neighbour has not been visited before */
          if (visited[adjVertex] == false)
          {
            /* Make that neighbour visited */
            visited[adjVertex] = true;
            /* Push it to the next frontier */
            next_frontier.push_back(adjVertex);
          }
        }
    }

    //print(next_frontier);
    /* Assign the neighbours collected in this iteration to be used as source in
next iteration */
    frontier = next_frontier;
    /* Reset the next frontier list */
    next_frontier.clear();
    /* Increment the level */
    num_levels++;
  }
}
```

Possibilities for Parallelization in the above sequential code

- We have two lists (frontier and next_frontier) to keep track of vertices at the current and the next level
- We allocate an array 'visited' to keep track of all visited vertices.
- **Mark the visited of all vertices as false** – *can be executed in parallel (different threads can work on different vertices)*
- Make the startVertex as visited
- Until the frontier list is not empty, do the following
    - **Iterate through all the elements of the frontier** – *can be executed in parallel (different threads can work on different vertices)*
        - Get an element from the front of the frontier list
        - Look at all the neighbours of the vertex obtained from the frontier list
            - **If that neighbour has not been visited before, make that neighbour visited and add that element to the 'next frontier' list. –** *possible race conditions. If there are two threads examining 2 different vertices but has a common next neighbour, then it needs to be handled and taken care accordingly. Same vertex may get added multiple times to the 'next_frontier'.*
    - **Assign 'next_frontier' to 'frontier' –** *We need a barrier before this statement. Only after all the threads have finished processing their neighbours, we can go ahead and execute this statement. Also, we need to update this in a critical region.*
    - Reset the next frontier list
    - Increment the level

Parallel BFS using OpenMP (shared memory)

The major updates done in the BFS algorithm in order to support shared memory parallelism among threads are as follows:

- Parallelize the marking of all vertices as false using the **'omp parallel for'** construct. This worksharing construct splits the loop iterations among the threads in a uniform manner.

```
/* Mark the visited of all vertices as false */
#pragma omp parallel for
for (long int i = 0; i < numVertices; i++)
{
  visited[i] = false;
}
```

- Until the frontier list is not empty, introduce a **omp parallel construct** and get the thread ID and the num threads using the Open MP functions.

```
-    /* Until the frontier list is not empty */
-    while (!frontier.empty())
-    {
-  #pragma omp parallel
-  {
-      long int tid = omp_get_thread_num();
-      long int num_threads = omp_get_num_threads();
-      /* To keep track of vertices at the next level */
-      list<long int> local_next_frontier;
-
```

- For each thread, iterate across all the frontier elements in jumps of 'num_threads'. This ensures that **each thread has a uniform workload to perform and that the frontier elements processing are totally independent** – thereby increasing parallelism.

```
-      for(long int i = tid; i < frontier.size(); i += num_threads)
-      {
-          /* Get an element from the front of the fronteir list */
-          list<long int>::iterator it = frontier.begin();
-          advance(it, i);
-          long int currVertex = *it;
-          //printf("Thread %ld, Visited %ld\n", tid, currVertex);
```

- In order to avoid race conditions when we are updating the visited[adjVertex] as true and pushing that adjVertex neighbor into the 'next_frontier' list, we make use of the **'omp atomic capture'** region. This construct allows access of a specific memory location atomically. It ensures that race conditions are avoided through direct control of concurrent threads that might read or

write to or from the memory location. The presence of the capture clause indicates updating of a variable while capturing the original or final value of the variable atomically.

```
        /* Look at all the neighbours of the vertex obtained from the frontier
list */
        for (list<long int>::iterator iter = adjLists[currVertex].begin(); \
                                    iter != adjLists[currVertex].end(); iter++)
        {
            long int adjVertex = *iter;
            /* If that neighbour has not been visited before */
            if (visited[adjVertex] == false)
            {
                bool insert = false;
                #pragma omp atomic capture
                {
                    insert = visited[adjVertex];
                    /* Make that neighbour visited */
                    visited[adjVertex] = true;
                }
                if(insert == false)
                {
                    /* Push it to the next frontier */
                    local_next_frontier.push_back(adjVertex);
                }
            }
        }
```

- Finally, we make use of the omp critical pragma in order to merge all the 'local_next_frontier' elements into the global 'next_frontier' list.

```
    #pragma omp critical
    {
        next_frontier.merge(local_next_frontier);
    }
```

The complete working source code of the Parallel BFS using OpenMP is shown below

```
// BFS algorithm
void Graph::BFS(long int startVertex)
{
  /* To keep track of vertices at the current level */
  list<long int> frontier;
  /* To keep track of vertices at the next level */
  list<long int> next_frontier;
```

```
  /* Allocate an array to keep track of visited vertices */
  visited = new bool[numVertices];

  /* Mark the visited of all vertices as false */
  #pragma omp parallel for
  for (long int i = 0; i < numVertices; i++)
  {
    visited[i] = false;
  }

  /* Make the startVertex as visited */
  visited[startVertex] = true;
  frontier.push_back(startVertex);

  /* Until the frontier list is not empty */
  while (!frontier.empty())
  {
#pragma omp parallel
{
    long int tid = omp_get_thread_num();
    long int num_threads = omp_get_num_threads();
    /* To keep track of vertices at the next level */
    list<long int> local_next_frontier;

    for(long int i = tid; i < frontier.size(); i += num_threads)
    {
        /* Get an element from the front of the fronteir list */
        list<long int>::iterator it = frontier.begin();
        advance(it, i);
        long int currVertex = *it;
        //printf("Thread %ld, Visited %ld\n", tid, currVertex);
        /* Look at all the neighbours of the vertex obtained from the frontier
list */
        for (list<long int>::iterator iter = adjLists[currVertex].begin(); \
                                iter != adjLists[currVertex].end(); iter++)
        {
            long int adjVertex = *iter;
            /* If that neighbour has not been visited before */
            if (visited[adjVertex] == false)
            {
                bool insert = false;
                #pragma omp atomic capture
                {
                    insert = visited[adjVertex];
                    /* Make that neighbour visited */
```

```
                    visited[adjVertex] = true;
                }
                if(insert == false)
                {
                    /* Push it to the next frontier */
                    local_next_frontier.push_back(adjVertex);
                }
            }
        }
    }

    //print(local_next_frontier);
    #pragma omp critical
    {
        next_frontier.merge(local_next_frontier);
    }
}

    //print(next_frontier);
    /* Assign the neighbours collected in this iteration to be used as source in
next iteration */
    frontier = next_frontier;
    /* Reset the next frontier list */
    next_frontier.clear();
    /* Increment the level */
    num_levels++;
  }
}
```

Experiments and Results

We have run the sequential and the parallel versions of the BFS across the following datasets

Graph description	Type	Nodes	Edges
Social Circles - Facebook	Undirected	4039	88234
Communication networks - Enron email network	Undirected	36692	367662
Gnutella peer-to-peer network, August 25 2002	Directed	22687	54705

Command used to compile : icc -O2 -qopenmp -xHost ParallelBFS_OpenMP.c -o ParallelBFS_OpenMP

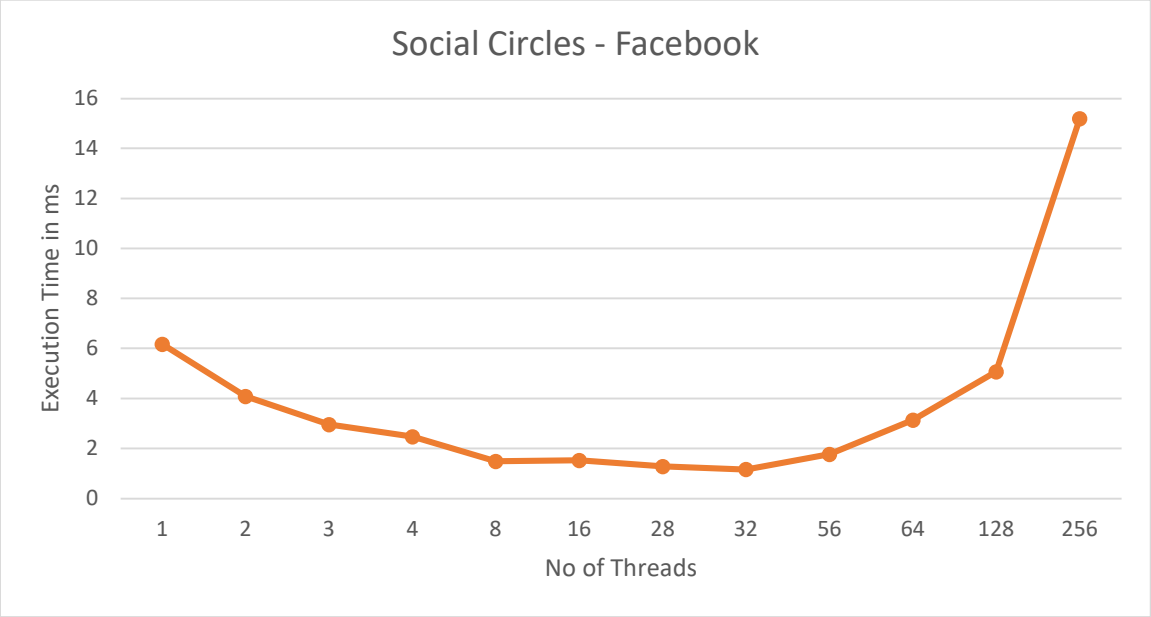
Command used to run: numactl --cpunodebind=0 --preferred=0 ./ParallelBFS_OpenMP

		Time in milliseconds	
No of Threads	**Social Circles - Facebook**	**Email Enron**	**p2p-Gnutella25**
1	6.166629	732.57836	219.169671
2	4.080912	370.058956	111.711105
3	2.965318	248.39853	75.147915
4	2.469523	185.931526	56.831563
8	1.489493	95.228941	29.2802
16	1.523512	51.888403	18.025255
28	1.287337	45.759681	24.285266
32	1.16196	66.477806	17.142152
56	1.775028	61.428326	20.671165
64	3.137911	76.733586	47.187626
128	5.071462	97.910846	53.792964
256	15.194762	130.936655	59.633547

Execution Time Comparisons

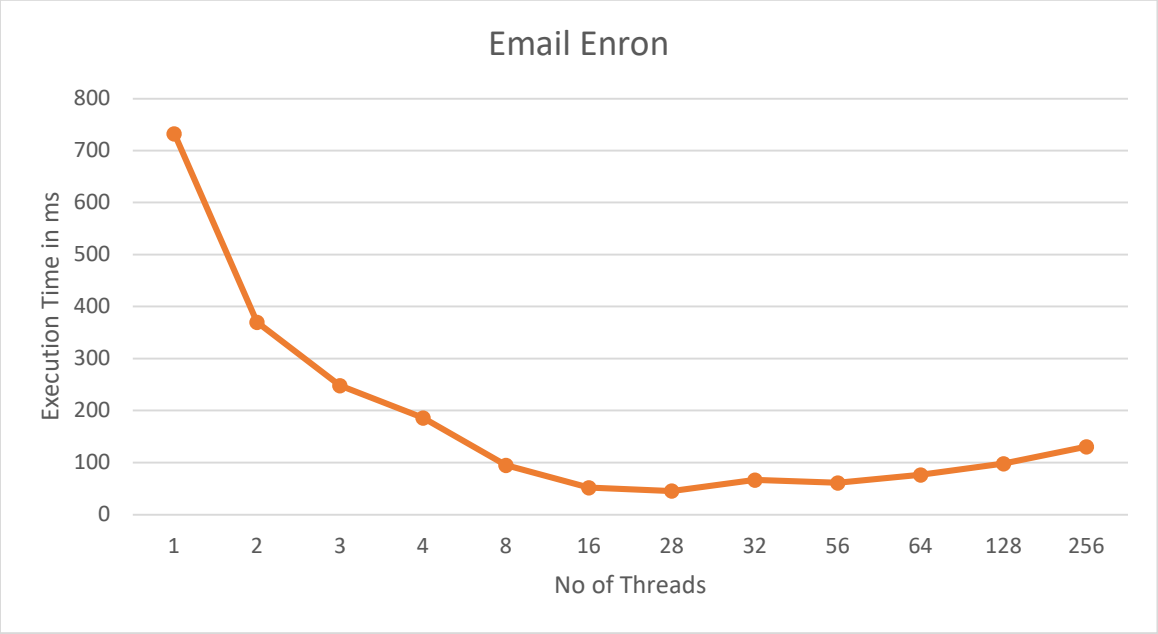
Social Circles

Serial Execution Time: 11.753339 ms



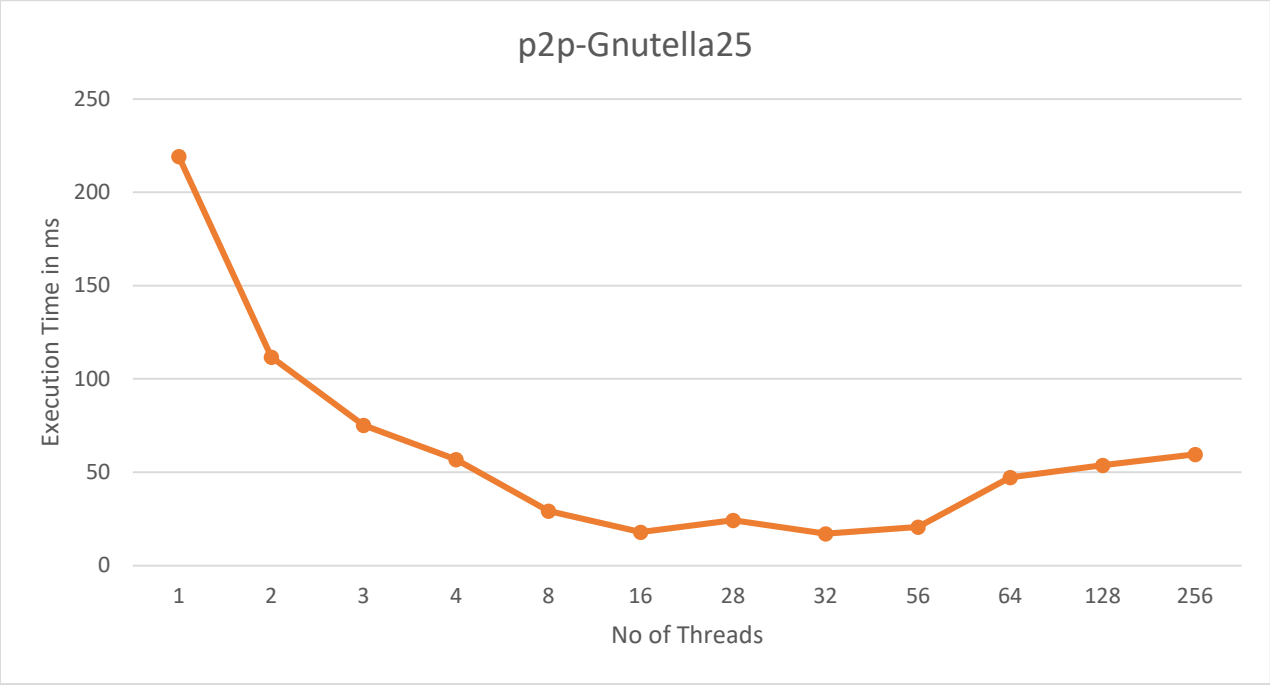
Email Enron

Serial Execution Time: 871.864181 ms



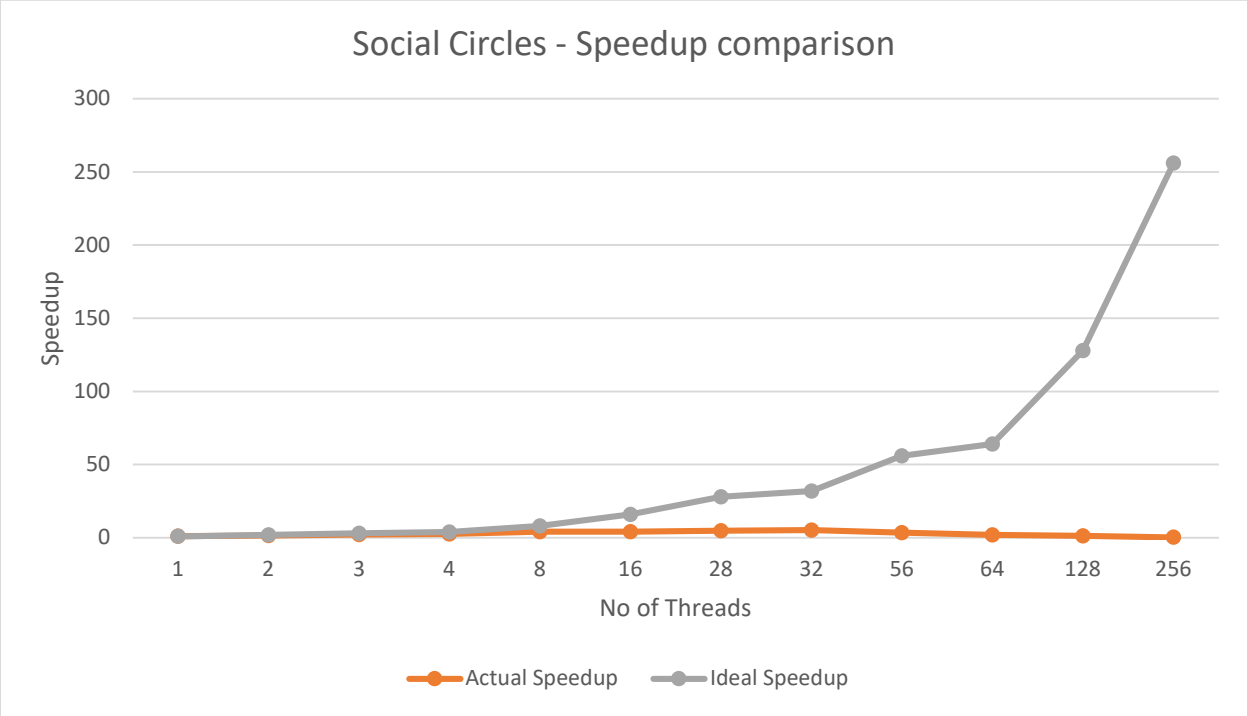
P2p-Guntella25

Serial Execution Time: 265.089673 ms

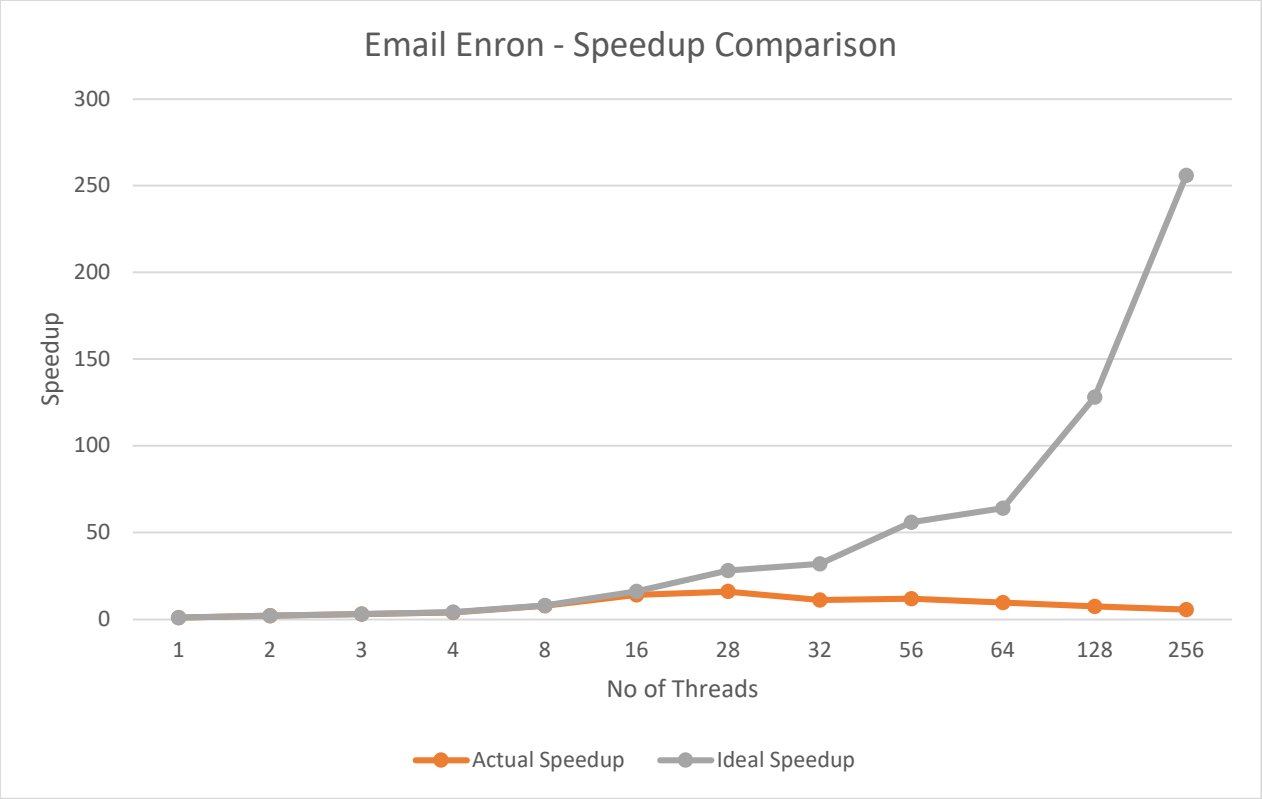


Speedup Comparisons

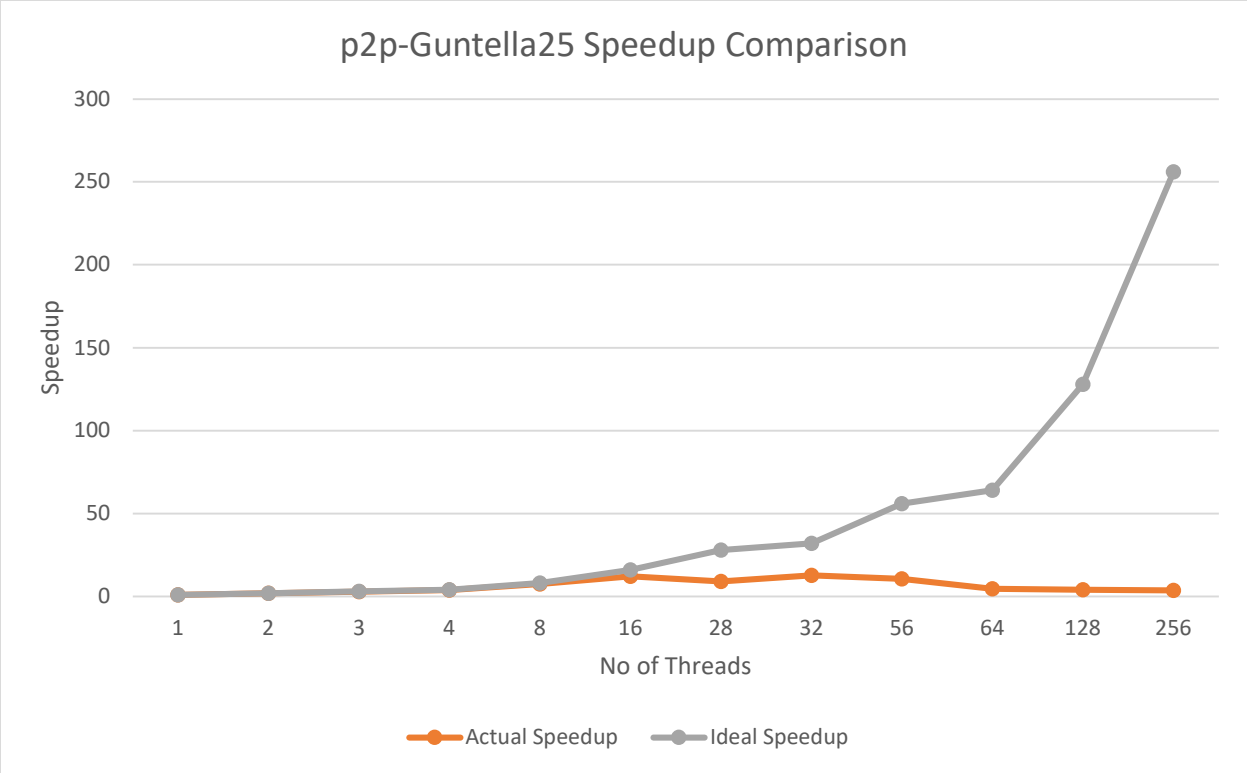
No of Threads	Social Circles - Facebook	Actual Speedup	Ideal Speedup
1	6.166629	1	1
2	4.080912	1.511090903	2
3	2.965318	2.079584382	3
4	2.469523	2.497093163	4
8	1.489493	4.140085922	8
16	1.523512	4.047640583	16
28	1.287337	4.790221209	28
32	1.16196	5.307092327	32
56	1.775028	3.47410238	56
64	3.137911	1.965202009	64
128	5.071462	1.215946999	128
256	15.194762	0.405839131	256



No of Threads	Email Enron	Actual Speedup	Ideal Speedup
1	732.57836	1	1
2	370.058956	1.979626079	2
3	248.39853	2.949205698	3
4	185.931526	3.94004382	4
8	95.228941	7.692812209	8
16	51.888403	14.11834471	16
28	45.759681	16.00925409	28
32	66.477806	11.01989377	32
56	61.428326	11.92574188	56
64	76.733586	9.547036678	64
128	97.910846	7.48209611	128
256	130.936655	5.594906636	256



No of Threads	p2p-Gnutella25	Actual Speedup	Ideal Speedup
1	219.169671	1	1
2	111.711105	1.96193271	2
3	75.147915	2.916510338	3
4	56.831563	3.856477975	4
8	29.2802	7.485251843	8
16	18.025255	12.15903303	16
28	24.285266	9.024800099	28
32	17.142152	12.78542338	32
56	20.671165	10.60267629	56
64	47.187626	4.644642877	64
128	53.792964	4.074318548	128
256	59.633547	3.675274774	256



Observations

- In the case of parallel execution, there are significant overheads associated with the Creation (fork) and deletion of threads at the start and end of the parallel regions. The compiler optimization O2 performs vectorization of loops in the serial case. In case of parallel with one thread, the presence of the “omp parallel” construct might restrict the loop vectorization.

- From the plots shown above, the actual speedup majorly deviates from the ideal speedup starting from Number of threads = 28. This might be since the number of cores per socket is 28. There are various reasons such as different kinds of overhead, communication costs between cores and sockets, load imbalance that attributes to the irregular speedup as compared to the ideal speedup for number of threads greater than 28. Since there are 2 sockets in total, the total number of active cores = 2*28 = 56. Hence, for anything beyond 56 threads, we experience saturation in the speedup and there are no gains associated accordingly. Out of the combinations given in the question, number of threads = 32, performs the best.

```
c206-008[clx](1011)$ lscpu
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                56
On-line CPU(s) list:   0-55
Thread(s) per core:    1
Core(s) per socket:    28
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 85
Model name:            Intel(R) Xeon(R) Platinum 8280 CPU @ 2.70GHz
Stepping:              7
CPU MHz:               3299.853
CPU max MHz:           4000.0000
CPU min MHz:           1000.0000
BogoMIPS:              5400.00
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              1024K
L3 cache:              39424K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34,36,38,40,42,44,46,48,50,52,54
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35,37,39,41,43,45,47,49,51,53,55
Flags:                 fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush
constant_tsc art arch_perfmon pebs bts rep_good nopl xtopology nonstop_tsc aperfmperf eagerfpu pni pclm
pdcm pcid dca sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand lahf_lm ab
ssbd mba ibrs ibpb stibp ibrs_enhanced tpr_shadow vnmi flexpriority ept vpid fsgsbase tsc_adjust bmi1 h
eed adx smap clflushopt clwb avx512cd avx512bw avx512vl xsaveopt xsavec xgetbv1 cqm_llc cqm_occup_llc c
nni md_clear spec_ctrl intel_stibp flush_l1d arch_capabilities
c206-008[clx](1012)$
```

Regarding Scaling

- Scaling is good until the point where we increase the number of threads = number of cores available in the system. Until this point, the speedup will increase. Post this point, increasing the number of threads will reach a point of diminishing returns. There will be context switches between the threads, and it leads to considerable overhead saturating or decreasing the speedup. The serial portion of the code will also be a bottleneck. (It cannot be parallelized irrespective of the number of the cores in the system). In the above plot, we can see that there is no advantage in speedup when increasing thread count beyond 32.

Regarding Scheduling (Keeping the number of threads as 4)

Chunk size	Static	Dynamic
100	73.305709	73.575226
1000	73.494781	73.608724
10000	73.571963	73.724385
100000	73.790149	73.881345

- From the above table, we can see that the static scheduling has a very slight edge over the dynamic scheduling type. Since there is only a small for loop where this work-

sharing scheduling is applied, the time differences between all the combinations are negligible.

- With static schedules, the iterations are assigned purely based on the number of iterations, the number of threads and the chunk parameter. In dynamic schedules, on the other hand, iterations are assigned to threads that are unoccupied. Dynamic schedules are a good idea if iterations take an unpredictable amount of time, so that load balancing is needed.
- With static scheduling, the compiler will determine the assignment of loop iterations to the threads at compile time, so, provided the iterations take roughly the same amount of time, this is the most efficient at runtime.
- In dynamic scheduling, blocks of iterations are put in a task queue, and the threads take one of these tasks whenever they are finished with the previous. While this schedule may give good load balancing if the iterations take very differing amounts of time to execute, it does carry runtime overhead for managing the queue of iteration tasks. The choice of a chunk size is often a balance between the low overhead of having only a few chunks, versus the load balancing effect of having smaller chunks.
- We can see that the runtimes are most efficient when the chunk size is 100. Increasing the chunk size beyond this is leading to increased runtimes as the chunk size assigned to each thread is more and it requires the current thread to execute the entire chunk size before another thread can start executing.

<u>Regarding numactl settings in the command</u>

Mode	Execution Time (in ms)	Where memory is allocated	Where threads are running
No Numactl	4.458092	Any node that's free	Any node that's free
Preferred = 0	2.570406	Node 0 mostly	Node 0 mostly
Interleave-0,1	2.636461	Node 0 and Node 1 in a round robin way	Node 0 mostly
Preferred = 1	2.778386	Node 1 mostly	Node 0 mostly

- 'numactl' gives the ability to choose core on which we want to execute tasks on and the ability to choose where we want to allocate data. Two policies : NUMA scheduling policy and NUMA memory placement policy.
- --cpunodebind=nodes, -N nodes meaning only execute a command on the CPUs of the specified nodes. Nodes may consist of several CPUs.
- --preferred=node specifies that we prefer it be allocated on the specified node is possible, otherwise fall back to other nodes.

- We see an increasing trend in the table. This can be explained due to the following: The cpunodebind parameter is given as 0. (Indicates that the processes will be bound mostly to the Node 0 if it has available cores).
    - Consider preferred=0: In this case, the memory will be allocated (eg: dynamic allocation) preferably in the Node 0. There will be less movement since the

memory and the processes execution are in the same node. Hence, this will have the least time.

- Consider interleave=0,1: In this case, it sets a memory interleave policy whereby memory will be allocated using a round robin mechanism on the nodes and when it cannot be allocated on the current interleave, target falls back to other nodes. So it's approximately a 50:50 share between both the nodes. Hence, the memory might be allocated to Node 1 and there will be some communication/movement across the two Nodes.
- Consider preferred=1: In this case, the memory will be allocated (eg: dynamic allocation) preferably in the Node 1. There will be more movement since the memory and the processes execution are in the different nodes. Hence, this will have the maximum time.

Conclusion

In this project, we have successfully explored the design space of parallel algorithms for ***Breadth-First Search (BFS)***, a key subroutine in several graph algorithms. We have also presented a highly tuned parallel approach for BFS on large parallel systems: based on shared memory using OpenMP. This approach was compared with the Sequential BFS approach across multiple large graph workloads and the necessary results are analyzed.