SOLVING SINGLE SOURCE SHORTEST PATH ON SHARED MEMORY IMPLEMENTATION OF THE Δ-STEPPING ALGORITHM

Erik Träff, Clara Brimnes Gardner, Andrey Oblupin, Gabriela Evrova

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

This report describes a shared memory parallel implementation of the Δ-stepping algorithm, which solves the Single Source Shortest Path problem. Various optimizations of the implementation are proposed and tested. We benchmark our implementation against the one of the *GAP Benchmark Suite* on five different graph structures which are part of the suite. The resulting runtimes and speed-up factors are highly graph dependent. We do not outperform the implementation of the *Gap Benchmark Suite*.

Keywords. Single Source Shortest Path (SSSP), Parallel Random Access Machine (PRAM), Δ-Stepping, Intel Xeon.

1. INTRODUCTION

Motivation. The Single Source Shortest Path (SSSP) problem arises in a number of real world scenarios - for instance computing shortest path in road graphs or finding shortest connections across social networks. The large size of the graphs representing the data creates the need for fast, parallel implementations.

Significance. In this report, we study the Δ-Stepping algorithm [1], which is fundamental to solving the SSSP-problem in PRAM. We implement the algorithm, analyse its performance and optimize our implementation in the parallel setting. We test the performance of our code on different graph structures, described in section 5. Our code is written in *C++11* and uses *OpenMP* [2] for parallelism. The resulting implementation has been made open source [3].

Related Work. For solving the SSSP-problem, there is an implementation of the Dijkstra's algorithm in the Boost Graph Library (BGL) [4]. However, in this library there is no implementation for solving SSSP in the shared memory setting.

An implementation of the Δ-stepping algorithm on shared memory is contained in the *GAP Benchmark Suite* [5]. The code is written in *C++* and uses *OpenMP* [2]. The implementation is built on top of their own vector structure, which differently than *std::vector*, allows for parallel initialization and avoids initialization after resizing the vector container. For representing the graph structure, *GAP* uses the Compressed Sparse Row (CSR) format. We describe their approach in greater detail in section 4.

2. BACKGROUND: SINGLE SOURCE SHORTEST PATH

Problem Definition. The SSSP-problem is defined as follows: given a graph with non-negative edge weights and a source-node contained in the graph, we want to calculate the shortest distances from the source node to every other node in the graph. If a node is not connected to the given source node, the corresponding distance is considered to be infinity or another predetermined value.

Algorithms. There exist a number of algorithms to solve the SSSP-problem. The well known are Dijkstra's algorithm and the Bellman-Ford algorithm. Generally speaking one can say that for the SSSP-problem there is a trade-off between work-efficiency and parallelizability. Dijkstra's algorithm is work-optimal and its worst case asymptotic performance is $O(|E| + |V|log|V|)$. However it is not parallelizable, due to the requirement of a fixed order of node traversal of the graph. In contrast, the Bellman-Ford algorithm is highly parallelizable and also works with negative edge weights, but it has worst case asymptotic performance of $\theta(|E||V|)$.

3. THE Δ-STEPPING ALGORITHM

In the Δ-stepping algorithm [1], the distance to each node can be updated multiple times until it is settled, thus we perform more work to gain on parallelizability.

We now explain the terminology used in the algorithm.

Buckets. In each bucket B_i, we store nodes of the graph which have a tentative distance to the source node contained in the $[i*\Delta, (i+1)*\Delta]$ interval, which means that we radix sort the nodes by the distance to the source node.
Request. A pair of a node and a new suggested tentative distance. Processing a request is equivalent to checking if we can *relax* the distance to the specified node in the request.
Types of edges. *Light* edges are edges which have weight $\leq \Delta$, and *heavy* edges have weight $> \Delta$.
Phase. Procedure of processing the smallest indexed non-empty bucket.

Pseudocode.

Input: $G(V, E)$, source point s
$relax(s, 0)$
foreach $v \in V$ **do** $tent(v) := \infty$
while $\neg isEmpty(B)$ **do**
  $i := min\{j \geq 0 : B[j] \neq \emptyset\}$ //smallest index of non-empty bucket
  $Req := findRequests(B[i], light)$
  $R := R \cup B[i]$ //*R*: set of visited nodes
  $B[i] := \emptyset$
  $relaxRequests(Req)$
end while
$Req := findRequests(B[i], heavy)$
$relaxRequests(Req)$

function $findRequests(B, edgeType)$:
  return $\{(w, tent(v) + c(v, w)) :$
  $v \in B \wedge$(v, w) is of *edgeType*$\}$

Detailed description of *relaxRequests* can be found in [1].

4. PROPOSED PARALLEL IMPLEMENTATION

In this section we describe our parallel implementation, explain how we arrive to different performance optimizations and highlight the difference between our and the *GAP* implementation.

Shared memory. In the parallel setting, the algorithm proceeds in a similar way as in the sequential setting, with the following modifications:
- Each node of the graph is assigned to an "owner" thread.
- We have thread local buckets and the globally smallest non-empty bucket is found by a reduction. Our implementation reuses bins cyclically.
- In each phase, we globally find the first non-empty bucket and each thread generates requests from its private bucket. The request is written to a request buffer of the thread that owns the node contained in the request. An illustration of this is shown on figure 1
- The above point implies that each request is processed by the thread which owns the node in it, which by definition ensures atomicity of the update step.

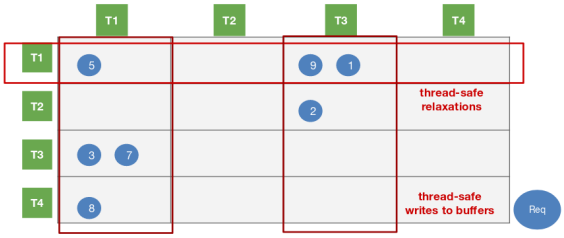

Fig. 1: Generation and processing of requests. We implement strided arrays to avoid false sharing between threads.

The Δ-stepping algorithm is memory bound, only few additions and comparisons are performed when requests are generated and relaxed. As these operations require random access to arrays which are far greater than the cache size, we expect this to lead to a stalled execution, as the threads are waiting for data to be transferred from RAM.

It is important to note that the above described approach leads to static load-balancing, which is determined by the node ownership.

Optimizing performance. In order to improve the performance of the implementation, we test several ideas - software prefetching and work sharing between threads when processing buckets.

Software Prefetching. Significant part of memory accesses are performed randomly, as node numbers are used as indexes to the adjacency array. Therefore, we expect cache-misses to be one bottleneck.

We studied the number of cache-misses using the Intel VTune profiler, which uses perf as backend [6] to obtain the performance counters. This test showed that there were quite a lot of last level cache misses during the execution of the program, and a strategy for reducing the cache-misses associated with random memory access was therefore desired.

We added *Software Prefetching* using the built-in function `__builtin_prefetch` [7] from gcc, in order to reduce the number of cache-misses. While the current node is being processed, we added prefetching to load the entry in the adjacency lists corresponding to the next node in the bucket. Results of this optimization are presented in section 5.

Shared Bucket Traversal. In our implementation each thread is generating requests from its private, but globally smallest indexed non-empty bucket. Now the idea is to make the bucket structures of each thread shared with all other threads, as in the phase of generating requests we have read-only accesses. This allows the workload of generating requests to be distributed dynamically between threads. We achieve

this with *guided OpenMP scheduling* of the bucket structure chunks across the threads. However we introduce one more change in this case, which is not keeping track of visited nodes. More specifically, we do not remove the nodes with light edges from the bucket once they are processed. We believe that as more threads are sharing the work, constantly adding and removing to data structures can create an additional overhead.

In contrast to our implementation, the *GAP* implementation consists of non-atomic distance updates - the threads first find nodes whose distances can be improved and add them in their corresponding thread local buckets. Afterwards, each thread copies all nodes from its local globally smallest non-empty bucket into *only one* shared bucket structure. They make use of the fetch-and-add and compare-and-swap atomic operations, in contrast to our implementation where the distance updates are atomic by definition. We use several critical sections - to compute the smallest non-empty bucket index and to detect when we are finished with one phase of the algorithm. We expect their implementation to outperform ours, as it incorporates dynamic load balancing of the updates. *GAP* also never remove nodes from the shared bucket and resize it instead. This is conceptually similar to our idea of not removing visited nodes from the bucket.

5. RESULTS

In this section, we show a sequential benchmark of our implementation, we outline the setup for all the experiments that we perform, we describe the different graph structures we use, analyse the performance of our proposed optimizations and we compare the runtimes and speed-ups of our implementation compared to the existing one of *GAP* for all graphs contained in the *GAP benchmark suite*.

Experimental Setup. Our code was compiled using `gcc 4.9.2` using also the flags `-Ofast -march=native`. We ran the experiments on the ETHZ Euler cluster. Euler consists of different types of nodes. We used full nodes consisting of two sockets, each with 12-core Intel Xeon processors, either:
- *Intel Xeon E5-2680v3*: 2.5-3.3 GHz and between 64 and 512 GB of DDR4 memory clocked at 2133 MHz;
- or *Intel Xeon Gold 5118*: 2.3 GHz nominal, 3.2 GHz peak, 96 GB of DDR4 memory clocked at 2400 MHz).

In experiments where different running times were compared to each other the same source nodes were used. This was ensured by using the same random number generator and a constant seed as implemented in *GAP*. This is of importance since the computational DAG, and thus the execution time, depend on the source node.

Benchmarking Data. The *GAP* benchmark specification [5] consists of five graphs:

US road: (23.9M nodes, 58.3M edges) A real world graph of the road network of the entire USA.

Erdős–Réyni: (33.6M nodes, 536.8M edges) A synthetic graph where all nodes draw their neighbours from a uniform distribution.

Kronecker: (33.6M nodes, 1.04B edges) A synthesised graph using the Kronecker graph synthetization, with parameters corresponding to the Graph500 framework [8]. This graph is not fully connected.

Web: (50.6M nodes, 3.62B edges) A real world graph of links between web-pages registered in the Slovakian internet domain `.sk`.

Twitter: (51.5M nodes, 2.4B edges) A real world graph of relations on the social media platform Twitter.

The two synthesised graphs, Erdős–Réyni and Kronecker, were chosen to be smaller than the size originally defined by the *GAP* benchmarks. This choice was made such that they fit in the main memory of a regular compute node on the Euler cluster (64GB).

Sequential Benchmark. Figure 2 shows a sequential benchmark of the code we implemented against the implementation of Dijkstra's algorithm in the *Boost Graph Library* [4] and the *GAP* implementation [5].

Even though Dijkstra's algorithm is a work optimal SSSP algorithm, our Δ-stepping implementation has better performance than the *BGL* Dijkstra's implementation. We do not manage to obtain better performance than *GAP*. As a consequence, we also expect to be slower than the *GAP* implementation in all subsequent experiments.

Benchmarks of Optimizations. Figure 3 shows the effects of introducing prefetching as described in section 4. We see that prefetching systematically improves the runtime for each number of threads. This modification is kept in all future experiments.

Figure 4 shows the effects of implementing the shared bucket traversal without removing visited nodes from the buckets. We see that when using one thread, the impact of not removing nodes from the structures significantly improves the performance. However, for a higher number of threads, we cannot deduce that this implementation results in an improved performance. When we have a Non Uniform Memory Access architecture (NUMA), in this case more than 12 threads, due to the cache coherency we have a high variation of the running time and a significant drop in performance.

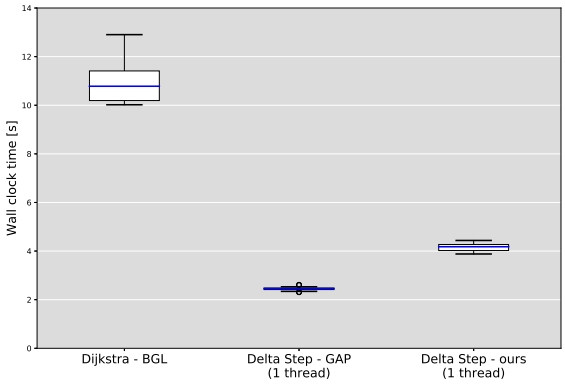

Fig. 2: Wall-clock time for road graph, 50 runs per implementation

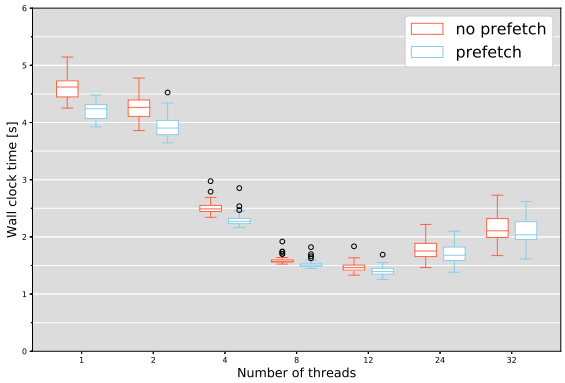

Fig. 3: Wall-clock time for road graph, with and without prefetching, 50 runs per thread number

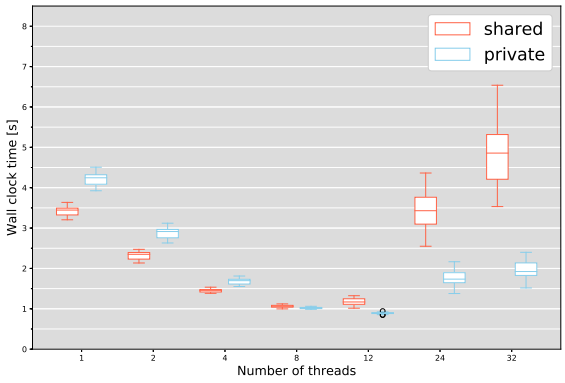

Fig. 4: Wall-clock time for road graph for our two implementations, 50 runs per thread number

Scaling Benchmarks. In this section, we present the runtimes and speedups of our initial implementation (with added prefetching) for the graphs described earlier, and we compare the performance of our implementation to the one of *GAP*.

To check the correctness of our parallelization, we compared the output of our runs to the output of the *GAP* implementation executed with a single thread.

All the experiments were run 64 times. We report the medians of the running times, along with their non-parametric confidence intervals as by looking at the histograms of running times, we saw that the data is not normally distributed.

For the speedup plots, we show the speedup relative to both base-cases, the GAP implementation compiled with *-fopenmp* and using one thread and ours.

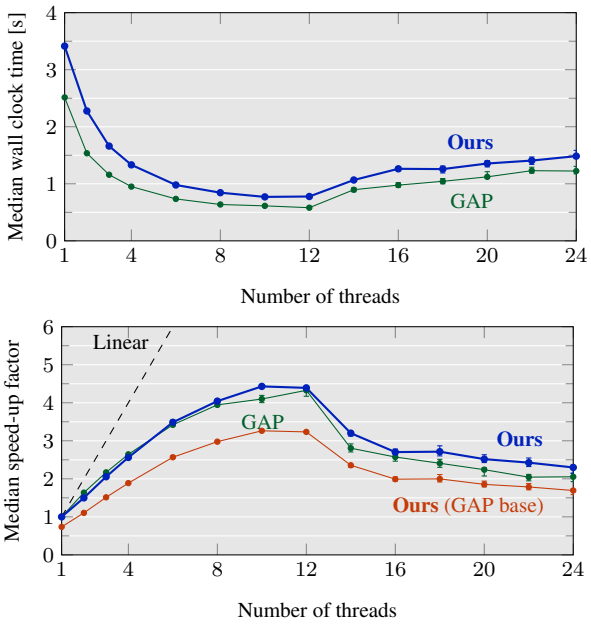

Fig. 5: Median wall-clock time and speed-up factor for road graph.

Figure fig. 5 shows the scaling benchmark of the US Road graph. Our implementation performs similarly to the GAP-suite although slightly slower. Both implementations have an increase in runtime when more than 12 threads are used, this is caused by NUMA patterns, as it coincides with the two sockets of 12 cores in the used compute nodes.

The benchmark performance of the Erdős–Réyni graph is shown in fig. 6. Again the *GAP-suite* performs better than our implementation and the speedup behaviour of the two implementations are similar. When having one thread, the *GAP-suite* performs significantly better than our implementation, which greatly affects the speed-up. However, asymptotically we approach the performance of *GAP*.

Figure fig. 7 shows the scaling of the Kronecker graph, which is not fully interconnected. It is seen that the runtimes of our implementation are greatly varying. The runtimes are

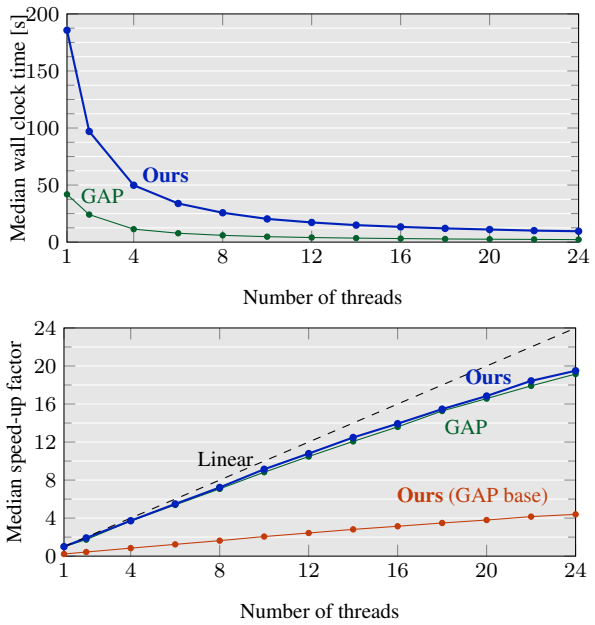

Fig. 6: Median wall-clock time and speed-up factor for Erdős–Réyni graph.

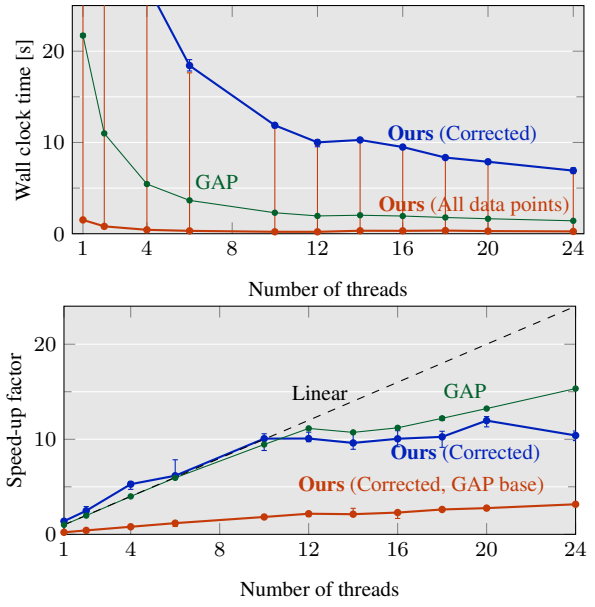

Fig. 7: Median wall-clock time and speed-up factor for Kronecker graph.

clustered in two groups; significantly faster than the *GAP* reference implementation, or slower than the *GAP* reference which is the expected result. This split is due to the algorithm completing much faster if the source point is located in a small disconnected subgraph. When only looking at the slow results, the runtimes are very similar to those of the US Road and Erdős–Réyni graphs (figs. 5 and 6). Figure 7 shows that our implementation has super-linear speed-up when using few cores. This is attributed to bad performance tuning of the sequential base-case.

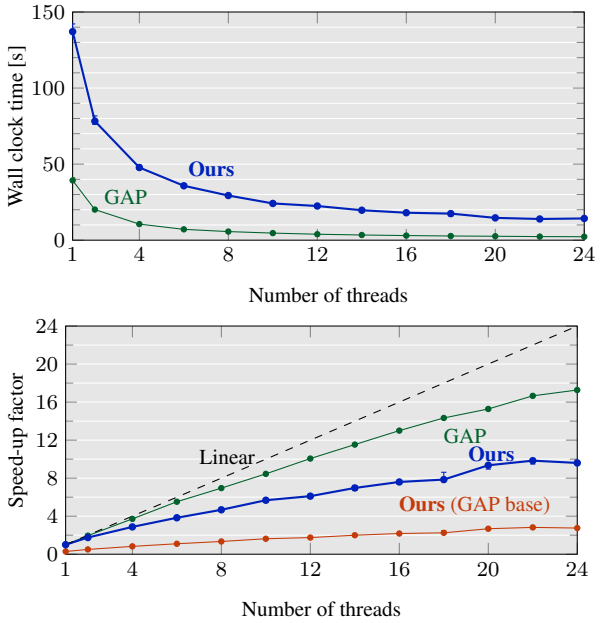

Fig. 8: Median wall-clock time and speed-up factor for the web graph.

The results from the benchmarks on the Web graph are shown in fig. 8. For this graph there is a notable difference between the scaling of our implementation and that of the *GAP*, even when using our own sequential version as a base case. This is due to the fact that this graph consists of nodes with a greatly varying number of neighbors. This is probably due to the dynamic load balancing of *GAP*.

The results for the Twitter graph are presented in fig. 9. The Twitter graph shows similar results as most other graphs.

6. THOUGHTS ON DISTRIBUTED MEMORY

The initial goal of this project was to implement a distributed memory version of the Δ-stepping algorithm. The main difficulties when considering a distributed memory setting of the Δ-Stepping algorithm are the graph partitioning and the communication overhead. As the algorithm requires relaxation requests to be handled by the process that ”owns” the node, we would need to send large amounts of requests between processes.

In the two-sided Message Passing setting, it is difficult to interleave the generation and sending of requests, as it would result in sending more messages to the same process, which

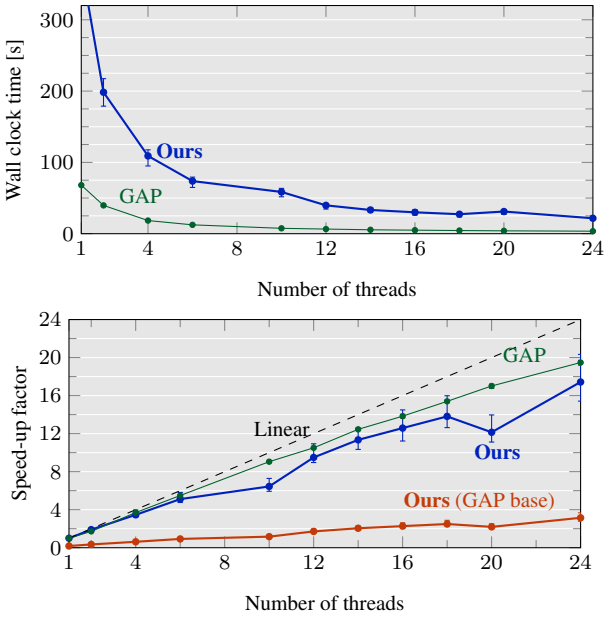

Fig. 9: Median wall-clock time and speed-up for twitter.

could reduce performance due to the communication latency. On the other hand it could be possible to interleave receiving messages with handling them. If good performance was to be obtained in a distributed memory setting, either one-sided MPI or an active messaging protocol should be used in order to obtain reasonable performance. An example of such an implementation can be found in [9]. Given these difficulties, we decided to focus on the shared memory setting.

7. CONCLUSION

We presented a shared-memory parallel implementation of the Δ-stepping algorithm. We reason about and test different optimizations - prefetching and shared bucket traversal. We tested our implementation on all graphs contained in the *Gap Benchmark Suite*. It can be concluded that both runtimes and speed-up factors are highly graph dependent. We do observe speedup when we use the *GAP* implementation with one thread as a base-case, however when we use our implementation as base case, we observe that we gain significant speedup and that we were able to successfully parallelize our code. Even though we do not outperform *GAP*, we observe comparable performance in the case of graphs with very high locality, such as the road graph. Also, the performance of our implementation asymptotically gets closer to the one of *GAP*, even though conceptually we approach the parallelization of the Δ-Stepping algorithm in a different way.

8. DISTRIBUTION OF WORK

The amount work performed has varied significantly between group members. Thus for evaluation purposes, a small notice of how the work was distributed is included; Erik and Gabriela performed the implementation, performance optimization, and benchmarking. Clara contributed to discussions on performance improvements and performed the statistical analysis of benchmark data. Andrey participated in most group meetings, and contributed by applying the roofline model to the algorithm, although it was found excessive for this report.

9. REFERENCES

[1] U. Meyer and P. Sanders, "δ-stepping: a parallelizable shortest path algorithm," *Journal of Algorithms*, vol. 49, no. 1, pp. 114 – 152, 2003. 1998 European Symposium on Algorithms.

[2] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *IEEE Comput. Sci. Eng.*, vol. 5, pp. 46–55, Jan. 1998.

[3] `https://github.com/evrova/dphcp_deltastep`.

[4] *The Boost Graph Library: User Guide and Reference Manual*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[5] S. Beamer, K. Asanovic, and D. A. Patterson, "The GAP benchmark suite," *CoRR*, vol. abs/1508.03619, 2015.

[6] "Perf wiki." `https://perf.wiki.kernel.org/index.php/Main_Page`, 2019. Accessed: 03-01-2019.

[7] I. Free Software Foundation, "Using the gnu compiler collection (gcc): 6.58 other built-in functions provided by gcc." `https://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html`, 2018. Accessed: 03-01-2019.

[8] G. S. Committee, "Graph500 website." `https://graph500.org/`, 2019. Accessed: 03-01-2019.

[9] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "Am++: A generalized active message framework," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, (New York, NY, USA), pp. 401–410, ACM, 2010.