Assignment 2 - Parallel BFS

Carlos Vega de Alba

INTRODUCTION

In the project we will be studying how parallelization of the breadth first search algorithm can improve the performance.
We will first develop a fully parallel version and afterwards we will study how an alternative mixture of parallel and sequential algorithm could improve the runtimes.
To be able to analyze the runtimes we will be using 4 big different graphs that have millions of vertices and edges.

INDEX

1. Parallel BFS

To develop the parallel breadth first search we will distribute the vertices among the threads in each layer. To make sure the results are correct we will not go any deeper in the graph until all threads have finished their own round

-We would use a local array for each thread that will store the new vertices discovered by itself for each round.

```
int *local_T;       // Local array to store the new vertices locally
int local_counter=0; // Counter of where to write

local_T = (int *) malloc(sizeof(int)*(n+2)); // Locate memory
if(!local_T){
    fprintf(stderr, "Memory allocation failed\n");
    return;
}
```

-Then we would make only one thread to prepare the global variables and set all the vertices to unvisited.

```
#pragma omp single
    {
T[0] = 1; // Num Vertices of this round
S[0] = 1; // Vertices of this round

for(i=1;i<=n;i++) {   // Set that every node is unvisited
    p[i] = -1;        // Using -1 to mark that a vertex is unvisited
    dist[i] = -1;
}

p[1] = 1;       // Set the parent of starting vertex to itself
dist[1] = 0;    // Set the distance from the starting vertex to itself
    }

#pragma omp barrier  // Wait before everything is ready
```

This could be done in parallel

-After this we are ready to begin exploring the graph. While there are still vertices to explore, distribute them through the threads and make every thread discover their regions locally.

```
  while(T[0]!= 0){
        T[offset+thread_id]=0; // Number of discovered vertices of this
thread for this round
        local_counter=0;       // Also store them locally

        //Start the searching in parallel
        #pragma omp for no wait
        for(i=0; i<T[0]; i++){
           v = S[i]; // The value of i is unique for each thread
           for(j=ver[v]; j<ver[v+1]; j++) {  // Go through the neighbors of v
              w = edges[j];                 // Get next neighbor w of v
              if (p[w] == -1) {             // Check if w is undiscovered
                 p[w] = v;                 // Set v as the parent of w
                 dist[w] = dist[v]+1;       // Set distance of w
                 local_T[local_counter]= w; // Add w to local_T and increase
number of vertices discovered
                 local_counter++;          // Increase the counter, in the shared
value and the local value
              }
           }
        }
        T[offset+thread_id] = local_counter;
        #pragma omp barrier  // Wait for all threads to finish this round
before moving to the next one
```

-Then we will need to prepare where each vertex should start writing in the global S variable, and then we can write in it in parallel as they will all write in different positions of the array.

```
        // Prepare the index where each vertex should write in S, it can be done in parallel
    // For example thread_0 writes at 0, thread_1 writes where thread_0 finishes...
    int start = 0; // Get the first amount of the first thread

    // Gets where do start writing in the memory, finishes when knowing where to start
    // to do not work more than needed
    for(i=0; i<thread_id; i++){
        start += T[offset+i];
    }

    // Copy to the global array in parallel
    for(i=0; i<local_counter; i++){
        S[start+i]= local_T[i];
    }

    if(thread_id == (num_threads-1)){ // only one vertex sets the total amount of threads
        T[0] = start+local_counter; //Set the amount of new vertices for next round
    }

    #pragma omp barrier // Wait until every thread has finished copying to S before moving to the next round
```

could have used a memcpy() here

- **Experiments Sequential vs Parallel**

  ➔Configurations:

  4 different graphs, 3 times ran each, 12 different number of threads (1, 6, 10, 14, 18, 20, 24, 28 32, 36, 40, 44)

  The results file had this output

```
x = [1 10 20 30 40 50 60 70 80 ];
name = {'/export/data/fredrikm/inf236/graphs/road_usa'
,'/export/data/fredrikm/inf236/graphs/delaunay_n24'
,'/export/data/fredrikm/inf236/graphs/hugebubbles-00020'
,'/export/data/fredrikm/inf236/graphs/rgg_n_2_22_s0' };
n = [23947347 16777216 21198119 4194304 ];
nz = [28854312 50331601 31790179 30359198 ];
TimeSequentialBFS = [1.674710 1.445387 3.639111 0.960999 ];
TimeParallelBFS = [1.744540 1.652610 1.916001 1.836442 1.758964
1.720948 1.832399 1.756774 2.111397 1.392295 0.382614 0.880357
0.959598 0.918394 0.933944 0.932108 1.042839 0.925943 3.609981
2.503040 1.975996 2.070375 1.602369 1.678707 1.743181 1.609793
1.771887 0.970593 0.475675 0.412278 0.492328 0.492713 0.497068
0.481711 0.526897 0.557378 ];
```

But let's represent this in a table to be able to better understand the results.
From here to the rest of the tables we will see, se would represent in the **first row the different types of approaches**, being **seq** the **sequential** one and the **numbers** the different amount of **threads** using when using parallelization. The second row will contain the acceleration of the parallel version vs the sequential one.

➔Road_usa.mtx:

Type	Seq	1	6	10	14	18	20	24	28	32	36	40	44
Time	1.730	1.636	0.418	0.455	0.400	0.373	0.387	0.347	0.388	0.379	0.408	0.409	0.432
SpeedUp	-	1.057	4.134	3.803	4.323	4.635	4.464	4.989	4.454	4.558	4.238	4.227	4.006

➔ Delaunay_n24.mtx

Type	Seq	1	6	10	14	18	20	24	28	32	36	40	44
Time	1.471	1.471	0.477	0.344	0.282	0.244	0.233	0.232	0.217	0.224	0.201	0.225	0.231
SpeedUp	-	1.000	3.081	4.275	5.210	6.027	6.300	6.338	6.792	6.573	7.303	6.542	6.374

➔ Hugebubbles-00020.mtx

Type	Seq	1	6	10	14	18	20	24	28	32	36	40	44
Time	3.165	3.425	0.791	0.526	0.441	0.402	0.331	0.350	0.328	0.363	0.344	0.349	0.365
SpeedUp	-	0.924	4.001	6.017	7.175	7.868	9.575	9.049	9.649	8.713	9.212	9.077	8.662

➔ rgg_n_2_22_s0.mtx

Type	Seq	1	6	10	14	18	20	24	28	32	36	40	44
Time	1.037	1.032	0.218	0.164	0.147	0.133	0.131	0.126	0.131	0.139	0.144	0.148	0.146
SpeedUp	-	1.005	4.759	6.326	7.057	7.800	7.919	8.233	7.919	7.463	7.204	7.009	7.105

➔Explanations:

Even Though all of them got some speedup, there are graphs that have made better improvements than others. This is directly connected to some factors we will study.

1. Firstly the **amount of vertices and edges the graph has**, the bigger these numbers are, the better performance a parallel version can give, as parallelization only makes sense when we have big amounts of data. So we will show the relation between the edges and the vertices, being V the vertices and E the edges and E/V the relation between them.

2. Secondly the **relation between edges and vertices**, the more edges it has compared the vertices they got, the better improvement the parallel version will do. This is because in each round, there would be more vertices to explore.

3. And finally the **depth of the graph**. The depth of the graph also affects, as for every layer, the parallel algorithm has to synchronize the threads, which takes some time because the threads would have to wait until they all get the same point (then it would not make sense to have a very big number of threads). So if the graph is very deep, then the threads would need to communicate many times (one per layer), losing time. In the other hand, the shallower the depth gets, the more can help having more threads working together (always knowing that there is a point where the limit is the cores the system has).

Now we will take a look in each of the graphs and study the performance in each of them taking these factors into account.

➔Experiments Explanations:

◆ Road_usa:

V=23.947.347, **E=**28.854.312, **E/V=**1.2, **Depth=**6262

The speedup is the lowest one of the four, but we still got an algorithm than runs **5 times faster**, this is because the relation between the edges and the vertices is almost 1, making the parallel algorithm not be able to use all the potential. Also the depth is very large, the largest graph of the 4.

◆ Delaunay_n24:

V=16.777.216, **E=**50.331.601, **E/V=**3.8, **Depth=**1651

Here we got some serious speed up, **7.3 times faster**. As we can see, the difference between the amount of edges vs the amount of vertices has increased to 3.8, this can make the parallel version take the advantage of having more vertices to explore in each round and therefore taking the advantage of having the parallel program. It also affects that the amount of vertices and edges is a big amount, so the larger the numbers are, the bigger improvement the parallel version can have. The depth also plays a good role, being the shallowest one of the 4.

◆ Hugebubbles-00020:

V=21.198.119, **E=**31.790.179, **E/V=**1.5, **Depth=**4500

In this one the relation of edges and vertices is not that big but the graph has many vertices and it is not that deep, as for example Road_usa, so the parallel version is giving good improvements compared to the sequential. We got almost **10 times faster** results, the best improvement.

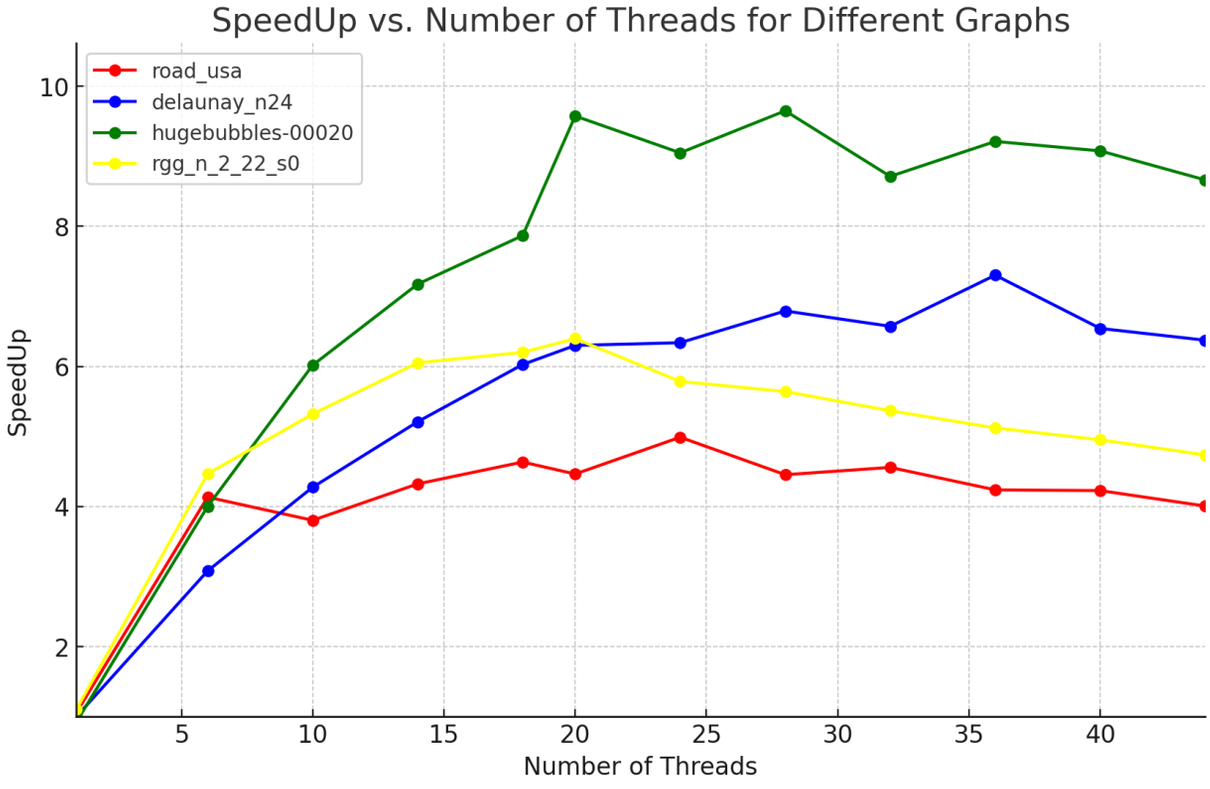
◆ **rgg_n_2_22_s0:**

V=4.194.304, **E=**30.359.198, **E/V=**7.2, **Depth=**1411

In this case, the relation between the edges and vertices seems to be very favorable as is the biggest one we have gotten. The deep is also the shallowest one we got, the results are one of the best ones, **8.2 times faster**. Maybe if the amount of vertices was bigger we could have got even better results, it has a tiny amount of vertices compared to the rest, so even though the edges and depth were favorable, the vertices also matter.

Plot of the different speed ups

For a better visualization let's take a look at a plot with the different speed ups for each graph.



(At the end of the report there are more reports with actually better speedups due to slow sequential runs)

Both the code and discussion is very nice!

2. Alternative Mixed Sequential and Parallel BFS

Now we would start doing the algorithm sequentially and then start at some deep going parallel. In the parallel region the vertices will be divided each k rounds. This means that every thread should work in their own vertices but they all are doing the same deep level at the same time.

To be able to choose the amount of rounds the algorithm should go sequentially or the amount of rounds the threads should work alone until redistributing the vertices we will need these two variables.

```
int sequentialRounds=50; // Rounds before parallel
int rounds_k=5;          // Rounds before copying to S
```

Now we will need to have 2 private lists, as every vertice will be working alone for k rounds.

```
int *local_current_T;       // Vertexs of this round
int current_counter=0;      // Counter of vertexs of this round

int *local_new_T;           // Vertexs of next round
int newVertexs_counter=0;   // Counter of vertexs of next round

local_current_T = (int *) malloc(sizeof(int)*(n+2));
local_new_T = (int *) malloc(sizeof(int)*(n+2));
```

It would have been possible to set this to a smaller value

We will make only one thread to initialize the global variables


```
#pragma omp single
    {
for(i=1;i<=n;i++) {     // Set that every node is unvisited
    p[i] = -1;          // Using -1 to mark that a vertex is unvisited
    dist[i] = -1;
}

  p[1] = 1;        // Set the parent of starting vertex to itself
  dist[1] = 0;     // Set the distance from the starting vertex to
itself
  T[0] = 1; // Num Vertexs of this roun
```

And then start with a sequential approach for the given number of rounds.

```
 current_counter=1;     // Amount of vertexs we are exploring
at this time
  k=0;
  while(current_counter != 0 && k<sequentialRounds){
    newVertexs_counter=0;  // Save how many vertexs will be
in the next round
    for(i=0; i<current_counter; i++){
      v = S[i];
      for(j=ver[v];j<ver[v+1];j++) { // Go through the neighbors
of v
        w = edges[j];              // Get next neighbor w of v
        if (p[w] == -1) {          // Check if w is undiscovered
          p[w] = v;                // Set v as the parent of w
          dist[w] = dist[v]+1;     // Set distance of w
```

```
                local_current_T[newVertexs_counter]= w; // Add w
to local_T and increase number of vertices discovered
                newVertexs_counter++;
            }
        }
    }
    current_counter = newVertexs_counter;
    k++;
    // Update S
    temp = S;
    S = local_current_T;
    local_current_T = temp;
    T[0] = newVertexs_counter;
}
    } // Finish of the sequential code
```

NB! Note that your code only works if you do an even number of sequential rounds. For an odd number thread 0 will have a different setting of S and T than the other threads.

Then we will get into the parallel search, and it will start as the parallel algorithm does, getting the vertices from the common array S.

```
while(T[0]!= 0){
    current_counter=0;     // Amount of vertices we are
exploring at this time
    newVertexs_counter=0;  // Save how many vertices will be
in the next round

    #pragma omp for nowait  // No wait to make the write in T
vertices they discovered
    for(i=0; i<T[0]; i++){
        // FIRST GET THE VERTEXS FROM S
        v =  S[i]; // first store localy the partition of the vertexs
```

```
        for(j=ver[v];j<ver[v+1];j++) {     // Go through the
neighbors of v
            w = edges[j];                  // Get next neighbor w of v
            if (p[w] == -1) {              // Check if w is undiscovered
                p[w] = v;                  // Set v as the parent of w
                dist[w] = dist[v]+1;       // Set distance of w
                local_current_T[newVertexs_counter]= w; // Add w
to local_T d
                newVertexs_counter++;      // Increase the number
of vertices discovered
            }
        }
    }
```

And now will appear a change in the fully parallel version versus this alternative one, and is that every vertex will keep on looking for new vertices alone, without redistributing the amount of possible vertices.

```
    if(rounds_k > 1){ // If not there would be 2 barriers
        #pragma omp barrier // FIRST ROUND FINISHED

        for(k=1; k<rounds_k; k++){
            current_counter = newvertices_counter;  // Change
the new vertices to current ones
            newvertices_counter=0;                  // Reset the new
ones to 0
            for(i=0; i<current_counter; i++){  // Each round
                v = local_current_T[i];
                for(j=ver[v];j<ver[v+1];j++) {     // Go through the
neighbors of v
                    w = edges[j];    // Get next neighbor w of v
                    if (p[w] == -1) {  // Check if w is undiscovered
```

```
                    p[w] = v;        // Set v as the parent of w
                    dist[w] = dist[v]+1;       // Set distance of w
                    local_new_T[newvertices_counter]= w; //
Add w to local_T d
                    newvertices_counter++;      // Increase the
num of vertices discovered
                }
            }
        }
        // Update the local_current_T
        temp = local_current_T;
        local_current_T = local_new_T;
        local_new_T = temp;
        #pragma omp barrier // Another round
    }
}
```

Finally we will need to prepare the redistribution of the vertices when the amount of specified rounds for the vertex to work alone is reached. Therefore we do the same as in parallel bfs and copy the local discovered vertices into the global variable S.

```
T[offset+thread_id] = newvertices_counter;

#pragma omp barrier

// Prepare the index where each vertex should write in
S, it can be done in parallel
// For example thread_0 writes at 0, thread_1 writes
where thread_0 finishes...
int start = 0; // Get the first amount of the frist thread
```

```
        // Gets where do start writing in the memory, finishes
when knowing where to start

        // to do not work more than needed
        for(i=0; i<thread_id; i++){
            start += T[offset+i];
        }

        // Copy to the global array in parallel
        for(i=0; i<newvertices_counter; i++){
            S[start+i]= local_current_T[i];
        }

        if(thread_id == (num_threads-1)){ // only one vertex
sets the total amount of threads
            T[0] = start+newvertices_counter; //Set the amount
of new vertices for next round
        }

        #pragma omp barrier
```

● Experiments Sequential vs Alternative-Parallel

We will check for each graph different configurations based on the information we know. In most approaches the amount of threads used will not be more than 44 because although the CPU has up to 80 cores we seem to get the best results between 20 to 40, and then a decrease once we pass these values.

The first thing we would need to do is to prove that our alternative bfs algorithm is working correctly is to see if we put no sequential rounds and a redistribution of the vertex between threads in every layer, we should get moreless the same runtime as in parallel bfs, as it is almost the same algorithm then.

-**TP** represents the **parallel BFS execution times**.
-**TAP** represents the **execution times of the alternative BFS**.
-**SU** is the **SpeedUp** of the **alternative BFS over parallel BFS** for each value of threads.

We will use the next pages just to show the different timing performance, to see the conclusions made, just skip the tables and go to page 26.

- Compare ABFS with PBFS in same conditions

◆ Road_usa:

V=23.947.347, **E=**28.854.312, **E/V=**1.2, **Depth=**6262

Type	Seq	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.548	1.504	0.574	0.409	0.356	0.344	0.342	0.351	0.347	0.330	0.376	0.397	0.415
TAP	1.528	1.528	0.590	0.386	0.385	0.356	0.355	0.367	0.353	0.383	0.412	0.397	0.437
SU	-	**0.984**	**0.972**	**1.060**	**0.923**	**0.967**	**0.963**	**0.957**	**0.983**	**0.862**	**0.909**	**1.000**	**0.949**

◆ Delaunay_n24:

V=16.777.216, **E=**50.331.601, **E/V=**3.8, **Depth=**1651

Type	Seq	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.411	1.390	0.472	0.320	0.282	0.239	0.233	0.223	0.218	0.193	0.212	0.219	0.207
TAP	1.408	1.408	0.481	0.353	0.289	0.257	0.238	0.222	0.218	0.219	0.212	0.235	0.242
SU	-	**1.002**	**0.981**	**0.907**	**0.975**	**0.929**	**0.979**	**1.005**	**1.000**	**0.881**	**1.000**	**0.932**	**0.855**

◆ Hugebubbles-00020

V=21.198.119, **E=**31.790.179, **E/V=**1.5, **Depth=**4500

Type	Seq	1	6	10	14	18	20	24	28	32	36	40	44
TP	2.819	2.791	0.677	0.557	0.448	0.422	0.443	0.418	0.423	0.427	0.441	0.472	0.483
TAP	2.802	2.802	0.683	0.567	0.467	0.443	0.431	0.428	0.424	0.438	0.447	0.480	0.518
SU	-	**0.996**	**0.991**	**0.982**	**0.959**	**0.952**	**1.028**	**0.977**	**1.000**	**0.975**	**0.986**	**0.983**	**0.932**

◆ rgg_n_2_22_s0:

V=4.194.304, **E=**30.359.198, **E/V=**7.2, **Depth=**1411

Type	Seq	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.014	0.874	0.263	0.174	0.148	0.145	0.139	0.140	0.139	0.149	0.169	0.172	0.178
TAP	0.872	0.872	0.254	0.173	0.152	0.143	0.141	0.139	0.134	0.159	0.170	0.171	0.177
SU	-	**1.002**	**1.035**	**1.006**	**0.974**	**1.014**	**0.986**	**1.007**	**1.037**	**0.937**	**0.994**	**1.006**	**1.005**

The **average speedup is of 0.974**, proving that the algorithms are almost the same and therefore that the runtime is similar.

- Find the best Sequential Rounds value

For this section we would be looking for the value of sequential rounds that provide us the best performance, trying in each graph different values. We will keep the distance of communication to 1, making the algorithm to distribute the thread in each layer, later on we will look for the best value of this variable. I will put the tables just to show the times, but we will focus on the overall mean Speed up.

◆ **Road_usa:** (best number of sequential rounds is 50)
V=23.947.347, **E=**28.854.312, **E/V=**1.2, **Depth=**6262

-Number of sequential rounds=0

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.524	0.580	0.418	0.378	0.338	0.335	0.345	0.338	0.347	0.343	0.410	0.421
TAP	1.743	0.599	0.432	0.380	0.362	0.358	0.362	0.362	0.361	0.364	0.407	0.403
SU	0.874	0.968	0.967	0.995	0.936	0.936	0.953	0.934	0.962	0.942	1.006	1.044

Overall mean SpeedUp: 0.960

-Number of sequential rounds=50

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.682	0.517	0.350	0.359	0.336	0.334	0.346	0.351	0.359	0.352	0.387	0.401
TAP	1.773	0.480	0.415	0.354	0.369	0.347	0.329	0.354	0.366	0.360	0.371	0.389
SU	0.946	1.077	0.844	1.015	0.912	0.963	1.052	0.995	0.976	0.978	1.045	1.030

Overall mean SpeedUp: 0.985

-Number of sequential rounds=100

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.541	0.578	0.435	0.371	0.362	0.337	0.344	0.331	0.344	0.338	0.371	0.378
TAP	1.588	0.495	0.447	0.376	0.372	0.345	0.343	0.351	0.365	0.377	0.390	0.393
SU	0.971	1.168	0.974	0.985	0.975	0.977	1.003	0.944	0.942	0.897	0.950	0.963

Overall mean SpeedUp: 0.979

-Number of sequential rounds=150

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.598	0.547	0.418	0.351	0.338	0.340	0.330	0.321	0.340	0.333	0.366	0.359
TAP	1.722	0.552	0.409	0.386	0.344	0.351	0.339	0.344	0.359	0.371	0.375	0.390
SU	0.928	0.990	1.023	0.908	0.982	0.969	0.974	0.934	0.948	0.898	0.976	0.922

Overall mean SpeedUp: 0.954

-Number of sequential rounds=200

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.512	0.458	0.411	0.383	0.329	0.351	0.332	0.328	0.345	0.370	0.357	0.407
TAP	1.567	0.570	0.403	0.368	0.378	0.350	0.357	0.359	0.366	0.361	0.380	0.404
SU	0.965	0.804	1.020	1.040	0.870	1.004	0.930	0.914	0.942	1.025	0.939	1.007

Overall mean SpeedUp: 0.955

-Number of sequential rounds=300

(omitted to make the rest fit in 2 pages)

Overall mean SpeedUp: 0.940

◆ **Delaunay_n24:** (best number of sequential rounds is 100)
V=16.777.216, **E=**50.331.601, **E/V=**3.8, **Depth=**1651

-Number of sequential rounds=0

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.377	0.477	0.326	0.274	0.236	0.226	0.225	0.207	0.198	0.191	0.202	0.207
TAP	1.399	0.487	0.355	0.287	0.255	0.250	0.242	0.224	0.211	0.223	0.227	0.242
SU	0.984	0.980	0.918	0.954	0.924	0.904	0.932	0.926	0.938	0.857	0.890	0.854

Overall mean SpeedUp: 0.922

-Number of sequential rounds=50

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.336	0.470	0.317	0.257	0.233	0.229	0.212	0.210	0.203	0.182	0.202	0.207
TAP	1.348	0.480	0.327	0.274	0.248	0.243	0.226	0.227	0.217	0.230	0.228	0.229
SU	0.992	0.981	0.972	0.939	0.939	0.944	0.941	0.923	0.935	0.792	0.885	0.905

Overall mean SpeedUp: 0.929

-Number of sequential rounds=100

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.561	0.439	0.318	0.278	0.241	0.230	0.220	0.209	0.208	0.212	0.205	0.224
TAP	1.578	0.371	0.346	0.290	0.250	0.248	0.241	0.238	0.233	0.230	0.229	0.244
SU	0.989	1.183	0.920	0.960	0.961	0.925	0.913	0.875	0.891	0.922	0.897	0.918

Overall mean SpeedUp: 0.946

-Number of sequential rounds=150

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.508	0.480	0.333	0.274	0.238	0.224	0.218	0.203	0.193	0.188	0.197	0.205
TAP	1.557	0.506	0.359	0.302	0.262	0.256	0.244	0.233	0.241	0.233	0.226	0.237
SU	0.968	0.949	0.927	0.907	0.907	0.876	0.892	0.872	0.803	0.808	0.871	0.866

Overall mean SpeedUp: 0.887

-Number of sequential rounds=200

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.372	0.484	0.335	0.272	0.238	0.228	0.206	0.197	0.199	0.195	0.215	0.214
TAP	1.568	0.494	0.375	0.304	0.279	0.269	0.264	0.255	0.249	0.245	0.246	0.265
SU	0.875	0.981	0.893	0.894	0.854	0.849	0.780	0.771	0.800	0.796	0.874	0.809

Overall mean SpeedUp: 0.848

-Number of sequential rounds=300

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.346	0.472	0.333	0.266	0.226	0.224	0.215	0.199	0.212	0.193	0.201	0.206
TAP	1.356	0.529	0.397	0.338	0.318	0.302	0.311	0.288	0.282	0.294	0.286	0.301
SU	0.992	0.892	0.838	0.785	0.710	0.741	0.693	0.691	0.752	0.658	0.702	0.684

Overall mean SpeedUp: 0.762

◆ **Hugebubbles-00020:** (best number of sequential rounds is 50)
V=21.198.119, **E=**31.790.179, **E/V=**1.5, **Depth=**4500

-Number of sequential rounds=0

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	2.840	0.687	0.545	0.444	0.397	0.376	0.374	0.347	0.354	0.364	0.379	0.376
TAP	2.859	0.687	0.549	0.462	0.409	0.386	0.394	0.370	0.372	0.363	0.381	0.414
SU	0.994	1.000	0.992	0.963	0.972	0.975	0.949	0.939	0.952	1.004	0.995	0.907

Overall mean SpeedUp: 0.970

-Number of sequential rounds=50

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	2.899	0.692	0.546	0.450	0.421	0.395	0.514	0.381	0.401	0.411	0.440	0.424
TAP	3.147	0.703	0.533	0.453	0.421	0.468	0.408	0.396	0.406	0.461	0.452	0.434
SU	0.921	0.984	1.024	0.995	0.999	0.845	1.261	0.962	0.986	0.892	0.975	0.977

Overall mean SpeedUp: 0.985

-Number of sequential rounds=100

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	3.542	0.702	0.540	0.450	0.429	0.411	0.407	0.389	0.396	0.384	0.428	0.445
TAP	3.585	0.667	0.571	0.467	0.442	0.408	0.438	0.400	0.424	0.434	0.457	0.459
SU	0.988	1.053	0.945	0.963	0.971	1.007	0.928	0.973	0.934	0.884	0.936	0.969

Overall mean SpeedUp: 0.963

-Number of sequential rounds=150

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	3.425	0.791	0.526	0.441	0.402	0.331	0.350	0.328	0.363	0.344	0.349	0.365
TAP	3.291	0.742	0.531	0.453	0.399	0.392	0.360	0.399	0.378	0.366	0.379	0.397
SU	1.041	1.066	0.991	0.973	1.008	0.844	0.970	0.823	0.962	0.937	0.921	0.921

Overall mean SpeedUp: 0.955

-Number of sequential rounds=200

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	2.727	0.724	0.530	0.459	0.427	0.425	0.414	0.397	0.416	0.423	0.442	0.454
TAP	3.552	0.694	0.529	0.470	0.437	0.445	0.427	0.410	0.437	0.435	0.466	0.489
SU	0.768	1.043	1.001	0.975	0.977	0.955	0.970	0.967	0.951	0.972	0.947	0.927

Overall mean SpeedUp: 0.955

-Number of sequential rounds=300

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	2.697	0.688	0.532	0.466	0.433	0.428	0.436	0.424	0.421	0.443	0.475	0.478
TAP	2.737	0.794	0.567	0.484	0.458	0.450	0.450	0.438	0.462	0.447	0.500	0.505
SU	0.986	0.866	0.939	0.965	0.946	0.953	0.969	0.968	0.910	0.990	0.950	0.947

Overall mean SpeedUp: 0.949

◆ **rgg_n_2_22_s0:** (best number of sequential rounds is 0)
V=4.194.304, **E=**30.359.198, **E/V=**7.2, **Depth=**1411

Number of sequential rounds=0

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	0.852	0.250	0.174	0.150	0.142	0.147	0.168	0.180	0.185	0.190	0.187	0.200
TAP	0.867	0.245	0.179	0.151	0.143	0.148	0.169	0.170	0.189	0.189	0.190	0.198
SU	0.983	1.019	0.970	0.992	0.993	0.991	0.995	1.056	0.980	1.004	0.981	1.013

Overall mean SpeedUp: 0.998

-Number of sequential rounds=50

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	0.855	0.255	0.173	0.158	0.153	0.159	0.174	0.180	0.193	0.203	0.206	0.209
TAP	0.867	0.253	0.181	0.162	0.158	0.160	0.177	0.187	0.195	0.199	0.204	0.207
SU	0.986	1.007	0.955	0.973	0.971	0.996	0.986	0.964	0.993	1.023	1.011	1.009

Overall mean SpeedUp: 0.989

-Number of sequential rounds=100

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.048	0.255	0.178	0.149	0.139	0.140	0.151	0.157	0.164	0.164	0.175	0.183
TAP	0.933	0.266	0.189	0.162	0.152	0.154	0.162	0.163	0.172	0.172	0.182	0.190
SU	1.123	0.957	0.946	0.920	0.914	0.905	0.934	0.963	0.956	0.950	0.960	0.963

Overall mean SpeedUp: 0.958

-Number of sequential rounds=150

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.043	0.253	0.175	0.145	0.134	0.135	0.137	0.148	0.161	0.167	0.178	0.182
TAP	0.992	0.278	0.190	0.179	0.166	0.163	0.161	0.169	0.179	0.190	0.200	0.200
SU	1.052	0.911	0.923	0.807	0.806	0.825	0.849	0.875	0.900	0.880	0.893	0.910

Overall mean SpeedUp: 0.886

-Number of sequential rounds=200

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	0.851	0.255	0.181	0.158	0.152	0.151	0.168	0.170	0.180	0.183	0.198	0.199
TAP	0.864	0.298	0.218	0.196	0.181	0.187	0.206	0.205	0.209	0.220	0.229	0.235
SU	0.984	0.854	0.828	0.804	0.838	0.808	0.817	0.829	0.862	0.831	0.865	0.848

Overall mean SpeedUp: 0.847

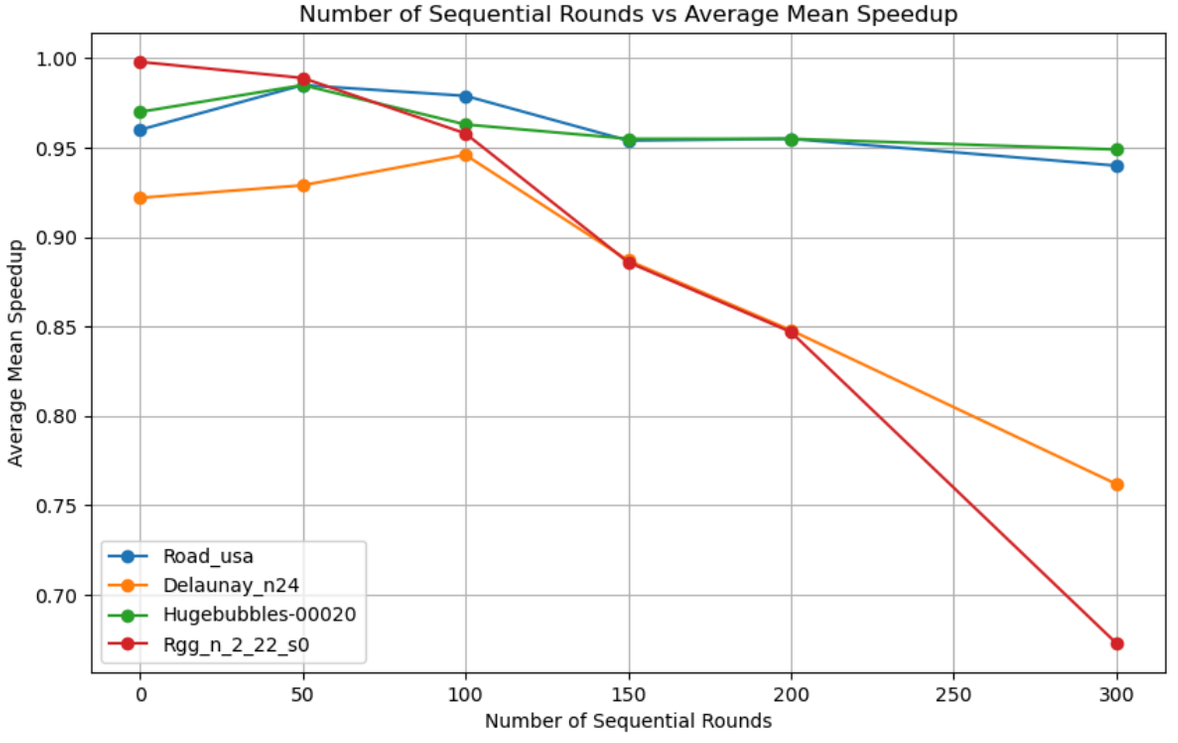
-Number of sequential rounds=300

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	0.858	0.259	0.176	0.147	0.140	0.140	0.156	0.161	0.165	0.175	0.182	0.188
TAP	0.874	0.374	0.275	0.246	0.237	0.230	0.236	0.271	0.249	0.263	0.276	0.262
SU	0.982	0.693	0.640	0.600	0.591	0.610	0.660	0.594	0.663	0.667	0.659	0.716

Overall mean SpeedUp: 0.673

Results

So after all this search, we can now tell the best number of sequential rounds before going parallel but would depend on the graph we are working with. Let's represent this in a plot to be able to appreciate it better. The y value represents the average speedup of the alternative algorithm with respect to the parallel one; among the different configurations of threads. The x represents the amount of sequential rounds. We can clearly see that for more than 100 sequential rounds, the parallel algorithm is faster.



Best values for each were:

- **Road usa** = 50
- **Delanuay =** 100
- **HugeBubbles** = 50
- **Rgg** = 0

We can use a **general value of 50 first sequential rounds**.

You could have tried smaller values. Also, note that what we are realy interested in here is to speed up abfs(). Thus it is not so important how it compares to pbfs().

-Find the best number of rounds until redistribution.

Now we will try the best amount of layers each vertex should investigate alone until redistributing the vertices again. This time we will run the 4 graphs with a sequential start of 50 rounds, then as we did before we will try different values of k and we will get the best average speed up. We will use the next pages just to show the different timing performance, to see the conclusions made, just skip the tables and go to page 38.

- **Rounds until redistribution = 1**

  This was calculated before so we will just show the average speedup for each graph.

  - **Road_usa:** Mean SpeedUp: 0.985
  - **Delaunay_n24:** Mean SpeedUp: 0.929
  - **Hugebubbles-00020:** Mean SpeedUp: 0.985
  - **rgg_n_2_22_s0:** Mean SpeedUp: 0.989

The average mean Speed Up is 0,972

- **Rounds until redistribution = 2**

  - **Road_usa:**

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.624	0.582	0.406	0.382	0.326	0.319	0.321	0.322	0.314	0.342	0.348	0.358
TAP	1.825	0.611	0.455	0.400	0.363	0.362	0.353	0.342	0.355	0.365	0.370	0.370
SU	1.124	1.051	1.121	1.048	1.116	1.135	1.099	1.061	1.128	1.066	1.064	1.034

Mean SpeedUp: 1.087

◆ Delaunay_n24:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.384	0.496	0.341	0.275	0.244	0.243	0.240	0.215	0.218	0.214	0.212	0.206
TAP	1.440	0.512	0.359	0.291	0.273	0.257	0.240	0.242	0.238	0.235	0.241	0.245
SU	1.040	1.033	1.054	1.060	1.118	1.056	1.001	1.125	1.092	1.097	1.137	1.189

Mean SpeedUp: 1.084

◆ Hugebubbles-00020:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	2.859	0.711	0.557	0.457	0.412	0.387	0.388	0.381	0.390	0.399	0.416	0.429
TAP	3.153	0.695	0.576	0.469	0.428	0.422	0.414	0.389	0.398	0.408	0.436	0.452
SU	1.103	0.978	1.034	1.026	1.040	1.091	1.066	1.020	1.019	1.023	1.048	1.054

Mean SpeedUp: 1.042

◆ rgg_n_2_22_s0:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	0.873	0.239	0.166	0.144	0.141	0.137	0.133	0.149	0.160	0.174	0.171	0.176
TAP	0.875	0.234	0.182	0.148	0.145	0.131	0.140	0.144	0.155	0.167	0.171	0.169
SU	1.003	0.982	1.099	1.024	1.028	0.953	1.049	0.971	0.973	0.961	1.001	0.961

Mean SpeedUp: 1.000

The average mean Speed Up is 1,053

- **Rounds until redistribution = 3**

◆ Road_usa:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	2.116	0.542	0.447	0.356	0.313	0.316	0.325	0.336	0.308	0.394	0.511	0.659
TAP	2.278	0.608	0.403	0.357	0.335	0.331	0.314	0.345	0.383	0.365	0.354	0.336
SU	0.929	0.892	1.110	0.997	0.933	0.953	1.035	0.973	0.804	1.079	1.444	1.961

Mean SpeedUp: 1.064

◆ Delaunay_n24:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.486	0.412	0.292	0.246	0.224	0.245	0.276	0.292	0.235	0.260	0.205	0.625
TAP	1.597	0.408	0.312	0.260	0.254	0.269	0.263	0.296	0.249	0.630	0.556	0.199
SU	0.931	1.011	0.935	0.946	0.883	0.911	1.048	0.986	0.942	0.413	0.369	3.140

Mean SpeedUp: 1.141

◆ Hugebubbles-00020:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	3.285	0.802	0.617	0.566	0.532	1.434	0.405	0.438	0.470	0.483	0.967	1.473
TAP	3.412	0.865	0.629	0.586	0.681	1.127	0.437	0.401	0.469	0.636	0.577	1.133
SU	0.962	0.927	0.981	0.967	0.780	1.271	0.927	1.092	1.002	0.759	1.678	1.299

Mean SpeedUp: 1.021

◆ rgg_n_2_22_s0:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	0.977	0.239	0.170	0.145	0.141	0.137	0.130	0.130	0.124	0.135	0.137	0.140
TAP	0.977	0.234	0.180	0.149	0.133	0.131	0.130	0.125	0.129	0.132	0.138	0.148
SU	1.000	1.022	0.944	0.974	1.060	1.046	1.000	1.040	0.961	1.022	0.993	0.946

Mean SpeedUp: 1.001

The average mean Speed Up is 1.057

- **Rounds until redistribution = 4**

◆ Road_usa:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.664	0.511	0.382	0.360	0.345	0.322	0.302	0.311	0.315	0.331	0.342	0.362
TAP	1.768	0.605	0.433	0.426	0.336	0.312	0.312	0.305	0.304	0.326	0.334	0.335
SU	0.941	0.844	0.882	0.845	1.028	1.034	0.968	1.021	1.035	1.016	1.024	1.080

Mean SpeedUp: 0.976

◆ Delaunay_n24:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.424	0.466	0.311	0.258	0.240	0.222	0.214	0.212	0.216	0.195	0.204	0.214
TAP	1.380	0.482	0.331	0.303	0.252	0.247	0.238	0.233	0.220	0.210	0.233	0.222
SU	1.032	0.966	0.940	0.852	0.954	0.900	0.899	0.908	0.982	0.931	0.872	0.966

Mean SpeedUp: 0.933

◆ Hugebubbles-00020:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	2.830	0.673	0.537	0.462	0.431	0.407	0.395	0.409	0.427	0.496	0.495	0.493
TAP	2.929	0.740	0.541	0.467	0.421	0.420	0.416	0.416	0.421	0.447	0.445	0.464
SU	0.966	0.910	0.993	0.988	1.023	0.969	0.951	0.983	1.014	1.108	1.112	1.062

Mean SpeedUp: 1.007

◆ rgg_n_2_22_s0:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	0.885	0.232	0.176	0.152	0.138	0.138	0.149	0.159	0.161	0.176	0.179	0.187
TAP	0.900	0.260	0.183	0.151	0.146	0.149	0.137	0.148	0.154	0.157	0.159	0.166
SU	0.984	0.894	0.964	1.006	0.949	0.925	1.086	1.077	1.044	1.118	1.120	1.124

Mean SpeedUp: 1.024

The average mean Speed Up is 0.985

- **Rounds until redistribution = 5**

◆ Road_usa:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.712	0.609	0.338	0.380	0.348	0.329	0.332	0.322	0.319	0.341	0.341	0.343
TAP	1.826	0.641	0.478	0.385	0.351	0.352	0.316	0.318	0.307	0.328	0.312	0.331
SU	0.937	0.950	0.706	0.987	0.993	0.935	1.053	1.015	1.037	1.040	1.092	1.036

Mean SpeedUp: 0.982

◆ Delaunay_n24:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.523	0.491	0.337	0.276	0.241	0.227	0.214	0.207	0.196	0.184	0.199	0.195
TAP	1.565	0.508	0.361	0.288	0.247	0.243	0.221	0.226	0.209	0.199	0.206	0.211
SU	0.973	0.965	0.936	0.960	0.976	0.935	0.969	0.914	0.934	0.922	0.964	0.927

Mean SpeedUp: 0.948

◆ Hugebubbles-00020:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	3.770	0.854	0.725	0.528	0.474	0.466	0.443	0.407	0.426	0.431	0.497	0.449
TAP	3.564	0.837	0.663	0.524	0.467	0.424	0.409	0.389	0.397	0.406	0.411	0.408
SU	1.058	1.020	1.094	1.009	1.014	1.099	1.083	1.047	1.075	1.061	1.210	1.101

Mean SpeedUp: 1.072

◆ rgg_n_2_22_s0:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	0.994	0.344	0.156	0.139	0.139	0.141	0.145	0.162	0.180	0.185	0.192	0.199
TAP	1.170	0.238	0.167	0.142	0.137	0.138	0.144	0.156	0.165	0.171	0.172	0.173
SU	0.850	1.447	0.933	0.977	1.014	1.019	1.005	1.040	1.091	1.081	1.117	1.152

Mean SpeedUp: 1.061

The average mean Speed Up is 1.016

- **Rounds until redistribution = 6**

◆ Road_usa:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.518	0.501	0.440	0.357	0.362	0.348	0.336	0.334	0.349	0.377	0.395	0.395
TAP	1.848	0.522	0.439	0.386	0.345	0.341	0.304	0.321	0.309	0.333	0.372	0.347
SU	0.821	0.960	1.001	0.926	1.047	1.021	1.104	1.039	1.128	1.132	1.062	1.138

Mean SpeedUp: 1.032

◆ Delaunay_n24:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.381	0.347	0.321	0.271	0.233	0.235	0.219	0.208	0.197	0.202	0.204	0.215
TAP	1.432	0.501	0.343	0.287	0.248	0.247	0.231	0.229	0.210	0.221	0.222	0.204
SU	0.964	0.693	0.935	0.942	0.940	0.951	0.947	0.907	0.936	0.914	0.920	1.056

Mean SpeedUp: 0.925

◆ Hugebubbles-00020:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	2.842	0.673	0.557	0.482	0.454	0.426	0.406	0.432	0.427	0.461	0.461	0.477
TAP	2.821	0.709	0.564	0.482	0.435	0.404	0.407	0.401	0.419	0.422	0.438	0.451
SU	1.008	0.949	0.989	0.999	1.045	1.054	0.996	1.078	1.021	1.092	1.052	1.058

Mean SpeedUp: 1.028

◆ rgg_n_2_22_s0:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	0.858	0.258	0.170	0.145	0.138	0.141	0.151	0.164	0.177	0.179	0.190	0.195
TAP	0.872	0.256	0.181	0.147	0.135	0.136	0.136	0.147	0.159	0.162	0.167	0.169
SU	0.984	1.007	0.940	0.984	1.029	1.035	1.109	1.116	1.112	1.105	1.133	1.157

Mean SpeedUp: 1.059

The average mean Speed Up is 1.011

- **Rounds until redistribution = 7**

◆ Road_usa:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.584	0.567	0.413	0.362	0.321	0.326	0.326	0.326	0.335	0.352	0.357	0.335
TAP	1.860	0.452	0.465	0.372	0.332	0.330	0.322	0.298	0.304	0.301	0.311	0.316
SU	0.852	1.254	0.887	0.974	0.966	0.987	1.014	1.091	1.104	1.170	1.146	1.061

Mean SpeedUp: 1.042

◆ Delaunay_n24:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.379	0.423	0.327	0.266	0.235	0.233	0.215	0.198	0.208	0.191	0.192	0.211
TAP	1.427	0.510	0.351	0.282	0.263	0.247	0.236	0.217	0.221	0.201	0.221	0.221
SU	0.966	0.828	0.932	0.941	0.895	0.943	0.910	0.911	0.939	0.951	0.871	0.955

Mean SpeedUp: 0.92

◆ Hugebubbles-00020:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	2.776	0.654	0.559	0.456	0.449	0.436	0.416	0.411	0.412	0.437	0.470	0.495
TAP	2.816	0.763	0.569	0.452	0.443	0.408	0.394	0.381	0.379	0.400	0.428	0.440
SU	0.986	0.858	0.983	1.010	1.012	1.070	1.054	1.080	1.088	1.095	1.098	1.124

Mean SpeedUp: 1.038

◆ rgg_n_2_22_s0:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.114	0.240	0.178	0.147	0.142	0.138	0.124	0.149	0.149	0.157	0.179	0.176
TAP	1.111	0.266	0.186	0.155	0.142	0.131	0.122	0.128	0.133	0.149	0.160	0.160
SU	1.003	0.900	0.961	0.949	1.001	1.054	1.020	1.164	1.120	1.056	1.118	1.102

Mean SpeedUp: 1.037

The average mean Speed Up is 1.009

-Rounds until redistribution = 8

◆ Road_usa:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.730	0.626	0.441	0.382	0.334	0.341	0.324	0.323	0.299	0.340	0.340	0.359
TAP	1.816	0.688	0.468	0.393	0.369	0.350	0.333	0.316	0.306	0.318	0.301	0.326
SU	0.953	0.911	0.942	0.972	0.907	0.974	0.973	1.021	0.977	1.070	1.128	1.098

Mean SpeedUp: 0.994

◆ Delaunay_n24:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	1.419	0.492	0.336	0.269	0.244	0.234	0.209	0.206	0.199	0.195	0.200	0.201
TAP	1.469	0.390	0.354	0.291	0.259	0.242	0.222	0.216	0.206	0.211	0.212	0.220
SU	0.966	1.263	0.950	0.924	0.942	0.966	0.941	0.954	0.965	0.927	0.946	0.912

Mean SpeedUp: 0.971

◆ Hugebubbles-00020:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	3.797	0.854	0.631	0.470	0.473	0.448	0.394	0.382	0.394	0.381	0.372	0.381
TAP	4.417	0.904	0.554	0.506	0.468	0.424	0.389	0.366	0.350	0.352	0.349	0.359
SU	0.860	0.944	1.140	0.930	1.011	1.055	1.014	1.043	1.124	1.084	1.065	1.063

Mean SpeedUp: 1.028

◆ rgg_n_2_22_s0:

x	1	6	10	14	18	20	24	28	32	36	40	44
TP	0.947	0.226	0.182	0.148	0.137	0.132	0.132	0.125	0.130	0.137	0.141	0.144
TAP	0.966	0.279	0.191	0.154	0.142	0.132	0.124	0.121	0.120	0.124	0.129	0.129
SU	0.981	0.811	0.952	0.967	0.960	1.002	1.065	1.036	1.087	1.112	1.089	1.118

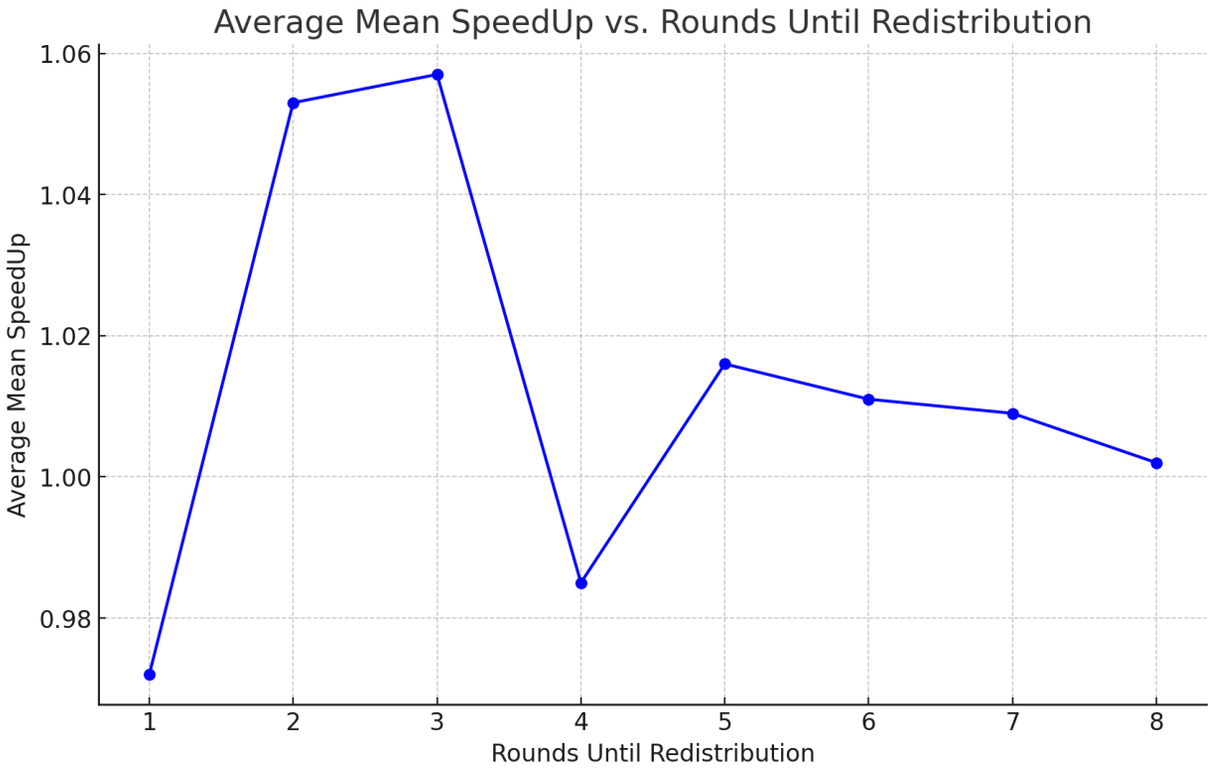
Mean SpeedUp: 1.015

We have gotten some improvement, the average general speedup is of 1.002

Results

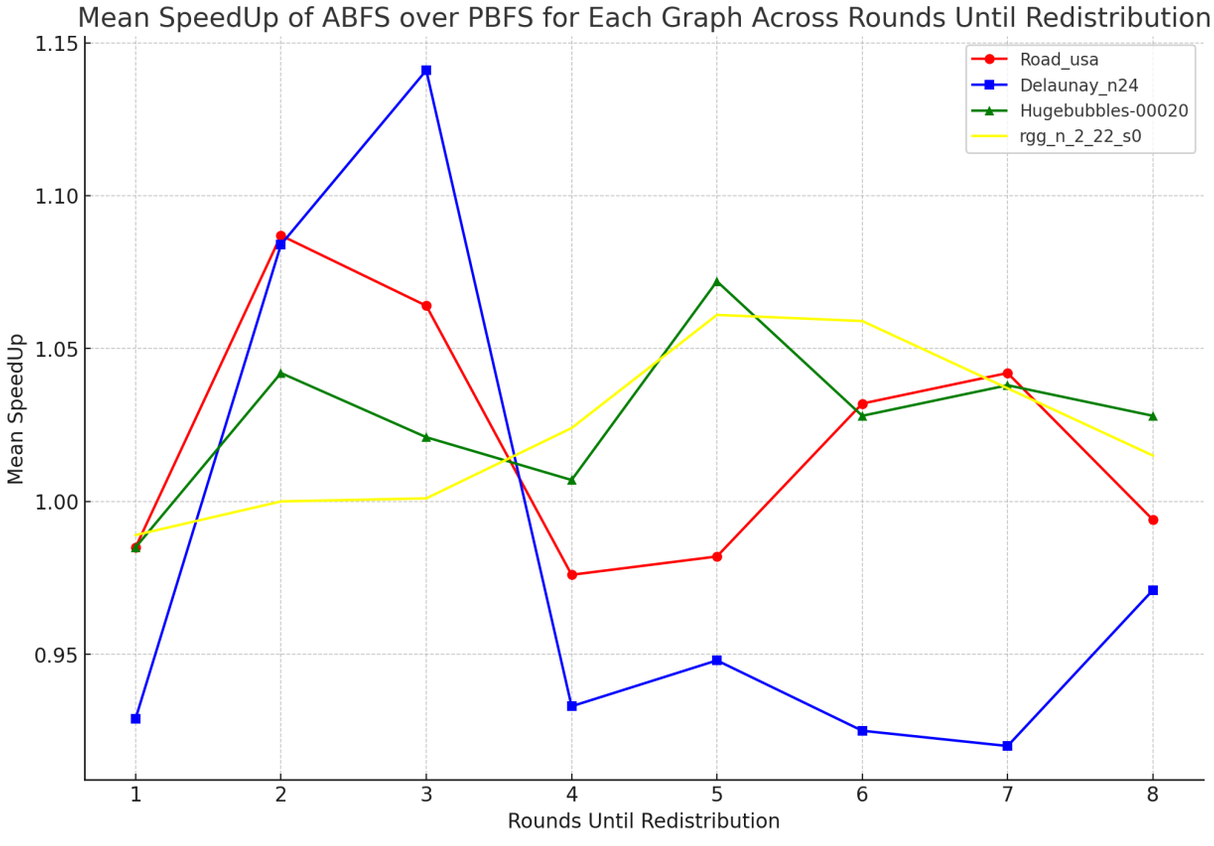
After all this data gathering the best idea to know which approach we should use is to plot the different average speedups and see which value of k gives the best results.

First let's plot the general average speedup of all the graphs changing the value of the rounds until redistribution.



We can see here that f**or a general approach the best value of k should be 3**

But if we want to go deeper and analyze each ~~vertex~~ graph itself we will see that the best value is not the same for all.



Best values for rounds until redistribution each were:

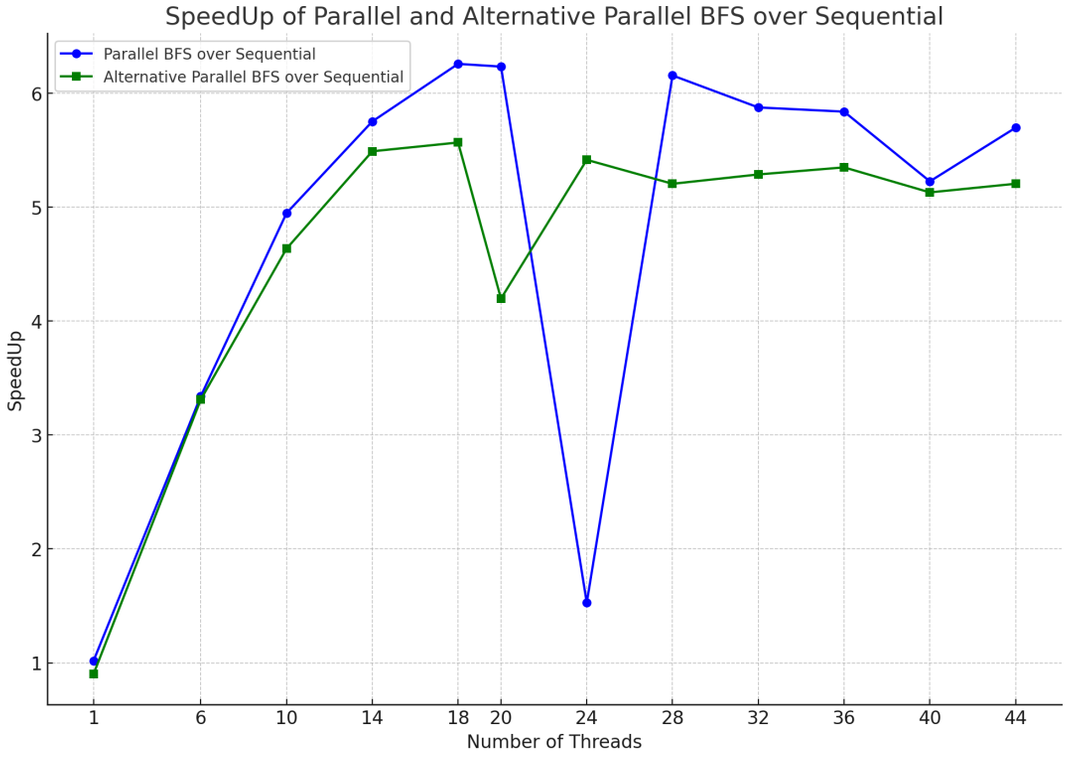
- Road usa = 2
- Delanuay = 3
- HugeBubbles = 5
- Rgg = 5

Now is time to see the performance of the alternative and parallel speed ups with this selected best values.

Best values for sequential rounds each were:

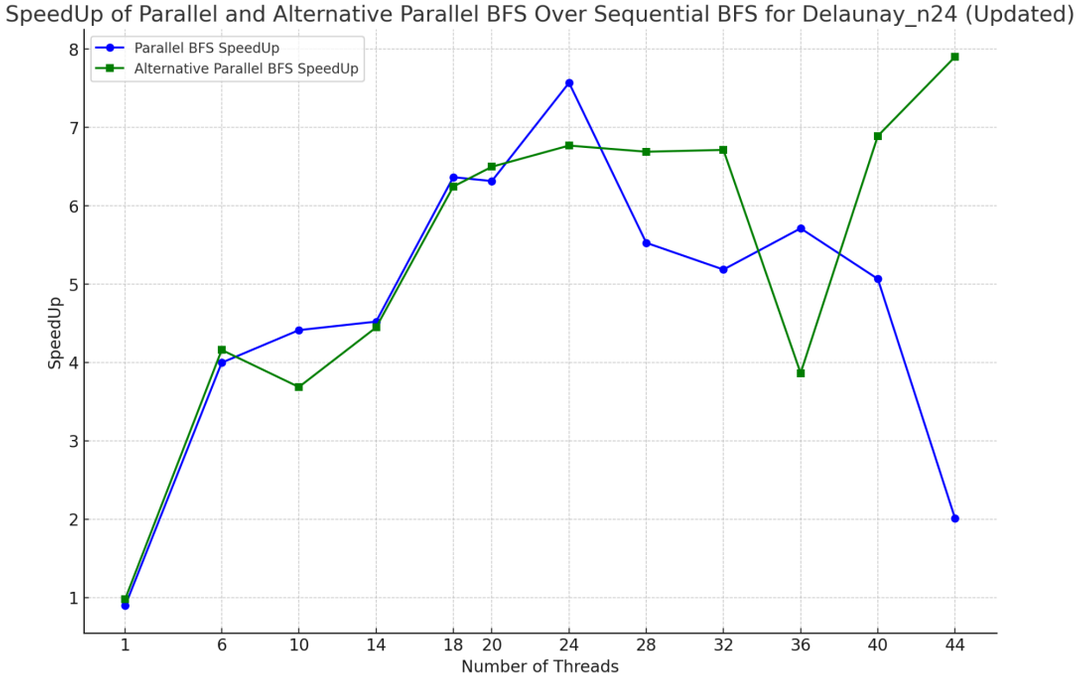
- Road usa = 50
- Delanuay = 100
- HugeBubbles = 50
- Rgg = 0

ROAD_USA



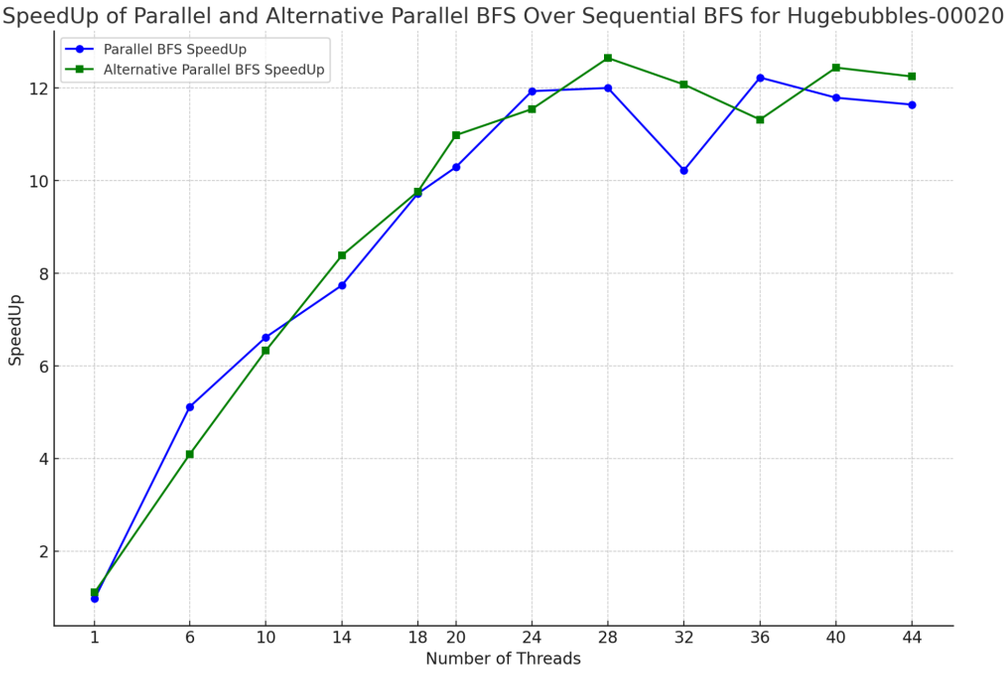
We can see for Road_USA the results were pretty similar to the normal parallel, not getting a any improvement.

DELAUNAY



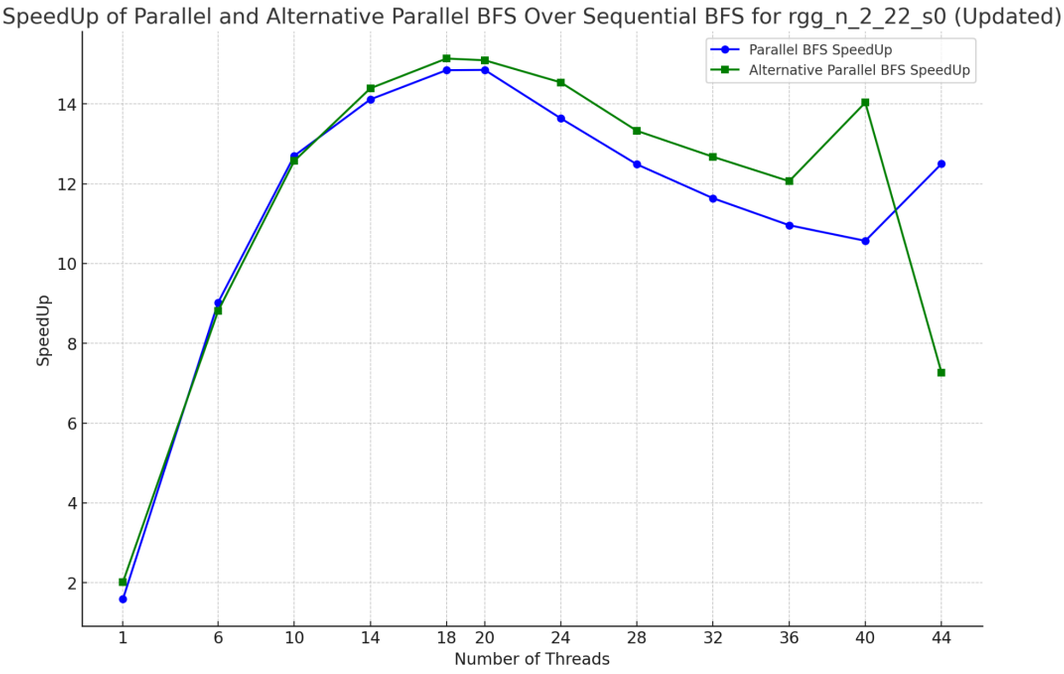
For Delaunay we got the best time with the alternative parallel, making 8 times faster than the sequential algorithm.

HUGEBUBBLES



For HugeBubbles we got the best time again with the alternative parallel, making 12.65 times faster than the sequential algorithm.

RGG



For this one I got a slow sequential run that made amazing best results for parallel and alternative algorithms, making it almost 15 times faster. In this algorithm we clearly see that the alternative approach gives better results.

All in all a very nice report. Some minor shortcomings in the coding, but clear and understable explanations and testing. Report is on the long side and could have been shortened. 30/30