TU WIEN Informatics

INSTITUTE OF COMPUTER ENGINEERING

HPC - Graph500

Member:
Alexander Leitner, 01525882
Mario Hiti, 01327428

Submission: Februar 1, 2021

Abstract

The Graph500 project is a good choice to test the MPI program skills and to get more experience in this section of education. The goal of this project is to understand the structure of the provided Graph500 code, to benchmark it and to implement a "custom BFS" code. The final part is to test the "custom BFS" on a cluster named "hydra" with configurations of 1x32, 32x1 and 32x32 nodes and processes.

Contents

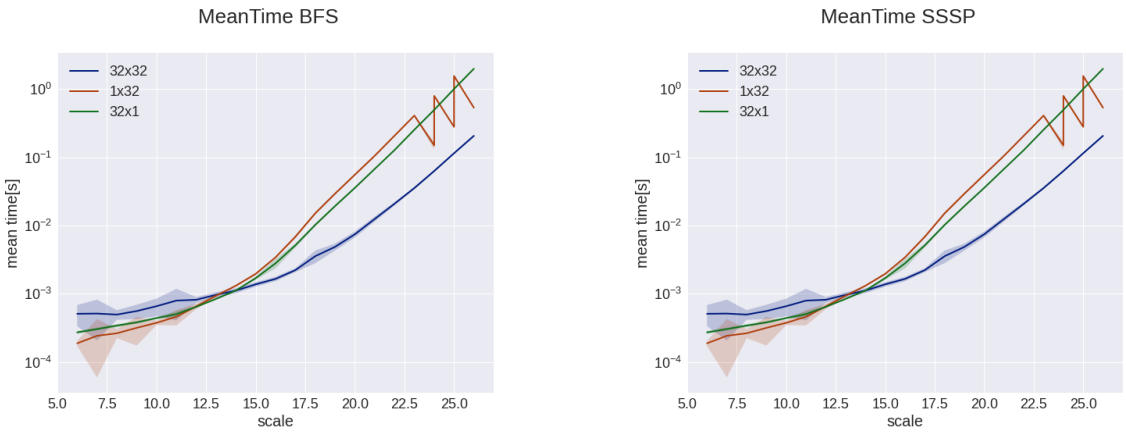
1 Introduction

The Graph500 is a huge project and also the choice to test and benchmark custom BFS (Breath First Search) and custom SSSP(Single Source Shortest Path) to compete against other groups in the world. Our exercise was first to benchmark the provided code and then implement a custom BFS code with MPI. For the custom BFS we first tried to code a serial implementation and then tried to paralyze it with MPI.

2 Provided code benchmark

The scale-factor represents the number of nodes of the graph. For example a scale of 10 means $2^{10} = 1024$. For all plots we chose a range of scale-factor from 6 to 26. We chose that range because we wanted to see the point where the performance of the parallel implementation wins against the serial implementation. One run consists of starting randomly at 64 nodes, measures its runtime and provides automatically some statistics from that run.

2.1 graph500 reference benchmark



(a) Runtime with standart deviation

(b) Runtime with standart deviation

Figure 1: 2 Figures side by side

On the left side we could see the plotted meanTime to the corresponding scale-factor. The shadow area in the lower part of the scale-factor represents the standard deviation of the certain scale-factor. This area is larger for a small scale-factor. That indicates that the measured runtime changed more with a different root for a smaller graph. For example the 32x1 configuration means that the program runs on the cluster on 32 nodes with 1 kernel. Each of these kernels has only one thread. On the left side of the cross-section the 32x32 setting is the slowest, followed by the 32x1 and the fastest is the 1x32 setting. With rising number of the scale-factor the 32x32 configuration is the fastest due to the high number of nodes with a high number of kernels. For the 1x32 configuration there are some pikes in the upper range of the scale-factor. The same tendency is also shown in the other graphs.
On the right side the SSSP is plotted. The time is a bit higher in average than the pure BFS algorithm. Due to the different algorithm the SSSP algorithm needs the BFS. There is also a cross-section for the three lines due to the communication time between the processes. All in all the 32x32 implementation is the fastest. For these two algorithms it seams that for a high number

of the scale-factor the communication is faster between multiple nodes than the communication between multiple threads on one node.

3 The BFS problem and project workspace

3.1 BFS (serial) algorithm

As a first step we implement a commonly used BFS algorithm [Lóp17, p. 34 Alg. 1] which we then started to parallelize.

Algorithm 1: Serial BFS algorithm

```
1  Global: array rowstart, array pred_glob
2  Input: root (start node), array pred, queue Q, std::vector vis
3  Output: array pred_glob integers with visited nodes
4  Q ⟵ root; vis[root] ⟵ true
5  while(Q is not empty)
6      u ⟵ Q.front() // current element in Q
7      u ⟵ Q.front() // current element in Q
8      for j in rowstart
9          v ⟵ COLOUMN(j) // COLOUMN Marko to indicate which vertex
10         if vis[v] is false
11             vis[v] ⟵ true
12             Q.push(v)
13             pred_glob[v] ⟵ u
14         end if
15     end for
16 end while
```

3.2 Code structure and C++ integration

Initially we tried to implement our solution in C to match the language of the graph500 framework. However even basic data structures are very tedious to implement and memory management is very prone to errors.

Therefore we have written our solution with C++ to make use of STL containers and cleaner syntax. Unfortunately the graph500 makefile is very convoluted so we created our own for easier maintenance. Furthermore we removed some redundant files from the reference implementation such as files for SSSP or the BFS-reference algorithm. Therefore in order run the reference solution the corresponding repository needs to be downloaded and built separately from `https://github.com/graph500/graph500`. A repository for our code can also be found at `https://github.com/hitimr/HPC`.

Our serial and parallel implementation can be found in *bfs.cpp*. Due to some compiler errors we were unable to include various graph500 headers in C++ Code. Therefore we had to copy some definitions and macros into that file. Copied code is marked using comments.

The configuration file *config.h* can be used to toggle various improvements described in Section 4.2.

3.3 Project configuaration

The following optimizations can be turned on or off in *config.h*:

BFS_PARALLEL/BFS_SERIAL Switch between serial and parallel implementation. Only one option can be selected

USE_OMP Use OMP SIMD to speed up loops. For more information see Section 4.2.4

USE_TESTVISIT_FAST Use advanced bit testing functionality. For more information see Section 4.2.3

DO_NOT_SEND_LOCAL_DATA Some data does not need to be sent through the communication interface and can be processed locally. When turned off all data is sent via aml. This is useful for debugging on a single process

TAG_POOL_DATA Initially we used different types of messages (tags). The final implementation only uses one tag so changing this option has no affect

AML_MAX_CHUNK_SIZE Defines the maximum size of aml messages. Smaller numbers result in smaller packets but more transmissions. Values above 4095 require changes to the aml interface. For more information see Section 4.2.1

TEST_VISITED_EMPTY_CUTOFF Relative number of visited vertices after which the bit testing function is replaced. I.e.: 0.2 means that after 20% of vertices have been marked as visited the bit testing function is changed. For more information see Section 4.2.3

Note that those configurations are applied by using preprocessor directives. Deactivated features are not included in the final binary to maximize performance. Changing the configuration requires rebuilding the project.

3.4 Project configuration and build process

To download and build the project run the following commands in your favorite shell:

```
1 git clone github.com/hitimr/HPC
2 cd HPC
3 module load mpi/openmpi-3.1.3-slurm # required on the hydra cluster
4 make
5 make run
```

To run the benchmark use one of the following commands

```
1 make run    # launch with 2 processes, scale = 18, edgefactor = 16
2 cd ./build/bfs [SCALE] [EDGEFACTOR] # manually specify scale and edgefactor
```

3.5 Output file

To get the results in a compact file we implemented also a file stream to get all different results from every setting.

4 BFS parallel implementation

The following chapter will describe the structure of our solution. The first part covers the concepts that we have reused from the reference implementation, followed by a basic description on how we restructured the code in order to try to achieve better results.

The second part will go into more detail on how we solved certain problems and various optimizations we used. At the end of each subsection we also provide some ideas for further improvement.

4.1 Basic structure

4.1.1 Reference Implementation

While every part of our BFS implementation was written by ourselves we used several concepts from the reference code and adapted them for use in our solution. Furthermore the data structures for the graph and bit array have been reused.

The main idea of the reference implementation is to use two separate queues for each process to avoid race conditions. Furthermore the adjacency matrix and bit array is evenly split amongst all processes, therefore some calculations can only be performed on a specific rank. The first queue `q_work` is used similar to the one in the serial implementation described in Section 1 but every vertex is sent through a communication interface (aml) to be processed by the responsible rank. Once processed unvisited vertices are added to a separate queue `q_buffer`. Once `q_work` is empty a barrier is used to synchronize all processes and then the queues are swapped. This process is repeated until no vertices are left in any queue including queues from other processes. This ensures that only direct neighbours are processed thus avoiding any race conditions.

Alg. 2 shows the basic structure of the graph500 reference. As the number of iterations of the for-loop is equal to the number of non-zero entries of the a given row in the adjacency matrix, this loop is a good place to start looking for improvements.

Algorithm 2: graph500 pseudocode

```
1  q_work, q_buffer, vis ⟵ 0
2  if rank == owner(root)
3      enqueue(q_work, root)
4  endif
5  do
6      while(! q_work == 0)
7          u = dequeue(q_work)
8          for v in Adj[u]
9              send_visit(v, u)    // check visit and set bit array on responsible
                   process
10         end for
11     end while
12     barrier
13     swap(q_work, q_buffer)
14 end do(all q_work == 0) // repeat as long as there are any vertices left globally
15
16 recieve_visit(v,u)  // callback routine performed on responsible core
17     // identical to BFS serial
18     if vis[v] is false
19         vis[v] ⟵ true
20         enqueue(q_buffer, v)
21         pred_glob[v] ⟵ u
22     end if
```

4.1.2 Own implementation and aml usage

The first problem we identified was that vertices where `owner(u) == owner(v)` do not need to be sent through the aml interface as they can be processed locally.

Algorithm 3: filtering out local vertices. Doing it this way actually makes things worse

```
for v in Adj[u]
    if owner(v) != rank
        send_visit(v, u)    // check visit and set bit array on responsible
            process
    else
        check_visit_local(v,u)  // check visit immediately on the same process
end for
```

However inserting a single branch within the loop as shown in Alg. 3 resulted in a big hit in performance even on the smallest possible system (2 cores) where `check_visit_local()` has a 50% chance of being called. This further emphasizes how important this loop is.

Another problem we found is the fact that for every iteration the same vertex u is sent thus sending significantly more data than actually necessary. One approach would be to only send u once and then continue to send v until a new vertex u is de-queued. Additionally every call of `send_visit` creates some overhead as it needs to be processed by the aml/mpi interface.

In order to combat the problems described above we decided to implement a pooling system. The main idea is to have a pool for every process on every process. Instead of calling `send_visit` immediately we add v to the respective pool and send the complete pool once the loop is complete. Furthermore we now can quickly identify the pool that can be processed locally without the need of repetitive checks.

Algorithm 4: Basic pooling concept

```
q_work, q_buffer, vis, pools ⟵ 0

if rank == owner(root)
    enqueue(q_work, root)
endif
do
    while(! q_work == 0)
        u = dequeue(q_work)
        for v in Adj[u]
            append(pools[owner(v)], v)
        end for
        for p in pools
            if owner(p) == rank
                process_pool_local(p)
            else
                send_pool(p)    // send pool to responsible processor
            end if
            clear(p) // set pool as empty
    end while
    barrier
    swap(q_work, q_buffer)
end do(all q_work == 0) // repeat as long as there are any vertices left globally
```

Alg. 4 shows the basic structure of our pooling approach. The pools are implemented using `std::vector<std::vector>` and every process has pools equal to the total number of processes.

One problem with this approach using the MPI Interface is that the receiver has no information about the actual size of the pool. However MPI does not support straighforward way of receiving objects of unknown size. One way of resolving this issue is to send a header with fixed size, containing the pool length, to tell the receiver how much data has to read from the buffer.

Luckily this is exactly what aml does which is why we chose to use it for our implementation. Alg. 5 shows a basic example of aml usage. Please note that we could not find any documentation on the aml interface. Thus all information presented here reflects our own understanding of the interface after studying the provided source code.

Algorithm 5: Example AML usage

```
void sender(vector<int64_t> & pool, dest) {
    aml_register_handler(analyze_pool, TAG_POOLDATA); // only required once
    aml_send(&pool[0], TAG_POOLDATA, sizeof(pool), dest); // send message
    aml_barrier();  // messages are processed within the barrier
}

void analyze_pool(int from, void* data, int sz) {   // from represents the rank of the sender
    int64_t* pool_data = static_cast<int64_t*>(data); // cast data for further use
    int size = sz/sizeof(int64_t);  // calculate  pool size

    // ... code to analyze pool
}
```

As noted in the comments in Alg. 5 messages are not sent immediately but much rather added to an internal queue. Only once `aml_barrier()` is called this queue and the code on the receiver's end is processed. Alternatively `aml_poll()` can be used for processing.

At the end of every iteration `MPI_Allreduce` is used to check if there are any elements left in any work queue including queues from other processes. If vertices are left the iteration is repeated, otherwise the algorithm has finished.

Algorithm 6: Checking queues globally

```
do {
    ... // BFS algorithm

    swap(q_work, q_buffer);

    // check how many elemtens are left in the working queue globally
    queue_size = (int64_t) q_work->size();
    MPI_Allreduce(
        MPI_IN_PLACE,
        &queue_size,
        1,
        MPI_INT64_T,
        MPI_SUM,
        MPI_COMM_WORLD
    );
} while(queue_size);
```

4.2 Further optimization and improvements

4.2.1 Package Splitting

One limitation of aml is a hard-coded limit for message sizes. The maximum number of vertices sent is depending on the number of processes (more processes result in smaller pools) and the graph size. For example: the transmission breaks down for 2 processes at a scale of 13 or bigger due to a buffer overflow.

In order to solve this, pools are split into chunks with a maximum of AML_MAX_CHUNK_SIZE.

Algorithm 7: Package splitting

```
1     while(vertices_remaining){
2         if(vertices_remaining <= AML_MAX_CHUNK_SIZE) {  // one package is enough
3             chunk_len = vertices_remaining;
4             vertices_remaining = 0;
5         } else {
6             chunk_len = AML_MAX_CHUNK_SIZE; // package splitting required
7             vertices_remaining -= chunk_len;
8         }
9
10        // send package
11        aml_send(&pool[chunk_start],
12            POOLDATA,
13            sizeof(int64_t)*chunk_len,
14            dest
15        );
16    }
```

Note that every call of `aml_send()` invokes `analyze_pool()` on the reciever's end. This function also requires the vertex u to be sent with each chunk, just like `send_visit()` in Alg. 2 requires u as a parameter. More about that in section 4.2.2.

Further improvement ideas: Currently the maximum package size is set to 4095 64-bit integers. In order to increase the buffer size any further changes to the aml code are necessary. *aml.c* provides definitions `AGGR` and `AGGR_intra` which apparently controll the buffer size. However we did not test this as this will most likely cause problems somewhere else.

4.2.2 Data compression

One problem with the pool approach is that the pool itself does not contain any information about the vertex u which needs to be sent as well. A simple way to resolve this would be to just append it to each pool. However this no longer works once packages are split up. This is due to the fact that aml can only send one continuous array of data up to a size of 4095. Once the pool is split into multiple packages u has to be inserted into the array by overwriting one value. We have tried to come up with a custom pool data structure that incorporates u in the 0th entry and in regular intervals after that. This however has lead to various other problems on the receiver's end and a worse performance compared to using regular vectors.

Looking at the actual data on can see that vertices are addressed using ascending integers. Since we have 2^{SCALE} vertices we need $SCALE$ bits to address all of them. As we are using 64 bit integers and the assignment only requires us to work with $SCALE < 27$ we can actually store 2 vertices in one integer.

Using this knowledge we can simply write u (which also requires a maximum of $SCALE$ bits) into the first 4 bytes of the first entry in the chunk.

Once recieved u can be extracted from the 0th entry and removed by using a bit-mask.

Algorithm 8: encoding additional information

```
// config.h
#define U_SHIFT 32  // number of bits used for u
#define U_MASK 0xffffffff00000000 // bitmask used for removing u

// sender code
pool[chunk_start] |= (u << U_SHIFT); // add u to the first entry of the chunk
aml_send(&pool[chunk_start], TAG_POOLDATA, sizeof(int64_t)*chunk_len, dest);
chunk_start += chunk_len;

// reciever code
int64_t u = (pool_data[0] >> U_SHIFT);  // Extract u from first entry
pool_data[0] &= ~ (U_MASK);  // clear out u with a mask
```

Further improvement ideas: Using the system shown in Alg. 8 one can compress data transmissions even further by also utilising the free bits of other pool entries. Thus drastically reducing the required bandwidth of messages. Smaller scales allow for more effective compression as less bits are required to address all vertices. This however comes at the cost of more complex send- and receive routines. While adding more processors increases processing power linearly, the same does not apply to communication bandwidth. Therefore at some point the benefits of compression outweighs the additional processing cost as the communication interface becomes the bottleneck for bigger scales.

4.2.3 Fast bit testing

Despite only having 2 valid states the actual size requirement for a boolean variable is implementation specific [Cpp11, Section 5.3.3, footnote 73]. In most cases `sizeof(bool) = 1` meaning that 8 bits are reserved. Depending on the target machine the the actual size may be even bigger if the smallest unit of addressable memory is more than 1 byte.

In order to save memory the reference implementation provides a convenient data-structure for the visited array that compresses boolean states down to single bits by using bit-shift operations and masking.

At the beginning of the BFS algorithm every bit in the array is set to false and over time more and more bits are changed to true. However checking a single bit requires multiple operations while checking an integer is faster.

We can use this idea to interpret a region around a bit as an integer and check if it evaluates to 0. If this is the case than we know that every bit in that region must also be 0. Knowledge of the actual position of the bit witin the region is not required. Only if the region != 0 we check for the single bit.

Algorithm 9: Pseudocode for faster bit checking. vis is an array containing n bits. The actual size is $1 + n/64$ 64-bit integers

```
def check_fast(pos)
    if vis[pos / 64] == 0   // check region around pos
        return false    // all bits are 0
    else
        return test_single_bit  // at least one bit is not zero. check with
            regular function
```

In a similar fashion a fast check can be used to check arrays that are almost full.

This method should work especially well at the start of the BFS algorithm when almost all bits are 0. With an increasing number of iterations the chance of encountering 64 bits that are all 0 diminishes. Therefore at some point the coarse check will lead to worse performance. We can

solve this by using a function pointer to the fast checking function and after some time change that pointer to reference the regular function as shown in Alg. 10.

Algorithm 10: Implementation of fast bit checking using swappable functions. Note how $\gg 6$ can be used for faster division

```
// bit checking function provided by graph500
#define TEST_VISITEDLOC(v) ((visited[(v) ulong_shift] & (1ULL << ((v) ulong_mask))
    ) != 0)

// bit test functions
inline bool test_visited_empty(int64_t v) { // for almost empty arrays
    return visited[(v) >>6] ? TEST_VISITEDLOC(v) : false;
}
inline bool test_visited_mixed(int64_t v) {
    return TEST_VISITEDLOC(v);
}
inline bool test_visited_full(int64_t v) { // for almost full arrays
    return visited[(v) >>6] == 0xffffffffffffffff ? true : TEST_VISITEDLOC(v);
}

// function pointer
// defined globally but each process has its own function pointer
bool (*test_visited_fast)(int64_t);

void BFS()
{
    test_visited_fast= &test_visited_empty; // assign function
    ...
    do {
        while(! q_work == 0) {
        ... // BFS Iteration
        }
        // calculate percent of filled bits and switch to regular function if
            above the threshold
        n_local_visits += pool[i].size();
        if((n_local_visits / (float) g_nlocalverts) > TEST_VISITED_EMPTY_CUTOFF)
            test_visited_fast = &test_visited_mixed; // switch to regular function
        aml_barrier()
        swap(q_work, q_buffer)
    }
    ...
}
```

Unfortunately this algorithm did not achieve the anticipated performance increase and in some cases even performed worse than the regular implementation. While we did not calculate the exact probabilities of encountering only 0-bits in an integer as a function of iterations, one can assume that the windows where this method is faster is only very short. Furthermore the regular implementation uses macros while we need to use additional function calls in order be able to swap out methods. Therefore this feature is not used in the final version of the code but it can be switched on by defining USE_TESTVISIT_FAST in *config.h*

Further improvement ideas: One way of improving this could be to implement the visited array with the smallest addressable space (usually 1 byte) instead of using 64 bit integers. This increases the chance of finding only 0s thus increasing the number of iterations where the faster function is used before we need to switch to the slower one. Also more testing can be performed to optimize the cutoff-level.

4.2.4 SIMD and multithreading

We did some experimentation with multi threading using OpenMP as there are some loops that can be parallelized very easily. In Alg. 4 the loops starting in line 9 and 12 are good candidates for multi-threaded code. However the hydra cluster only has one thread per core thus allowing no performance increase when using parallel for loops. One feature that the cluster does offer are AVX registers for SIMD instructions. Therefore we have added `#pragma omp simd` clauses to several loops.

While we did not make a full benchmark with and without this optimization on the cluster, testing on a personal computer did hint a slight performance increase, albeit still within the standard deviation. While it is possible that the compiler takes care of SIMD on its own we still left it in the code since it "doesn't hurt".

Further improvement ideas: While multi-threading loops will not increase performance, splitting calculation and communication into different threads and make use of the CPU's hyper-threading capabilities may lead to better results. Especially since aml occasionally uses blocking operations (MPI_Send). However this would also drastically increase software complexity.

4.2.5 Memory management and data structures

One of the main reasons we switched to C++ is the easier memory management for dynamically sized objects and containers.

For the work and buffer queues we chose pointers to `std::queue<int64_t>`. The usage of pointers allows for faster swapping by just swapping out the pointers.

The pools are implemented as `std::vector<int64_t>` because we need data to be stored consecutively for package splitting and transmission. The downside of using vectors is that by appending new vertices additional space must be reserved and in the worst case the complete data needs to be moved to a new location. We can resolve this by calling `reserve(g_nlocalverts)` after initialization in order to reserve enough space so the allocated memory never needs to grow. In total there are as many pools as processes which are also combined into one vector. The index of each pool corresponds to the rank of the assigned process.

The graph is stored in the compressed sparse row format while the visited array is stored in a bit-array. Both structures have been reused from the graph500. Their data is also distributed equally by the reference code amongst all processes to reduce the required memory.

Algorithm 11: Data structure usage

```
1  // queues
2  queue<int64_t>* q_work = new queue<int64_t>();
3  queue<int64_t>* q_buffer = new queue<int64_t>();
4  swap(q_work, q_buffer); // swap pointers to the queues
5
6  // pools
7  vector<vector<int64_t>> pool(comm_size);
8  for(int i = 0; i < comm_size; i++)
9          // reserve enough space to prevent vector from growing
10         pool[i].reserve(g_nlocalverts);
11
12 // adjacency matrix (rewference implemenmtation
13 // retrieve 3rd vertex in 0th row
14 j = rowstarts[0] + 3
15 vertex = COLUMN(j)
```

Further improvement ideas: There is a little bit of overhead to those structures compared to the

ones that the reference implementation used. For example dequeueing from `std::queue<int64_t>` also de-references the object from the queue. Since we are not limited by the amount of available memory this is not really necessary which is why the graph500 implementation uses a simple array combined with a head and tail counter.

4.3 Benchmark

All benchmarks shown in the following section were performed using our final version of the code. This includes the all optimizations mentioned in Section 4.2 except fast bit testing which produced a negative effect in many cases.

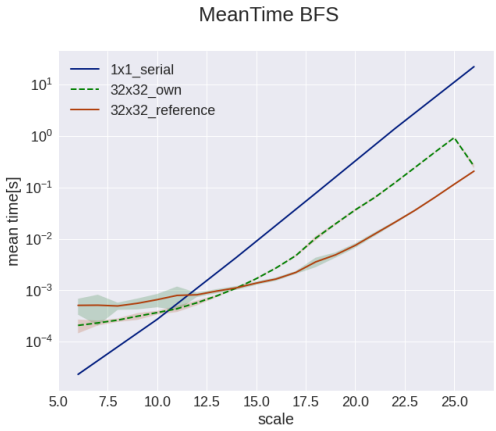


Figure 2: runtimes compression between custom and reference BFS

For the 32x32 configuration our implementation (green) is faster than the reference implementation in the lower part of the scale-factor until the scale-factor is 14. To have another comparison we also included the benchmark for the serial BFS implementation. We did not test the full range from the scale-factor up to 26 on the cluster for the serial implementation due to the expected very long runtime. To avoid that we only tested it until a scale-factor of 22 and then we linearly interpolated the rest. The same method we used for the other cluster configurations (1x32, 32x1). The parallel BFS implementation pays off in case of efficiency at a scale-factor of 11. After a scale-factor of 16 the serial and parallel BFS runtime is roughly parallel and the serial implementation is roughly 4 times slower than the parallel implementation. In the end for the last scale-factor the reference implementation is roughly 7 times faster.

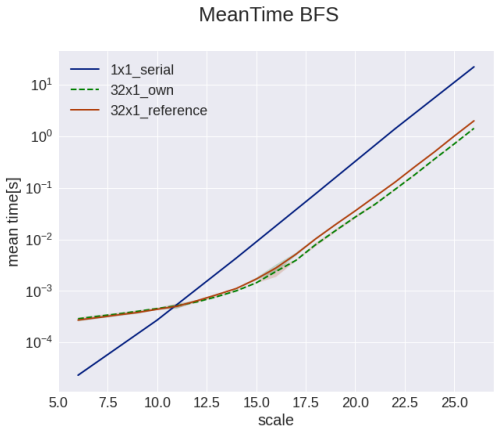


Figure 3: runtimes compression between custom and reference BFS

For the 32x1 configuration our implementation is a bit faster. This tendency is the same for the whole scale-factor range. In the lower part of the scale-factor the reference implementation is a bit faster but with increasing factor our implementation is faster.

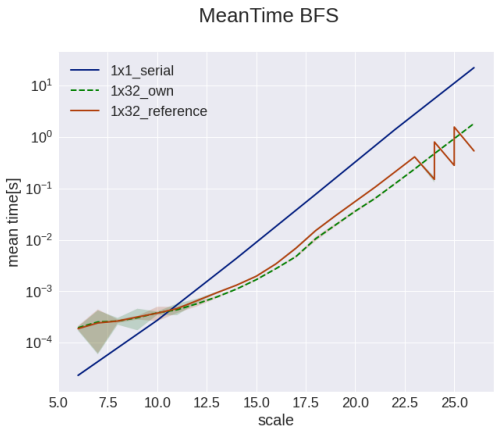


Figure 4: runtimes compression between custom and reference BFS

Again our implementation is a bit faster than the reference code.

Occasionally we encountered some weird spikes with the runtime which we are unable to explain. Since benchmarking the algorithm at high scales takes very long we have used the data from the run with the least amount of spikes.

5 Conclusion

We are happy to see that the effort we have put in this exercise paid off in the end. While we did not beat the reference implementation in all scenarios we managed to improve performance for small to medium sized systems.

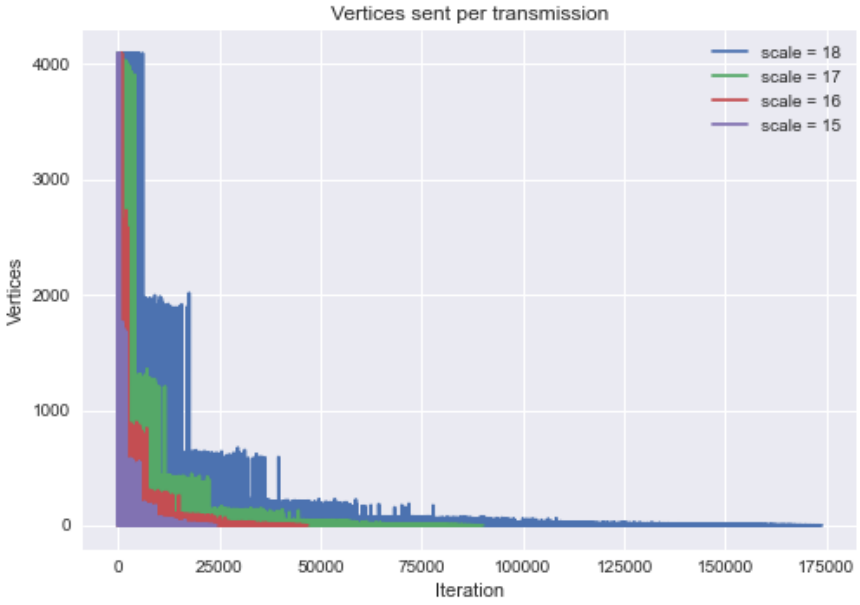


Figure 5: Number of vertices sent with each transmission for different scales on a 1x1 System

Figure 5 shows the number of vertices that are sent with each transmission i.e. the package size. One can clearly see that at the beginning of the algorithm the most data bandwidth is required. This is also where the pooling approach has its biggest impact. With increasing number of iterations packets get smaller and the effect diminishes. Furthermore with an increasing scale the relative amount of time the algorithm spends on sending small packets increases. Meaning that for small scales we need to send a lot of data in the beginning but soon after the algorithm has already completed. With bigger scales we still require a lot of bandwidth in the beginning and more package splitting (seen by the graph capping out at 4095) but very quickly we are stuck with sending small amounts of data for a long time compared to the total runtime of the algorithm.

While Figure 5 was created using only a 1x1 system [1] the argument still holds for bigger systems. The only difference is that the average number of vertices sent is about half for systems with twice as many processes but the total transmission requirement stays the same (ignoring data that can be sent locally)

This effect is most pronounced on the 32x32 configuration which is reflected in the benchmark. The packets quickly reach a very small sizes and the overhead we create with a more complex transmission routine, encoding u into the data, using stl-containers, etc. affects performance more than we gain by pooling the data.

On the other hand performance gains on smaller scales with less processes are huge. Testing on a personal PC with 4 processes and at scales of 16 has shown performance increases of up to 2.7 times!

Furthermore there is a small difference between Figure 3 and 4. The performance gain compared to the reference implementation on the 1x32 system is a little higher compared to the 32x1 system.

[1] combining filestreams from multiple processes to collect data is rather complicated

This is most likely due the fact that communication between nodes takes longer than communication within a node. Therefore the performance gain created by pooling diminishes if the time each transmission requires is increased by a fixed amount due to inter-node communication.

Further improvement ideas: Since pooling apparently no longer pays off after a certain amount of iterations one could switch to sending single vertices after a predefined amount of time.

References

[Cpp11] Cpp. *Standard for Programming Language C++*. 2011.

[Lóp17] Mireya Paredes López. *Exploring vectorisation for parallel breadth-first search on an advanced vector processor*. 2017.

[gra] graph500. *BENCHMARK SPECIFICATION*. URL: `https://graph500.org/?page_id=12`.