PARALLEL GRAPH COLORING

Alain Denzler, Philippe Mösch, Spyridon Margomenos

Department of Computer Science
ETH Zürich
Zürich, Switzerland

ABSTRACT

Graph coloring is an important problem and difficult to solve efficiently in parallel. We reimplemented Boman's graph coloring [1] for a distributed memory setup, with focus on compact data structures and predictable memory accesses. We are proposing a new color assignment algorithm, evaluated on synthetic and real-world graphs. Our experiments show good scalability, independent of the graph partitioning, and show a substantial performance gain in comparison to the Boost parallel graph library [2].

1. INTRODUCTION

Motivation. Graph coloring is an interesting problem that arises in many different applications. For example, in task scheduling a set of tasks with dependencies has to be scheduled to a finite number of workers in an efficient manner. Using a graph whose vertices correspond to tasks and edges denote dependencies, graph coloring can find a suitable conflict-free allocation to workers.

The problem of finding the minimum number of colors for a given graph is NP-complete, therefore there is no algorithm to find a solution in polynomial time. Instead, we resort to approximating the solution and parallelizing the algorithm to achieve better performance.

We chose to implement Boman's graph coloring [1], a parallel graph coloring algorithm for distributed memory setups. Due to the flexible definition of the algorithm, several design choices are left to the implementation. We implemented several versions of these choices and evaluate them using synthetic and real-world graphs. Specifically, we investigated the trade-off between using synchronous and asynchronous communication and implemented different variants of the color assignment algorithm. Our implementation focuses on predictable array accesses and compact data structures to fully leverage the power of the hardware.

Related work. We took our inspiration for this project from the original paper [1], investigating if their results stand the test of time. The authors assumed well-partitioned graphs and found a superstep size of 100 and first-fit color assignment to be the best choice. They performed their experiments on a cluster with dual 900 MHz Intel Itanium 2 CPUs and 4GB of memory. They themselves took inspiration in papers such as [3] where processors are assigned vertices, coloring is done with first fit and the simultaneous coloring is causing conflicts that need to be resolved.

2. BACKGROUND

In this section we introduce graph coloring in general and our algorithm of choice, Boman's graph coloring algorithm.

Graph coloring. Given a graph G = (V,E), the algorithm assigns a color to each vertex so that no two neighboring vertices share the same color. The goal is to minimize the number of colors needed to color a graph. In typical use-cases, colors are denoted by positive integers.

Graph coloring is NP-complete meaning an efficient algorithm that finds an optimal solution for a general graph does not exist. There exist however several approximative and randomized algorithms with satisfying runtime bounds. One simple example is the first-fit algorithm: While traversing the vertices of a graph in any order, always assign the smallest legal color to the vertex, then proceed to color the next vertex. While the quality of the solution depends on the order of the vertex traversal, the algorithm only needs to color each vertex once is guaranteed to finish without conflicts.

Boman graph coloring. Boman's graph coloring algorithm [1] is an iterative distributed memory algorithm that works by partitioning the vertices among the processes. The algorithm proceeds in rounds, each round consisting of two phases: the vertex coloring phase and the conflict resolution phase. During the vertex coloring phase, each process colors the vertices it owns independently. After all the vertices are colored, conflicts are detected and vertices are marked to be recolored during the next round. The algorithm terminates when there are no conflicts left.

Communication is performed in supersteps during the vertex coloring phase. After a batch of vertices has been colored, their color information is sent to the other processes. Packing the color updates for multiple vertices into the same message allows for more efficient communication due to fewer messages needed and less time spent assembling, waiting for, and disassembling messages. Superstep size is an important tunable parameter of the algorithm, and depends on the hardware and the graph involved.

Another important choice left to the implementation is the choice of color assignment algorithm. This algorithm determines in what order the colors are assigned to a vertex to be colored. We introduce three different choices in Section 3.

During the conflict detection phase, each process checks its color assignments for conflicts with vertices owned by other processes. Vertices with conflicting colors are marked to be recolored in the next rounds of the algorithm. This phase does not need any communication because at the end of the vertex coloring phase all the color information is available to all the processes.

3. IMPLEMENTATION DETAILS

This section introduces the technical details of our implementation of Boman graph coloring. Due to the flexible and general formulation of the algorithm, there are several strategies for implementing the algorithm, and their performance impact depends on both the system and their respective implementation. Specifically, this includes the choice of color assignment algorithm, the size of the superstep and the communication strategy. We implemented several versions of these choices to evaluate and reason about the performance impact of each choice.

We implemented the algorithm for a distributed memory setup using MPI and C/C++. We assume the graph is already distributed and every process has complete knowledge about the topology, in particular each process knows how many vertices and edges there are in total, as well as which process owns a particular vertex. Additionally, each process knows the set of boundary vertices, meaning the vertices which share an edge with another processor. We do not assume any special partitioning of the graph in contrast to the implementation in the original paper [1].

Each vertex in the graph has both a global and a local id, which are implemented using unsigned 32-bit integers. We assume that all the vertices of process i have smaller global ids than the vertices of process $i + 1$, and greater global ids than the vertices of process $i - 1$. If that is not the case, we compute a new vertex numbering that satisfies these conditions. This ordering is important because it allows computing the global id using only knowledge of the process that owns it and the local id. Additionally, the memory layout and addressing is simplified because the data can be laid out in memory in a contiguous manner.

Local for each process, the graph is stored using two C-style arrays and one integer variable. The integer variable holds the global offset, which is used to compute the global id of the vertices that are owned by this process. The two arrays are used to store the graph in compressed sparse row (CSR) format. Each process owns a continuous slice of rows in the adjacency matrix, and thus uses one of those arrays to store the offset at which the next row starts, and the other to store the columns that are nonzero. Because the adjacency matrix is binary, we do not need to store the values for the nonzero matrix entries, which are implicitly one. All the primitive data types are unsigned 32-bit integers to maximize the size of the graph we can store, while still keeping the memory footprint under control.

Colors are stored locally on each process in an array of unsigned 16-bit integers. Each process stores the color information for all the vertices. While this introduces a lot of redundancy and needs memory, it allows the processes to compute the next superstep independently and also facilitates the exchange of color informations during the communication phases. We minimize the memory overhead by using unsigned 16-bit integers, whose numerical range was sufficient for all our test graphs. The color array is indexed using the global id of the vertices.

After each superstep, each process has to communicate the colors it changed to all the other processors. Because we store all the color information on each process, we use the MPI Broadcast function to distribute the updated colors to all the other processes. Even though only the processes that have an edge to the sending process actually need the color information, and even in this case do not need all the updated colors of this superstep, we decided to use broadcasts for scalability reasons: With an increase in the number of processes, the number of inter-process edges grows, until eventually most of the processes share edges. Additionally, broadcasting contiguous chunks of memory is very efficient because the message does not need to contain any metadata and position information, especially in the first round of the algorithm. Knowing only the sender, the receiver can independently compute the position information and copy the message into the right place in the color array. There are no race conditions because each process only writes to the part of the array it owns and only performs read operations on the rest of the color array.

The communication using broadcasts is especially efficient during the first round of the algorithm because in this round every vertex is colored and all of the elements of the color array are set. Thus the messages only need to include the actual colors and the starting index to copy the colors into can be computed locally on each process knowing the sender of the colors. Starting from the second round however, the algorithm does not stream through memory as nicely as in the first round and only select colors are updated. To minimize the message size and the amount of communication needed, we treat the first round separately from the successive rounds. Starting from round 2, two messages are broadcast from each process: one including the indices to update, and another including the new color values. While this doubles the amount of communication, we only send the values that change. It also introduces more complexity locally at each process because instead of using a memcpy call, each process needs to parse the messages and update the colors. Nevertheless, because the number of conflicts after the first round is a small fraction of the number of vertices, this communication scheme performs better.

After performing one round of vertex coloring, each process locally computes the set of conflicting vertices for the vertices it owns. This requires no communication, because at the end of each round each process owns the complete color information and thus only performs a lookup on the local color array. For each conflict, there are two vertices involved and only the vertex with the lower global id is recolored. This vertex is added to a local std::vector that holds the vertices to be recolored in the next round. To guarantee termination and correct communication, the maximum number of conflicts has to be sent to all the processes because all the processes have to know how many supersteps to perform.

The main guiding principles for our implementation were to make things as simple and predictable as possible. Especially during the first round where all the vertices are colored, the implementation can greatly benefit from caching and prefetching effects because most of the memory accesses are streaming through memory in a continuous fashion. For later rounds, the adjacent vertices are stored closely together in memory where the implementation again benefits from spatial locality. All the data is stored using the smallest data type permissible to reduce the memory footprint. Compared to a primitive implementation using linked lists or pointers, our implementation is tuned to leverage the power of the hardware, especially allowing the application to use caches and memory level parallelism efficiently.

Communication style. One major design choice when implementing a message-passing based algorithm is whether to use blocking or non-blocking communication. In the blocking case, all the processes wait for all the communication calls to complete before they restart another computation phase. This introduces a synchronization barrier and usually involves some processes waiting idle until all the processes arrived at the barrier. In the non-blocking case, the communication call immediately returns and the process is allowed to continue the computation. This allows overlapping the communication latencies with the computation.

In our specific case, we implemented both styles and evaluate them in Section 4. The main trade-off involved is how coloring vertices based on outdated color information affects the number of conflicts at the end of each round. In the blocking communication case, a vertex can only conflict with vertices colored at the same superstep, while using non-blocking communication it can also conflict with vertices colored in the next superstep. This is the case because during the next superstep the updated colors are still in-flight and not ready to be looked up yet. The smaller time spent waiting for communication to complete may however compensate for having to recolor more vertices due to conflicts.

Color assignment algorithms. Another important choice is how to assign colors to vertices. In this section, we introduce the 3 variants of color assignment we implemented, the first two of which were proposed in the initial paper [1] and the third being a new proposition extending the second variant.

The first variant implemented is the simple first-fit (FF) algorithm: For each vertex, the colors of the neighboring vertices are looked up and the smallest legal color is assigned to the vertex. While FF is simple and efficient to implement, it has however one disadvantage in parallel applications: the color distribution is highly skewed. Small colors are assigned much more often, and thus the likelihood for conflicts is involving those small colors is much higher.

To reduce the likelihood of conflicts in parallel applications, staggered first-fit was introduced. Instead of all processes starting their color assignments at the smallest color, each process gets a different starting color and assigns colors starting from that value. Using an initial guess of the number of colors needed, the process assigns colors starting from 0 if there is no legal color in the interval between his starting color and the initial guess. This balances the color distribution because each process starts from its own base color and thus reduces conflicts.

As a third variant, we propose "conflict-avoiding staggered first-fit". The basic mechanism is identical to Staggered first-fit. Once the process overflows its assigned color range however, the conflict-avoiding variant does not proceed to assign the starting color of the next process. Instead, it starts assigning the last color of the range of the previous process, and then proceeds towards the smallest color. This allows for an even better distributed color assignment, because it avoids the most common colors of other processes and starts assigning rare colors.

Superstep size. The last major design choice is the choice of superstep size. Smaller supersteps introduce more communication and synchronization points, but reduce conflicts because the local color information is better. Larger supersteps on the other hand allow for less idle times waiting for communication to complete. We chose to expose superstep size to the user as a runtime parameter for fine-tuning the algorithm to the specific graph to be colored.

Option	Randomly Generated Graphs	Real World Graphs
CPU	2x Intel Xeon E5-2697 v3	3x Intel Xeon E5-2680 v3
#Cores	28 total cores (2x 14)	28 total cores (2x 12 + 4)
RAM	256GB DDR4 ECC 2133 MHz	150GB DDR4 2133 MHz
Operating System	Ubuntu Server 16.04.3 LTS	Centos 7.4.1708
Kernel	4.4.0-87	3.10.0
Compiler	gcc-5.4.0	gcc-4.9.2
MPI	OpenMPI 1.10.2	OpenMPI 1.6.5
Boost Library	1.58.0	1.59.0

Table 1: Experimental setup

4. EXPERIMENTAL RESULTS

In this section we present our experimental setup and evaluate the results taken from multiple runs of a large set of random and real-life graphs.

Experimental setup. Two different experimental setups were used. The first was a local server used for running randomly generated graphs of different sizes. The second was Euler Cluster [4] used to run real world graphs. Detailed information about both setups is shown in Table 1.

For the first setup we generated random graphs using the PaRMAT graph generator [5]. We generated graphs with a size of 1, 10 and 15 million vertices with a 1:3, 1:7 and 1:15 ratio to the edges of each graph. For each category 10 graphs were generated. For the second setup we ran each of the following real world graphs obtained from [6]:

- Californian Road Network with 1.9 million vertices and 2.8 million edges
- Google Web Graph with 0.9 million vertices and 8.6 million edges
- Livejournal Social Network with 4 million vertices and 69 million edges
- Orkut Social Network with 3 million vertices and 234 million edges

As mentioned in Section 3, we run all graphs through all the implemented approaches: first-fit, staggered first-fit and conflict-avoiding staggered first-fit, using both blocking and non-blocking communication.

Finally, all graphs were run through a serial implementation, to compare the speed and the validity of the results, and, through the parallel graph coloring algorithm implemented in the widely used Boost library [2]. Due to space constraints, we provide examples that illustrate our arguments. All of our observations apply for the complete test set we evaluated.

Results. In the experiments with the randomly generated graphs, there were no important observations to be made. As visible in Figures 1 and 2, even though there is a speed up compared to the sequential run time, it is not major and does not scale well. We believe that these results are mostly due to the fact that the graphs were not dense enough

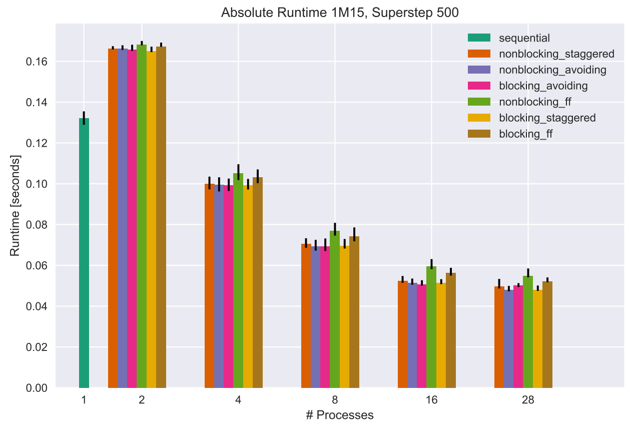

Fig. 1: Sample absolute runtimes for the randomly generated graphs

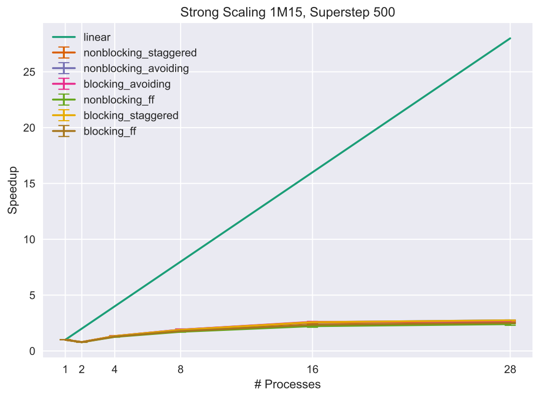

Fig. 2: Sample scaling for the randomly generated graphs

to produce any useful outcome, as mentioned in details below.

For the real world graphs, Figures 3 and 4 presents the number of colors that each implementation used. We observe that, in all cases, the first-fit approach is much more precise in comparison to all other approaches, and close, if not equal, to the Boost implementation. As discussed in Section 3, this was something expected as with the first-fit approach generally less colors are used, moving the weight to the higher conflict numbers. We do observe less conflicts using the advanced assignments, because they mostly do not reduce the number of rounds needed, which is what makes those approaches expensive as there needs to be another conflict resolution phase. Also, there is much more computational complexity involved in selecting the colors, and this being ”hot-path” code executed for every vertex makes it much more expensive than the simple first-fit assignment. In addition to this, there is an increase in the total number of colors used as the number of processors increases, as with

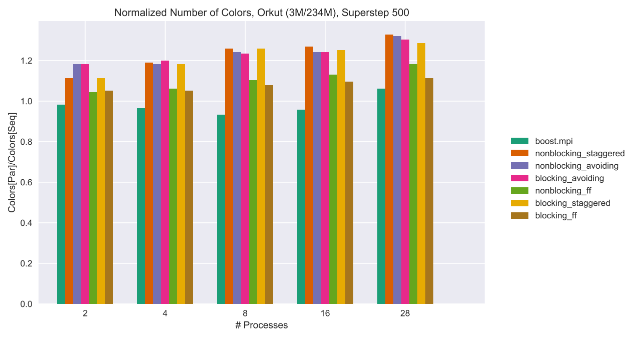

Fig. 3: Number of colors Orkut

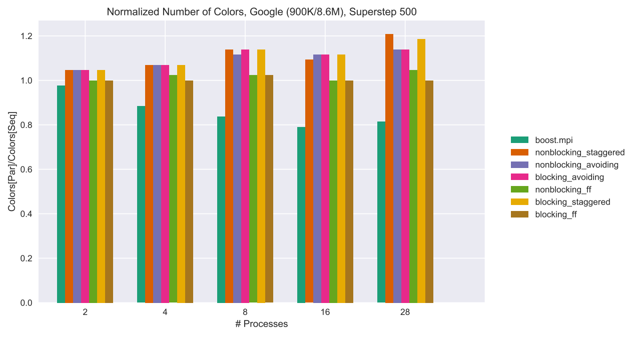

Fig. 4: Number of colors Google web graph

more processors, more conflicts arise, forcing the usage of higher number of colors to solve the higher number of conflicts.

In Figures 5 and 6 the run time of each implementation is shown. The first observation we make is that our implementation is reasonably faster than the serial implementation and extremely faster in comparison to the Boost's implementation. Moreover, we observe that all non-blocking approaches produce better results than the blocking approaches, and taking into account the results of the Californian Road Network graph, which is a sparse graph (1.9M nodes / 2.8M vertexes), we can see how the communication between cores affect the results. It is obvious that when the processors had to exchange more messages, in regards to the size of the graph, it heavily affected the result, even leading to higher run times more than the sequential implementation. Furthermore, there seems to be no impact on performance due to the use of the color assignment approach while we can observe a negative impact for the non-blocking approaches as the number of processors increases.

Blocking communication scales better than non-blocking communication. Initially the differences are negligible, but with more processors it starts becoming apparent. Non-blocking does introduce more conflicts because more vertices are colored with old information. With few processes it is possible to perform as good as blocking because even

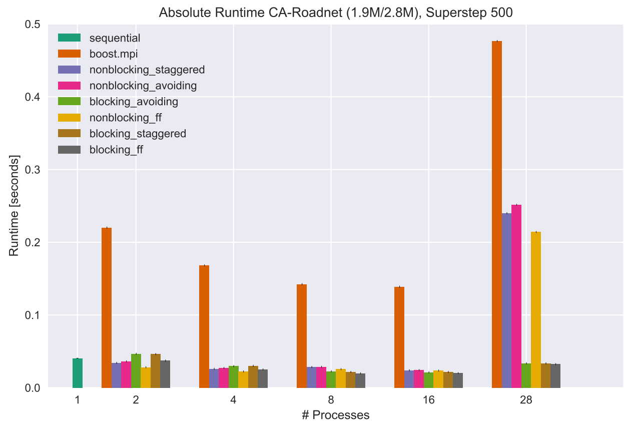

Fig. 5: Runtimes for Road Network

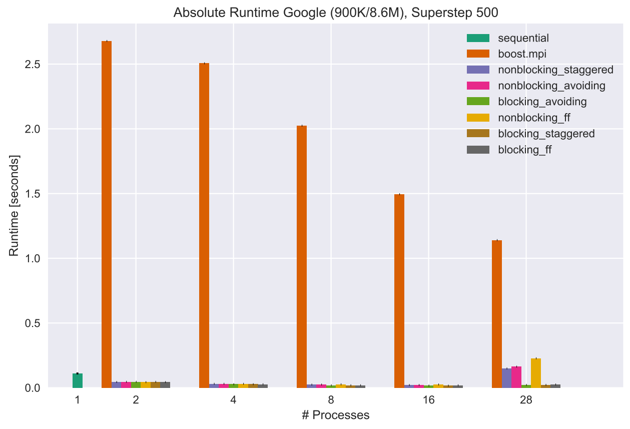

Fig. 6: Runtimes for Google Web Graph

though there are many more conflicts they can be hidden in the communication latency, but introducing more processors also introduces more inter-partition edges and thus even more conflicts. On the other side, when increasing the number of processors, the working set per processor decreases, and in total there are fewer communication phases, making the blocking communication approach more favorable.

Figures 7 and 8 show the scaling for the two denser real world graphs. The experiments show that the denser the graph, the better the scaling. There are even cases when the experiments show superlinear speedups. As for denser graphs, each processor has more computations to do in each step, meaning the time between communications is longer because they have to look up more colors from other vertices. In sparse graphs, the run time is completely dominated by communication latency. Nevertheless, our implementation is fast when looking at absolute run time. Scaling, though, stops at some point due to the communication and synchronization overhead becoming too large compared to the working set. Also, having more processors means that

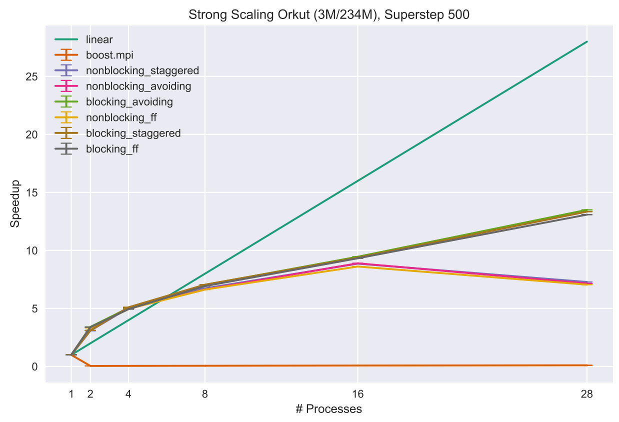

Fig. 7: Strong scaling Orkut network

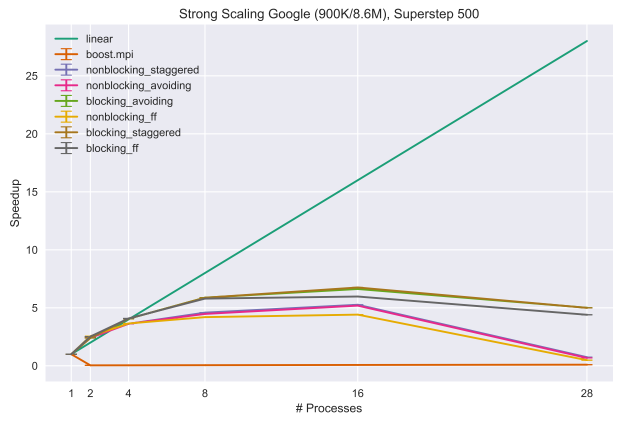

Fig. 8: Strong scaling Google web graph

there are more edges that cross partition boundaries, which in turn gives more conflicts.

In terms of the superstep size, the results show that it does not have a significant impact on the end result. In the original paper the superstep size was 100, while in our experiments 500 seems to be the best, and increasing it does not influence run times or scaling. It does introduce more conflicts, but this is balanced by the algorithm being able to proceed faster because there are fewer synchronization points and less time is spent waiting for communication to complete.

5. CONCLUSIONS

We presented a scalable implementation of Boman's graph coloring using MPI and C/C++. All our implementations outperform the existing implementation in the Boost parallel graph library on all our experiments. We observe no benefits of using more sophisticated color assignment algorithms to prevent conflicts, and conclude that the simple first-fit assignment is the best choice. To our own surprise using blocking communication proved to be more scalable than non-blocking communication. By focusing on a predictable and compact implementation we shifted the bottleneck even further towards the cost of communication, which is shown by the fact that our implementation scales best on dense and big graphs.

Several options for further optimizations exist: There are significant load-balancing issues starting from the second round of the algorithm because the processes have different numbers of conflicts. Additionally, we assumed a random partitioning of the graph to the processes. Repartitioning the graph to have minimal inter-partition edges could reduce the amount of conflicts.

6. REFERENCES

[1] Erik G Boman, Doruk Bozdağ, Umit Catalyurek, Assefaw H Gebremedhin, and Fredrik Manne, "A scalable parallel graph coloring algorithm for distributed memory computers," in *European Conference on Parallel Processing*. Springer, 2005, pp. 241–251.

[2] Douglas Gregor and Andrew Lumsdaine, "The parallel bgl: A generic library for distributed graph computations," *Parallel Object-Oriented Scientific Computing (POOSC)*, vol. 2, pp. 1–18, 2005.

[3] Assefaw Hadish Gebremedhin and Fredrik Manne, "Scalable parallel graph coloring algorithms," *Concurrency - Practice and Experience*, vol. 12, no. 12, pp. 1131–1146, 2000.

[4] "Euler - scientific computing," `https://scicomp.ethz.ch/wiki/Euler`, December 2017.

[5] Farzad Khorasani, Rajiv Gupta, and Laxmi N. Bhuyan, "Scalable simd-efficient graph processing on gpus," in *Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques*, 2015, PACT '15, pp. 39–50.

[6] Jure Leskovec and Andrej Krevl, "SNAP Datasets: Stanford large network dataset collection," `http://snap.stanford.edu/data`, June 2014.