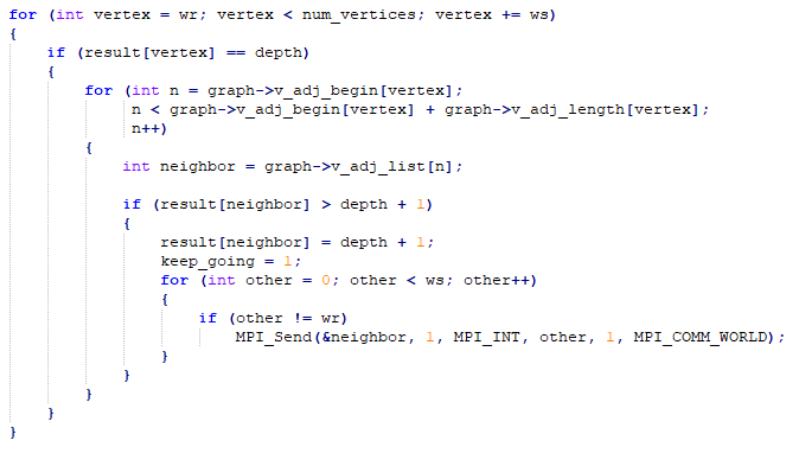
COMP429 HW3

Batuhan Başkaya 64240 – Doğuhan Çiftçi 50356

Part 1 Vertex Distribution

In this part, we implemented distributed approach by splitting graph vertices. In figure 1, each process starts its loop from its rank and increments by world size. So, the whole array will be covered by processors. When a process updates the result array, it sends this information to other processors.

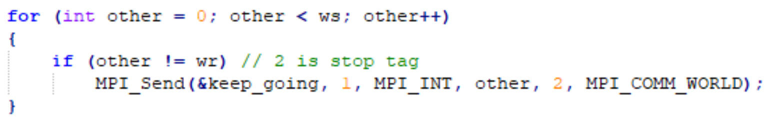


```
for (int vertex = wr; vertex < num_vertices; vertex += ws)
{
    if (result[vertex] == depth)
    {
        for (int n = graph->v_adj_begin[vertex];
             n < graph->v_adj_begin[vertex] + graph->v_adj_length[vertex];
             n++)
        {
            int neighbor = graph->v_adj_list[n];

            if (result[neighbor] > depth + 1)
            {
                result[neighbor] = depth + 1;
                keep_going = 1;
                for (int other = 0; other < ws; other++)
                {
                    if (other != wr)
                        MPI_Send(&neighbor, 1, MPI_INT, other, 1, MPI_COMM_WORLD);
                }
            }
        }
    }
}
```

Figure 1 Main loop of vertex_dist

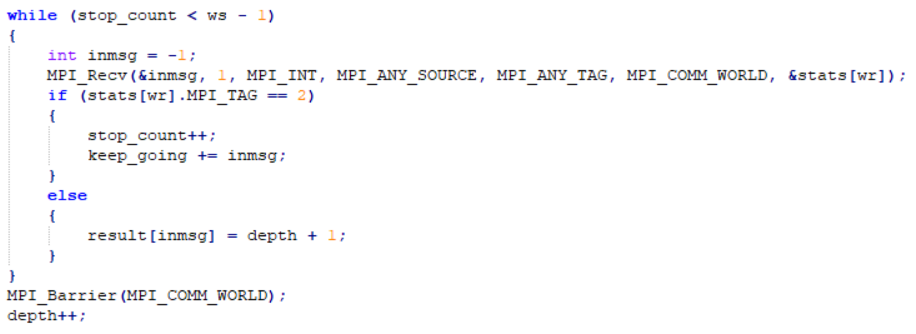
After the main loop, every processor lets others know it has finished updating the result array. Note that this message's tag is 2. The message contains the "keep_going" boolean which will be used by other threads.



```
for (int other = 0; other < ws; other++)
{
    if (other != wr) // 2 is stop tag
        MPI_Send(&keep_going, 1, MPI_INT, other, 2, MPI_COMM_WORLD);
}
```

Figure 2 Sending stop messages

Finally, each processor collects the incoming messages. If the message tag is 1, the result array is updated. Otherwise, it is the last incoming message with tag 2. We count the number of stop messages. If we receive world size – 1 stop messages, we finalize the iteration. We also update the "keep_going" when we receive the tag 2 messages to ensure that every processor finishes at the same time.

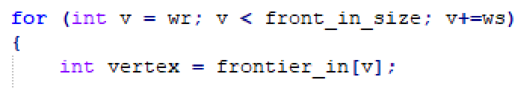


```
while (stop_count < ws - 1)
{
    int inmsg = -1;
    MPI_Recv(&inmsg, 1, MPI_INT, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COMM_WORLD, &stats[wr]);
    if (stats[wr].MPI_TAG == 2)
    {
        stop_count++;
        keep_going += inmsg;
    }
    else
    {
        result[inmsg] = depth + 1;
    }
}
MPI_Barrier(MPI_COMM_WORLD);
depth++;
```

Figure 3 Receiving the messages

Part 2 Frontier

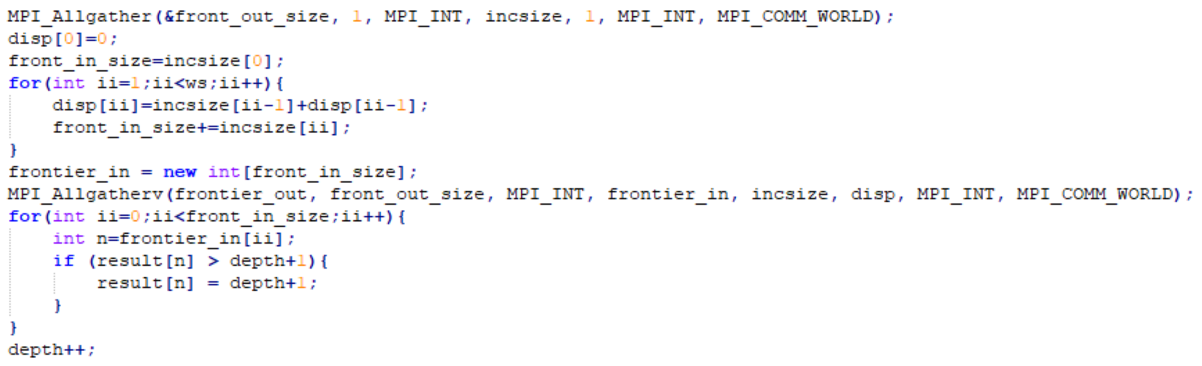
In this part, we implemented work efficient approach by dividing the frontier among processors. The division works similarly to part 1.



```
for (int v = wr; v < front_in_size; v+=ws)
{
    int vertex = frontier_in[v];
```

Figure 4 Main loop of frontier version

After each processor fills their frontier_out arrays, we allgather'ed the frontier_out arrays. To do that we needed to find the length of each frontier array. Since these lengths are not the same for each processor, we had to use "allgatherv" function to gather these frontiers. Finally, the result array is updated according to the accumulated frontier.

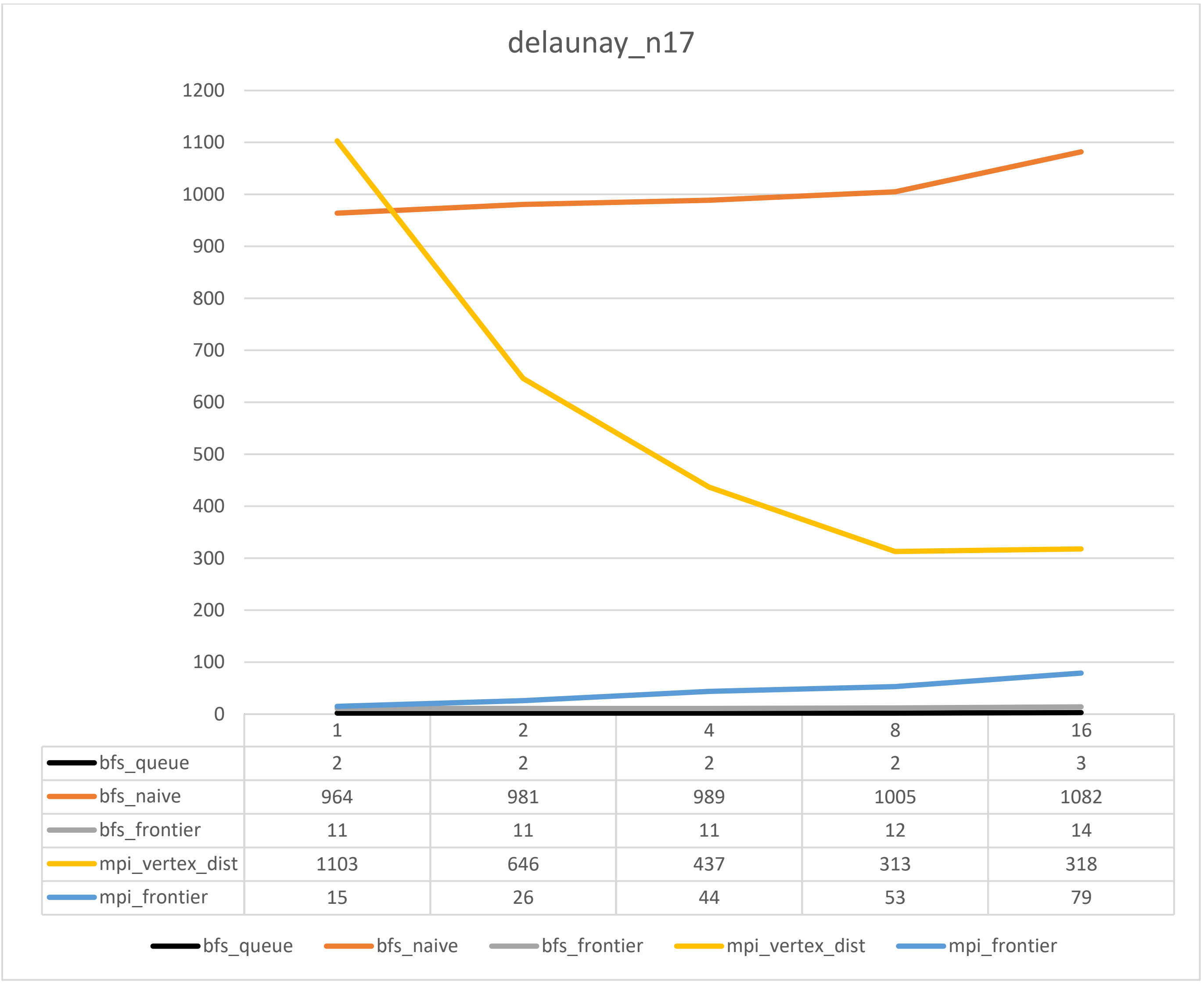


```
MPI_Allgather(&front_out_size, 1, MPI_INT, incsize, 1, MPI_INT, MPI_COMM_WORLD);
disp[0]=0;
front_in_size=incsize[0];
for(int ii=1;ii<ws;ii++){
    disp[ii]=incsize[ii-1]+disp[ii-1];
    front_in_size+=incsize[ii];
}
frontier_in = new int[front_in_size];
MPI_Allgatherv(frontier_out, front_out_size, MPI_INT, frontier_in, incsize, disp, MPI_INT, MPI_COMM_WORLD);
for(int ii=0;ii<front_in_size;ii++){
    int n=frontier_in[ii];
    if (result[n] > depth+1){
        result[n] = depth+1;
    }
}
depth++;
```

Figure 5 Gathering the frontiers

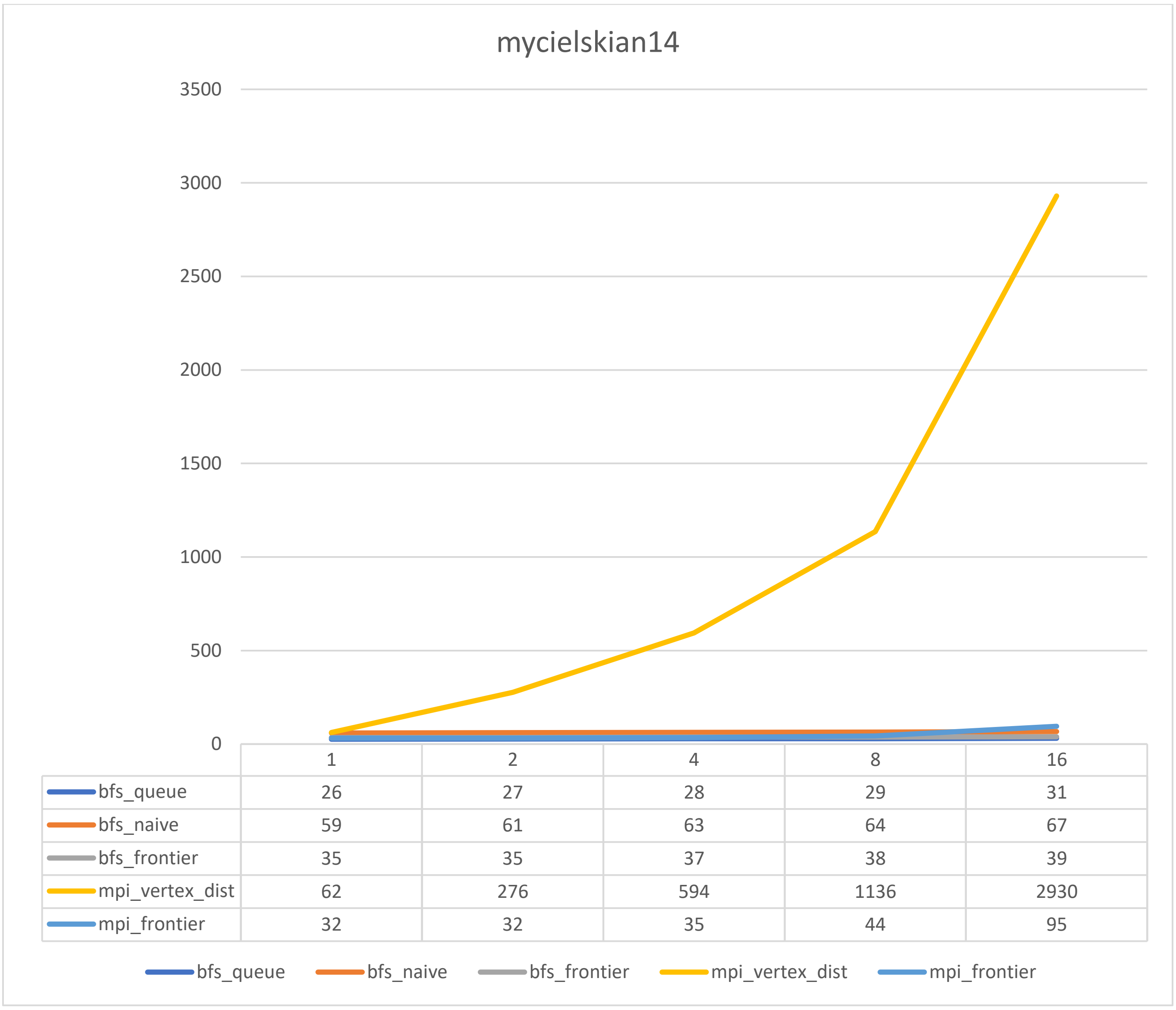
Experiments

Delaunay_n17.mtx



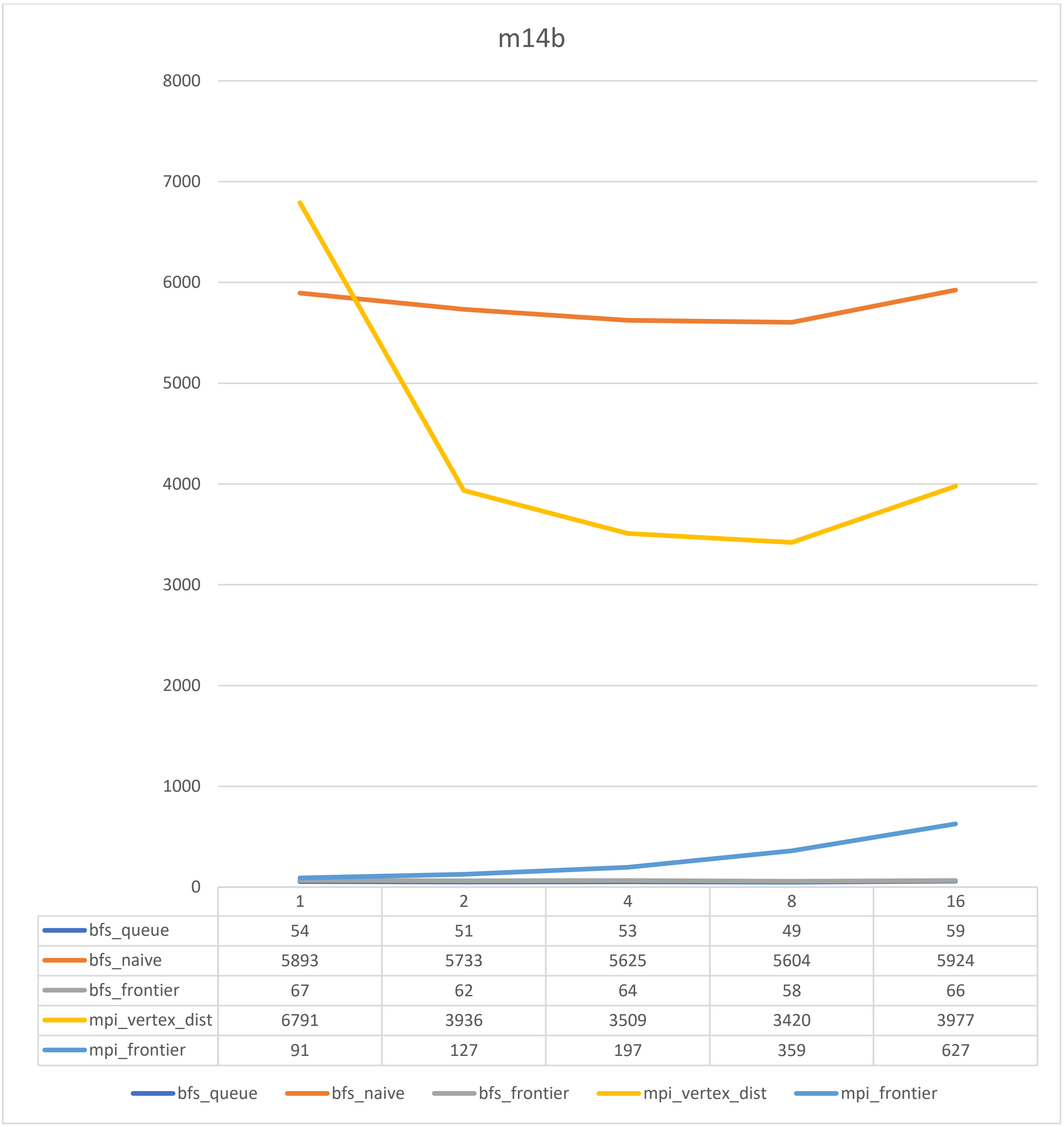
	1	2	4	8	16
bfs_queue	2	2	2	2	3
bfs_naive	964	981	989	1005	1082
bfs_frontier	11	11	11	12	14
mpi_vertex_dist	1103	646	437	313	318
mpi_frontier	15	26	44	53	79

Mycielskian14.mtx



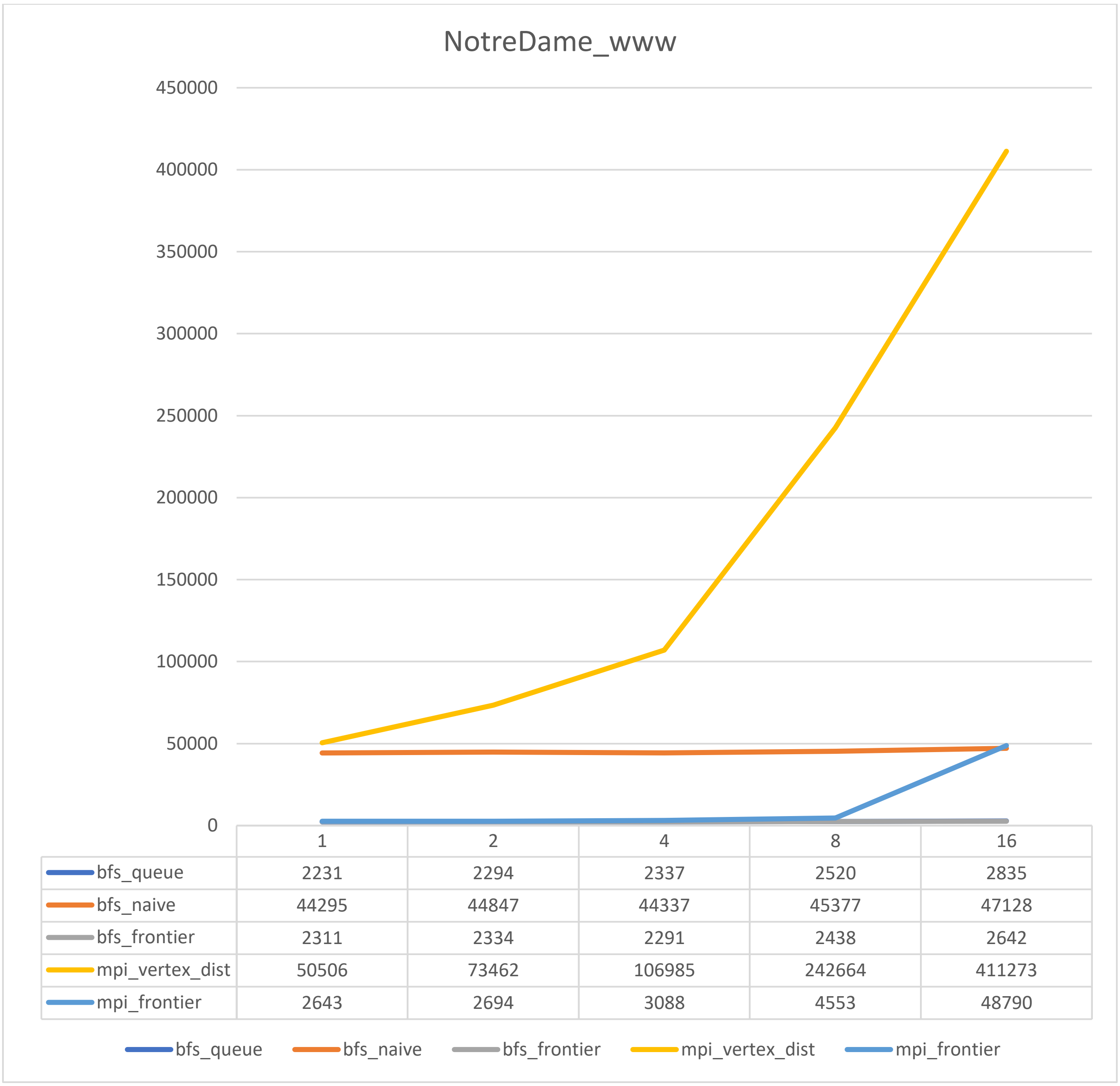
	1	2	4	8	16
bfs_queue	26	27	28	29	31
bfs_naive	59	61	63	64	67
bfs_frontier	35	35	37	38	39
mpi_vertex_dist	62	276	594	1136	2930
mpi_frontier	32	32	35	44	95

M14b.mtx



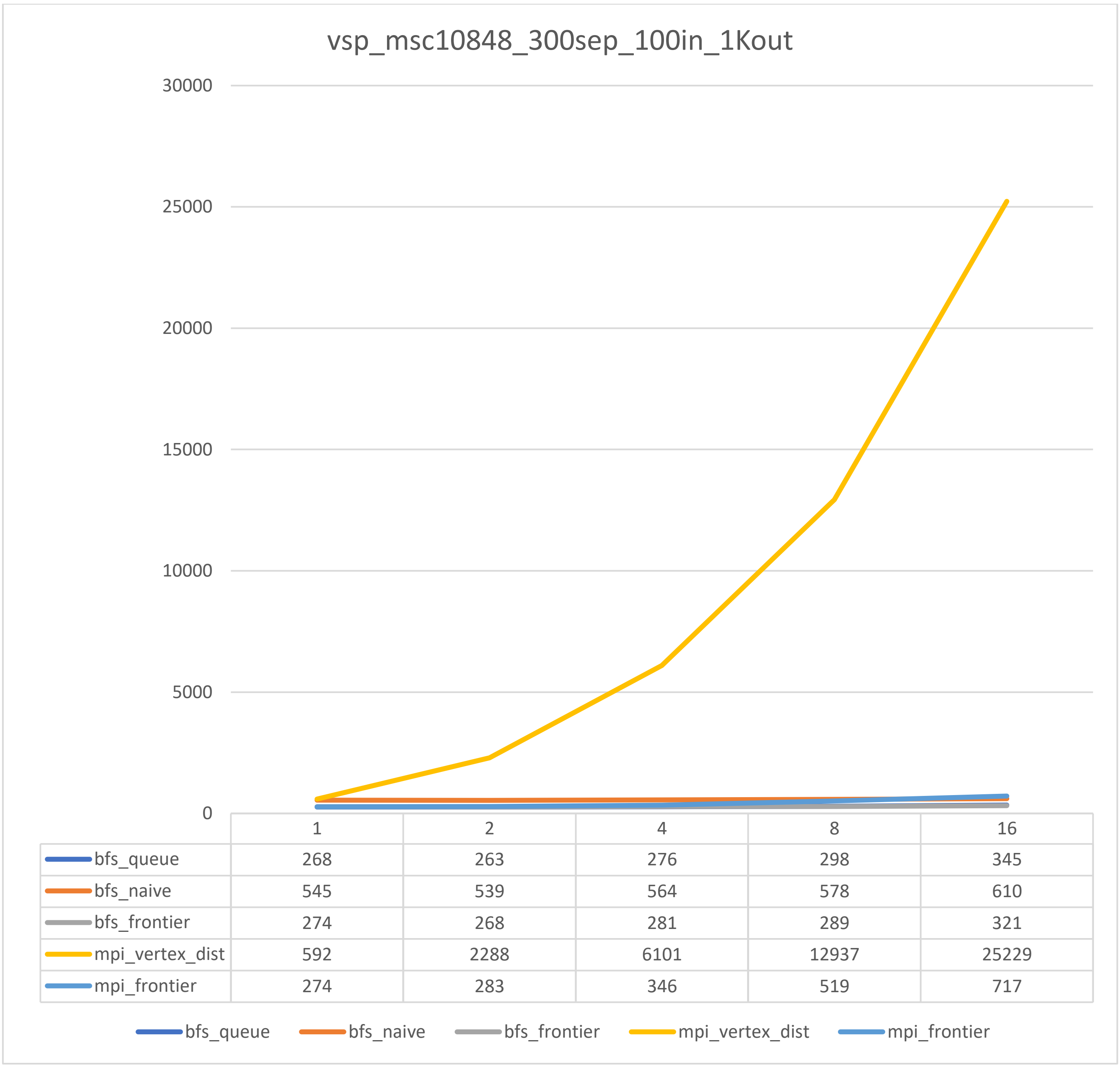
	1	2	4	8	16
bfs_queue	54	51	53	49	59
bfs_naive	5893	5733	5625	5604	5924
bfs_frontier	67	62	64	58	66
mpi_vertex_dist	6791	3936	3509	3420	3977
mpi_frontier	91	127	197	359	627

NotreDame_www.mtx



	1	2	4	8	16
bfs_queue	2231	2294	2337	2520	2835
bfs_naive	44295	44847	44337	45377	47128
bfs_frontier	2311	2334	2291	2438	2642
mpi_vertex_dist	50506	73462	106985	242664	411273
mpi_frontier	2643	2694	3088	4553	48790

vsp_msc10848_300sep_100in_1Kout.mtx



	1	2	4	8	16
bfs_queue	268	263	276	298	345
bfs_naive	545	539	564	578	610
bfs_frontier	274	268	281	289	321
mpi_vertex_dist	592	2288	6101	12937	25229
mpi_frontier	274	283	346	519	717

Conclusion

In conclusion, our parallelization implementation worked better for Delaunay and m14b graphs where it did not increase the performance for other graphs. We collected the characteristics of the graphs in the below table. It is evident that the degree of average standard deviation for Delaunay and m14b is very small. It means that all vertices have a similar number of edges. In both distributed approach (part 1) and the work efficient approach (part2), we are distributing the workload among the processes. The vertexes are distributed in part 1 whereas the frontier array is distributed in part 2. Therefore, our best performance comes from the slowest process. That's why, if graph edges are distributed homogenously among the vertices, our parallelization methods perform very well.

Graph	**#Vertices**	**#Edges**	**avg deg**	**deg stddev**	**median**
Delaunay	131072	393176	2,99	1,95	6
m14b	214765	1679018	7,81	4,81	13
NotreDame	253435	929849	3,66	6,07	5
vsp	21996	1221028	55,51	42,79	21
mycelskian14	12287	1847756	150,38	210,46	11