High Performance Computing

Parallel
Breadth-first search

Enrico Coluccia, 12005483
Dominik Schreiber, 01326110

1 Introduction

Graphs are a core part of most analytic workloads. Backed by a steering committee of over 30 international HPC experts from academia, industry, and national laboratories, the *Graph500* specification establishes a large-scale benchmark for these applications. This is the first serious approach to augment the Top 500 with data-intensive applications. [1]

The intent of benchmark problems ("Search" and "Shortest-Path") is to develop a compact application that has multiple analysis techniques (multiple kernels) accessing a single data structure representing a weighted, undirected graph. In addition to a kernel to construct the graph from the input tuple list, there are two additional computational kernels to operate on the graph.

This benchmark includes a scalable data generator which produces edge tuples containing the start vertex and end vertex for each edge. The first kernel constructs an undirected graph in a format usable by all subsequent kernels without subsequent modifications. The second kernel performs a breadth-first search of the graph. The third kernel performs multiple single-source shortest path computations on the graph. All three kernels are timed.

1.1 Setup

The experimental results were obtained on the *Hydra* Linux Server, a 36-node x 32-core Intel Skylake OmniPath cluster, with a total number of 1152 cores, provided by the Technical University of Vienna. The system uses SLURM as jobs scheduler in the main node. The MPI version that we used to perform our tests and benchmarks is OpenMPI ver. 3.1.3 SLURM. The created files are attached with this document and can be easily compiled with the *make* command.

[1]https://graph500.org

2 Exercise 1

The reference implementation of the Graph500 project starts by generating an undirected graph. In the first step, it generates tuples that represent edges of the graph, which are then converted and stored in memory using arrays, in compressed sparse row (CSR) notation. The graph is distributed between the processes using a 1D, cyclic distribution: all edges incident to vertex v are stored on rank (v % size).

Per default, it runs 64 iterations of the BFS and SSSP algorithms respectively, all starting with a different randomly chosen start vertex, with additional validation after every run. For performance reasons, the process count per node is restricted to be a power of 2, but the compilation can be customized with the following macros:

macro	**effect**
SSSP	build code which runs BFS and then SSSP
REUSE_CSR_FOR_VALIDATION	reuse reference CSR generated by kernel 1 for validation
DEBUGSTATS	gives information about the traversed graph levels
PROCS_PER_NODE_NOT _POWER_OF_TWO	allows process per node to be an arbitrary number (leads to performance decrease of $\sim 20\%$)

Table 1: Makros for compiling customizations

Additionally, there are several environment variables to customize the runtime behaviour of the binary:

environment variable	**effect**
SKIP_BFS	Only run SSSP algorithm
SKIP_VALIDATION	Skipps the validation after each run
REUSEFILE	reuse previously generated graph
VERBOSE	print information during runs
TMPFILE	points to a filesystem that is globally accessible and consistent on all ranks and has enough storage space for the graph data

Table 2: Environment variables to influence runtime behaviour

The final binary also takes parameter arguments, which allow the specification of the problem instance:

1. scale = $\log_2$(number of vertices)
2. edgefactor = (number of edges) / (number of vertices)
   default value is 16

As performance metrics, the two kernels compute some statistics about the running time and the TEPS. In particular, we considered for all our measures the mean time (for both BFS and SSSP) but other metrics are presented e.g. standard deviation, min/max time, median time etc . . .

The TEPS (traversed edges per second), as stated and presented in the specification, is a new kind of rate to evaluate the performance of distributed applications that deal with graphs. The measure is defined as $TEPS(n) = m/time_k(n)$. Where $time_k(n)$ is the measured execution time for a kernel run and m is the number of undirected edges in a traversed component of the graph counted as number of self-loop edge tuples within component traversed added to halved number of non self-loop edge tuples within component traversed. We did not consider this new performance in this report.

2.1 Breadth-First Search

Figure 1 shows the results for the reference implementation of the graph500 project, for the BFS algorithm for different graph scale factors and process distributions. The edge factor was left at the default value of 16 for all runs. The exact numbers in seconds are given in Table 6. The test runs for scale factor 29 only completed, when using 32 nodes with 32 processes. The other runs for scale factor 29 timed out before a single instance was finished.

Most interesting is the fact, that the algorithm performs much ($\sim$ 1.5 times) faster on 32 nodes with each running only one process, than when using 1 node running 32 processes. This can have multiple reasons. Either the algorithm needs more memory bus performance than is available on the system, which would make it memory bound or it is slower because of cache misses introduced by multiple processes overwriting each others cached values. In our scenario, it is most likely a cache issue.

2.2 Single Source Shortest Paths

Figure 2 shows the results for the reference implementation of the Graph500 project, for the SSSP algorithm for different graph scale factors and process distributions. The edge factor was left at the default value of 16 for all runs. The exact numbers in seconds are given in Table 3.

We can observe the same trend for the SSSP reference algorithm, as we did for the BFS reference algorithm, that distributing processes between nodes

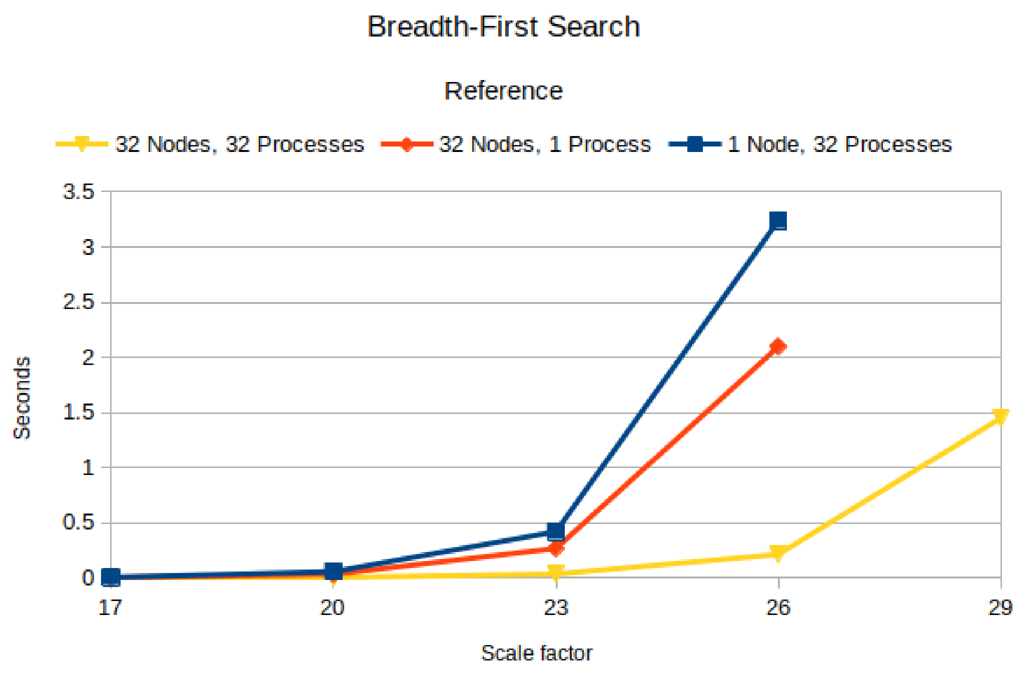

Figure 1: Performance of BFS algorithm.

is better than running all processes on a single node. The reasons for such a behaviour were already given in Section 2.1.

scale factor	1x32	32x1	32x32
26	8.25	5.43	0.51
23	0.98	0.65	0.086
20	0.1488	0.1072	0.0306
17	0.02	0.0183	0.0132

Table 3: SSSP results

2.3 Non-Power of two Processes

We compiled the reference implementation with the additional macro -*DPROCS_PER_NODE_NOT_POWER_OF_TWO*, to execute it with arbitrary numbers of processes. For every run, 32 compute nodes were used, while the process number per node and scale factor were altered. An overview of the results is depicted by Figures 3 and 4.

The documentation states, that using process counts which are not powers of two, can lead to performance degradation of approximately 20%. This is the

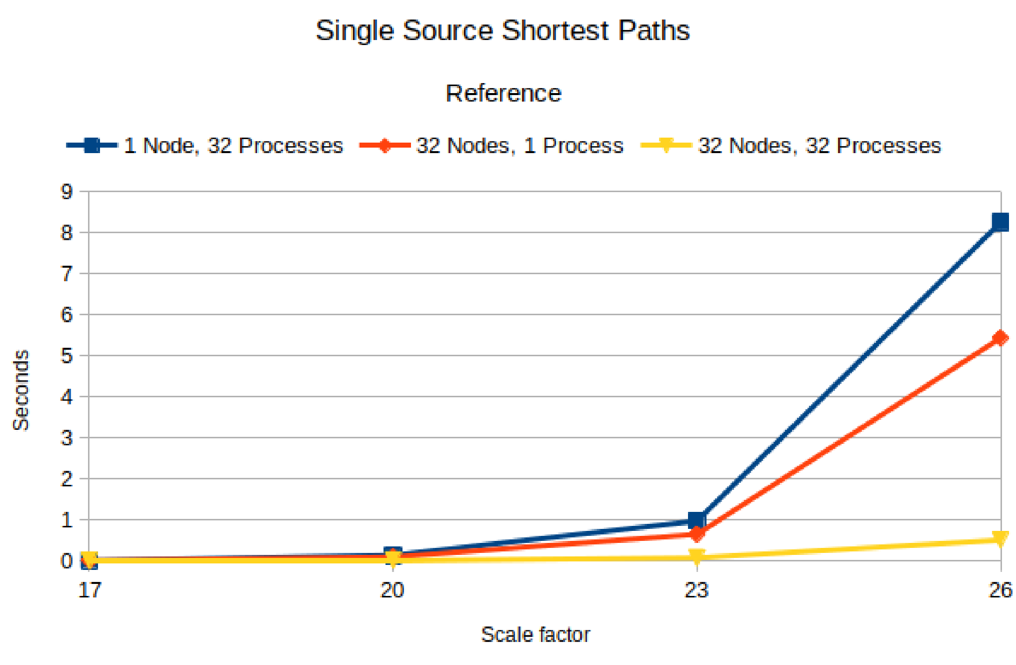

Figure 2: Performance of SSSP algorithm.

scale factor	4	6	8	12	16	24	32
17	0.0047	0.0043	0.0038	0.0034	0.0033	0.0035	0.002
20	0.0265	0.02	0.0164	0.0132	0.0127	0.0111	0.0079
23	0.186	0.138	0.1076	0.0824	0.0702	0.0566	0.037
26	1.311	1.030	0.7823	0.5708	0.446	0.3395	0.214

Table 4: BFS results with different process counts on 32 nodes.

case because the graph has a power of two SCALE (number of nodes) and thus the vertices are distributes across the processes unequally.

For test runs with small instances with a scale factor below 23 and process counts smaller than 16, there is not really any measurable degradation in performance when using non power of two process counts. There are even cases, where non-power of two process counts have scaled better or the execution time was longer with more processes than with less (e.g. BFS with scale factor 17, using 12, 16 and 24 processes). This is most likely because at this small scales, variances in the cache, operating system and process creation time have stronger effects on the execution time, than the algorithm itself.

For larger instances and larger amounts of processes on the other hand, we can observe better scaling for process counts that are a power of two, like it is expected from the documentation, for both BFS and SSSP. Still even for the

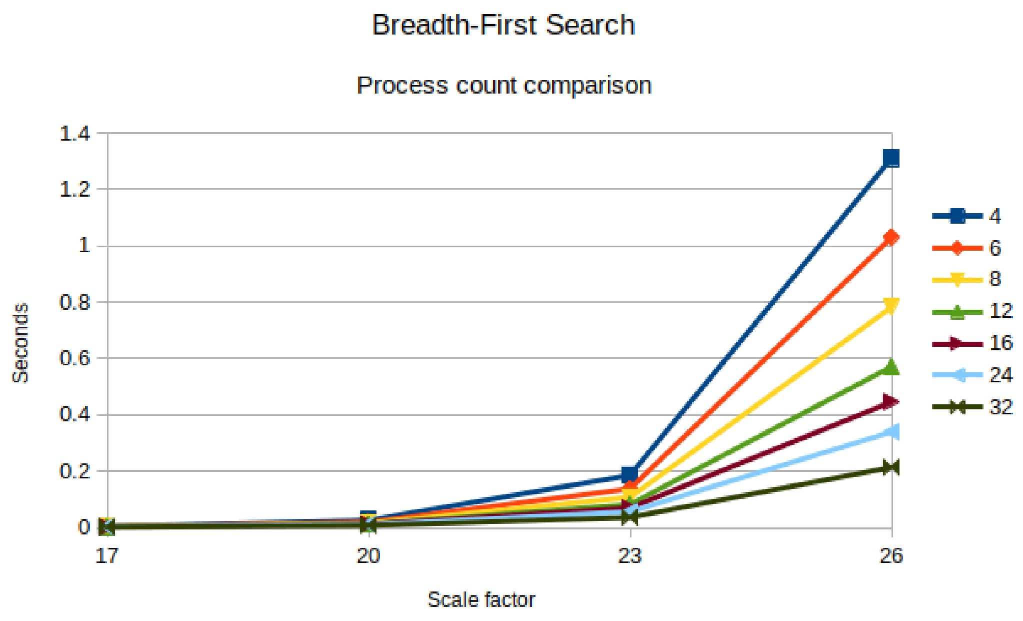

Figure 3: Performance of BFS with different process counts.

scale factor	4	6	8	12	16	24	32
17	0.0128	0.0124	0.0125	0.012	0.0119	0.0134	0.0132
20	0.0514	0.0416	0.0375	0.0317	0.0309	0.0293	0.0306
23	0.3631	0.262	0.2036	0.16	0.1388	0.1157	0.0864
26	2.9328	2.0737	1.6355	1.2909	0.9684	0.7522	0.5164

Table 5: SSSP results with different process counts on 32 nodes.

largest instance of our test set (scale factor 26), the performance increase was only around 10% better, when using process counts that are a power of two. The difference will probably be more noticeable on larger graph instances.

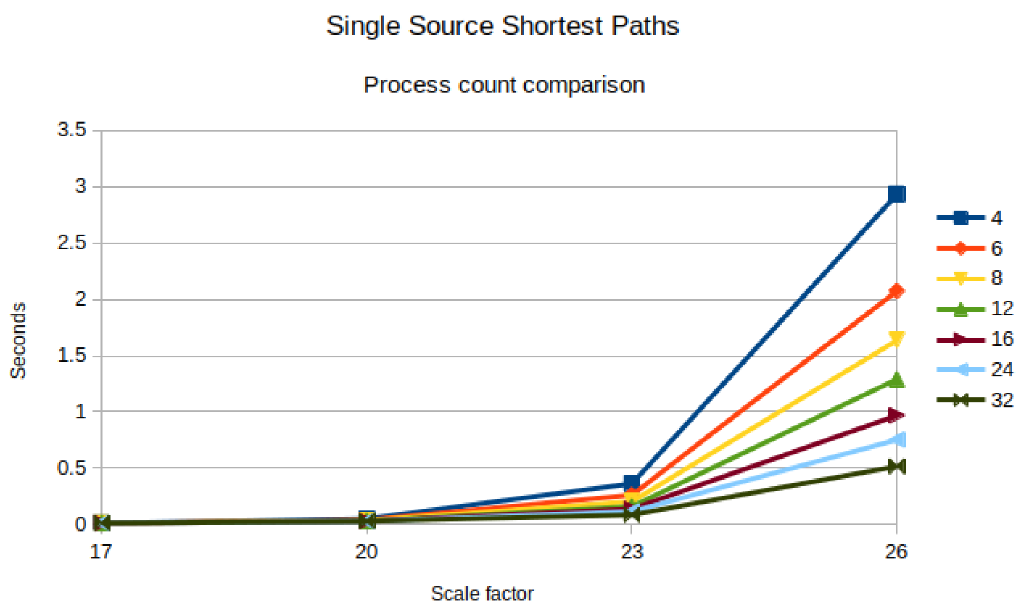

Figure 4: Performance of SSSP with different process counts.

3 Exercise 2

As recommended in the assignment description, we used the graph structure from the reference implementation for our algorithm, which allowed us to reuse the graph generation, performance measurement, and validation code. To plug our implementation into the reference framework, we used the "bfs_custom.c" template, that was provided with it. We also took advantage of the convenient macro definitions from the reference.

Additionally to the global variables used by the reference implementation, our implementation defines the following global variables for managing the inter process communication over MPI:

```c
// Integer array, where entry i specifies the number of sendmsg
// structs (vertices) to send to rank i.
int *send_counts;
// replication of the array from above
int *send_counts_2;
// Integer array, where entry i specifies the displacement (offset from
// sendbuf, in units of sendtype) from which to send data to rank i.
int *send_disps;
// Array of visitmsg struct from reference implementation, extended by
// also containing the rank of the sending process, acting as send buffer.
struct visitmsg* send_buf;
// Integer array, where entry j specifies the number of sendmsg structs
// to receive from rank j.
int *recv_counts;
// Integer array, where entry j specifies the displacement (offset from
// recvbuf, in units of recvtype) to which data from rank j should be written.
int *recv_disps;
// Array of visitmsg struct from reference implementation, extended by
// also containing the rank of the sending process, acting as receive buffer.
struct visitmsg* recv_buf;
// Size of the mesasge to send.
long send_size;
// Size of the message to receive.
long recv_size;
```

Our algorithm starts similar to the reference implementation, by marking the root node as visited and its predecessor as the root node itself, since this is required for the validation at the end.

After that we start an infinite while loop, which starts by initializing and zeroing the global arrays, that are used for the MPI all-to-all communication.

```c
void initialize_list(){
        // count the total number of vertices that need to be sent to each process
    // store the number of vertices that need to be sent to each process
    send_counts = (int *)malloc(sizeof(int) * size);
    send_disps = (int *)malloc(sizeof(int) * size);
    // inform each process, how many vertices to expect in the next communication
    recv_counts = (int *)malloc(sizeof(int) * size);
    recv_disps = (int *)malloc(sizeof(int) * size);
    for(int i=0; i<size; ++i){
        recv_counts[i] = 0;
        recv_disps[i] = 0;
        send_counts[i] = 0;
        send_disps[i] = 0;
    }
    recv_size = 0;
    send_size = 0;
}
```

Following that, we prepare the *send_counts messages*, which contain the number of vertices, each process should expect to receive, to calculate paths for the next level.

```c
for(i = 0; i < qc; i++){
        for(j = rowstarts[q1[i]]; j < rowstarts[q1[i] + 1]; j++){
            send_counts_2[VERTEX_OWNER(COLUMN(j))]++; //fill the send count size;
            send_size++;
        }
}
```

Then we generate the messages containing the actual vertices, which are going to be distributed over all processes. Every process rank gets all the vertices of the current level, that he is the owner of.

```c
//create a buff of send_size so that we can fill it with the messages
send_buf = (struct visitmsg*)calloc(send_size, sizeof(struct visitmsg));
for(i = 0; i < qc; i++){
    for(j = rowstarts[q1[i]]; j < rowstarts[q1[i] + 1]; j++){
        int offset = 0;
        int owner = VERTEX_OWNER(COLUMN(j));
        for(int l = 0; l < owner; l++){
            offset += send_counts_2[l]; //calculate the offset in which we
            ↪  set the message
        }
        visitmsg m = {VERTEX_LOCAL(COLUMN(j)), q1[i], rank};
        send_buf[(send_counts[owner] + offset)] = m;
        send_counts[owner]++; //fill the send count size;
    }
}

for(i=0; i < size; ++i){
    if(i == 0){
        send_disps[i] = 0;
    }
    else {
        send_disps[i] = send_disps[i-1] + send_counts[i-1];
    }
}
```

We use the MPI function *MPI_Alltoall()* to distribute the message counts between all processes, so each process knows how much memory to allocate for the receive buffer.

```
//what we will expect to receive from each processors?
MPI_Alltoall(send_counts, 1, MPI_INT, recv_counts, 1, MPI_INT,
↪  MPI_COMM_WORLD);
```

After every process has received it's message/vertex counts, we can allocate the necessary memory and distribute the actual messages describing the vertices that should be visited next, using the MPI function *MPI_Alltoallv()*. With this function, we can send messages of different sizes to all processes.

```
recv_buf = (struct visitmsg*)calloc(recv_size, sizeof(struct visitmsg));

MPI_Alltoallv(send_buf, send_counts, send_disps, mpi_message_type, recv_buf,
↪  recv_counts, recv_disps, mpi_message_type, MPI_COMM_WORLD);
```

Now that every process knows which vertices he has to visit, we can perform the actual test, if the vertex was already visited before. If not we set its state to visited, add it to q2 and enter its global position into the array, containing the predecessors of each visited vertex.

```
for(i = 0; i < recv_size; ++i){
    if (!TEST_VISITEDLOC(recv_buf[i].vloc)) {
        SET_VISITEDLOC(recv_buf[i].vloc);
        q2[q2c++] = recv_buf[i].vloc;
        pred_glob[recv_buf[i].vloc] = VERTEX_TO_GLOBAL(recv_buf[i].ra,
        ↪  recv_buf[i].vfrom);
    }
}
```

At the end of the loop, we synchronize all processes using *MPI_Barrier()*. Further we swap the two arrays holding visited VERTEX_LOCALs for current and next level (q1 and q2), similar to the reference implementation and check if there are still vertices, that have not been visited. If global_sum is 0, we have covered all vertices and break out of the while loop to end the execution.

```
MPI_Barrier(MPI_COMM_WORLD);

qc = q2c;
int *tmp = q1;
q1 = q2;
q2 = tmp;
sum=qc;
global_sum = 0;
MPI_Allreduce(&sum, &global_sum, 1, MPI_LONG_LONG, MPI_SUM,
↪  MPI_COMM_WORLD);

if(global_sum == 0){
    break;
}

nvisited += sum;
q2c = 0;
```

3.1 Collectives & Data Structure

The collective operations that we used were MPI_Allreduce, MPI_Alltoall and MPI_Alltoallv. The allreduce operation reduces the values and distributes the results to all processes; in our case each process has to know if there is still work to do and we performed a reduction on the size of the current level. In the alltoall operation all processes send the same amount of data to each other, and receive the same amount of data from each other; we used this collective operation to inform each processor about the data that it has to expect to allocate the memory.

With the alltoallv collective we performed the crucial step in our algorithm: we send and receive the the information about the current level. A visualization of this collective operation is presented in Figure 5. Each process has information about the other processes, these information has variable lengths; at the end of the operation each process has the information about its vertices in the current level.

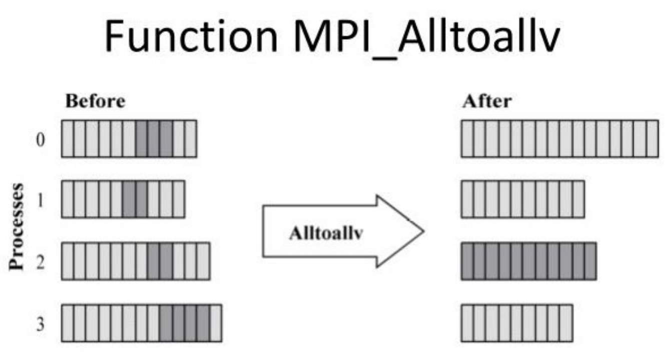

Figure 5: MPI_Alltoallv visualization.

Last but not least, we used a custom data structure to fill the send and recv buffers. We utilized the same struct proposed in the reference implementation with the addiction of an integer about the sender rank. This because we need the sender information to use the VERTEX_TO_GLOBAL macro in order to fill the pred list.

```c
typedef struct visitmsg {
        int vloc;
        int vfrom;
        int rank;   // send rank
} visitmsg;
```

Then we create the MPI structure with MPI_Type_create_struct and we committed the new type with MPI_Type_commit.

3.2 Perfomance Comparison

In this chapter we present the results obtained for the proposed MPI collective implementation. From table 6 we can see all the runtimes compared with the reference implementation, for different scales and for different nodes/tasks. It is immediately clear, from the obtained results, that the collective implementation did not beat the reference implementation in any of the proposed settings.

scale factor	type	1x32	32x1	32x32
17	custom	0.0102	0.009	0.025
	reference	0.007	0.005	0.002
20	custom	0.066	0.061	0.049
	reference	0.058	0.035	0.0079
23	custom	0.513	0.44	0.144
	reference	0.42	0.267	0.036
26	custom	3.884	3.42	0.654
	reference	3.239	2.097	0.21
29	custom	-	-	3.919
	reference	-	-	1.452

Table 6: Collective Custom vs Reference BFS results, time (s)

From the Plot 6 we can see the performances over 1024 core (32x32) and it is clear, that the proposed implementation starts to be very computational expensive after a certain SCALE factor. In particular, for SCALE 29 the runtime was twice as bad as the reference implementation (a factor of ~ 2.7). This exponential trend is probably caused by the double loop that has to be performed during the level analysis that is highly influenced by the number of vertices that has to be analyzed.

From Scatter Plot 8 and 7 we can do a comparison between the performances on 32x1 cores and 1x32 cores. Here we can see a very good performance on 1x32 cores, the graph shows that the trend is quite similar for both implementations. Of course with increasing SCALE factor, the drawback mentioned before is clearly noticeable; for SCALE 26 the collective implementation performs ~ 1.18 times worse than the reference implementation. With 32 nodes the situation does not change. For SCALE 26 we see that the collective solution is worse by a factor of ~ 1.75, compared to the reference implementation. From the graphs we can also notice how the algorithms perform better over 1 node with 32 cores than 32 nodes with 1 core. The reasons of this phenomena are explained in Chapter 2.

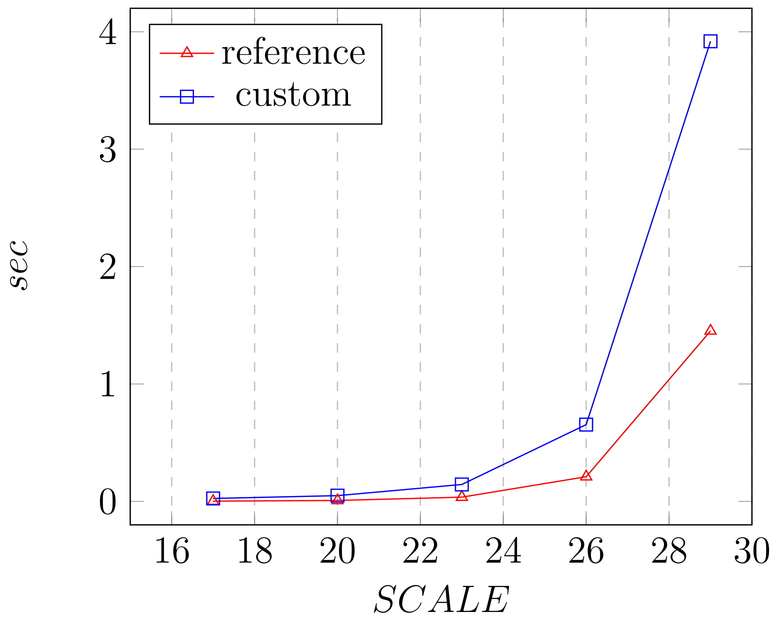

Figure 6: Running time comparison 32x32

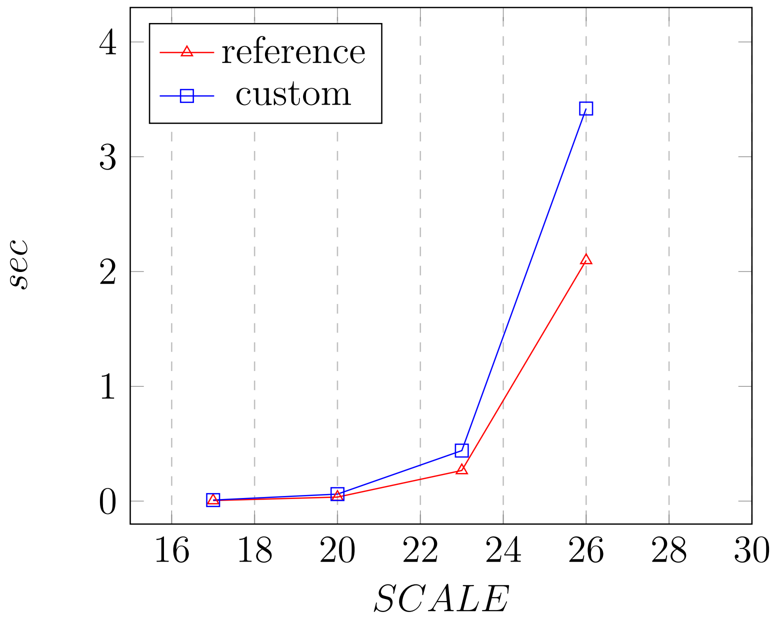

Figure 7: Running time comparison 32x1

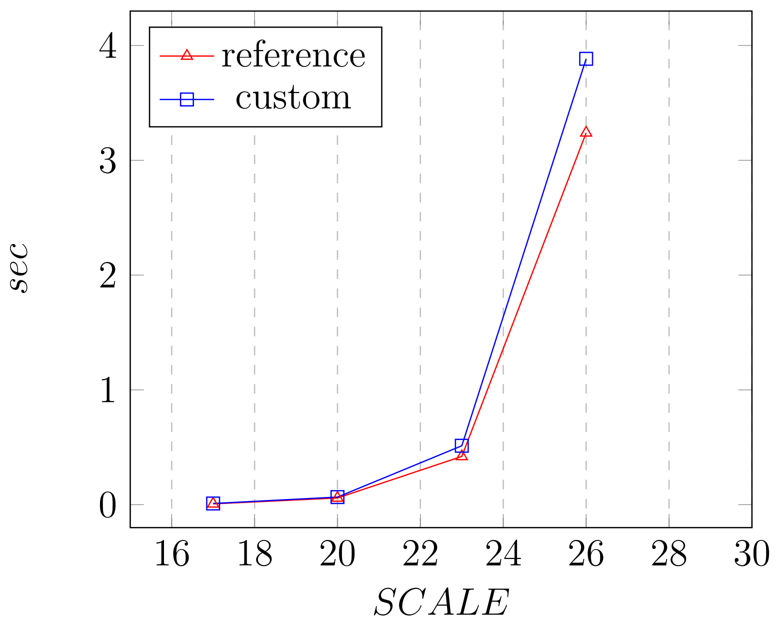

Figure 8: Running time comparison 1x32

4 Conclusion

In this paper we analyzed the *Graph500* benchmark and ran different test instances with various configurations on a 36-node x 32-core Intel Skylake cluster, with a total number of 1152 cores, provided by the Technical University of Vienna.

We could observe different problems that can occur when using clusters for high performance computing. At the beginning Section 2, showed how using all available cores on a compute node can negatively impact the overall speedup because of hardware limitations, while Section 2.3 showed how the process count and its relation to the input size can influence the performance in different ways.

We also proposed a collective implementation by meaning of the MPI library for the Bread-First Search kernel. It was interesting to see how the reference implementation is hard to beat even with an implementation without point-to-point message. Moreover, we identified the possible reasons for this behaviour in the collective implementation and it is clear that we can reach better results with an improved version and a better level processing (e.g. with the help of other data structures).