Multilevel Parallelism for the Exploration of Large-Scale Graphs

Massimo Bernaschi, Mauro Bisson, Enrico Mastrostefano, and Flavio Vella

Abstract—We present the most recent release of our parallel implementation of the BFS and BC algorithms for the study of large scale graphs. Although our reference platform is a high-end cluster of new generation Nvidia GPUs and some of our optimizations are CUDA specific, most of our ideas can be applied to other platforms offering multiple levels of parallelism. We exploit multi level parallel processing through a hybrid programming paradigm that combines highly tuned CUDA kernels, for the computations performed by each node, and explicit data exchange through the Message Passing Interface (MPI), for the communications among nodes. The results of the numerical experiments show that the performance of our code is comparable or better with respect to other state-of-the-art solutions. For the BFS, for instance, we reach a peak performance of 200 Giga Teps on a single GPU and 5.5 Terateps on 1024 Pascal GPUs. We release our source codes both for reproducing the results and for facilitating their usage as a building block for the implementation of other algorithms.

Index Terms—Large graphs, graph algorithms, parallel algorithms, parallel programming, distributed programming, GPU, CUDA

1 Introduction

Nowadays, analysing *large*-scale graphs is a common activity not only in the study of communication networks but also in many other disciplines dealing with, for instance, social or protein networks. When graphs have several millions of nodes and hundreds million (or even billions) of edges, apparently simple tasks, like the visit of all the vertices, become costly operations. The problem is even worse in case of more complex analyses whose results may become outdated before being completed. High Performance Computing (HPC) techniques and, in particular, parallel processing may alleviate that situation provided that existing and new algorithms are modified to fit (or designed, if new) the features of modern hardware/software platforms for HPC. Among the most promising candidates to that role there are computing nodes equipped with Graphics Processing Units (GPU) working as *accelerators* of standard CPU and connected by means of low-latency and high-bandwidth networks. As a matter of fact, GPU may be used not only for classic arithmetic intensive applications with regular memory access patterns (e.g., linear algebra on dense matrices) but also for problems with limited, if any, numerical calculations and irregular memory access patterns, like most graph algorithms. In the present work we describe our approach to the analysis of large scale graphs based on clusters of GPUs. We exploit parallel processing at multiple levels through a hybrid programming paradigm that combines highly tuned CUDA kernels, for the computations performed by each node, and the Message Passing Interface (MPI), for the communications among nodes.

We present solutions and results for two problems: i) the Breadth First Search (BFS); and ii) the computation of the Betweenness Centrality (BC) score. We chose BFS since it is one of the most representative elements in a group of ubiquitous algorithms for traversing and analysing graph structure.

Unlike other important building blocks, like Depth First Search (DFS) where it is hard extracting parallelism, BFS can be implemented efficiently using parallel processing techniques. Concerning BC score, it is one of the most widely used metrics to measure the relevance of a node in a graph. Moreover, computing the BC score for all nodes in a graph requires information about the shortest paths between all pairs of nodes. Some of the techniques herein described can be applied also to a wider range of problems requiring the evaluation of Single Source Shortest Path (SSSP) and All Pair Shortest Path (APSP). In particular, the evaluation of BC score is an interesting paradigm of multi-scale graph parallel processing: it requires a large set of independent BFS operations (for unweighted graphs), that can be executed in parallel, and each BFS operation can be, in turn, carried out in a parallel fashion.

The rest of the paper is organized as follows: Section 2 introduces the notation and, in a more formal way, the problems we are going to study; Section 3 presents the latest version of our implementation of the BFS; Section 4 shows how the BC score can be determined by using the BFS as a building block; Section 5 describes the platforms we used for the experimental tests, the results we obtained and a comparison with other state-of-the-art solutions; finally, Section 7 concludes the paper indicating possible future related activities.

- *M. Bisson is with NVIDIA Corp., Santa Clara 95051, CA 95051. E-mail: maurob@nvidia.com.*
- *M. Bernaschi, F. Vella, and E. Mastrostefano are with the Istituto per le Applicazioni del Calcolo, National Research Council of Italy, Rome 00185, Italy. E-mail: massimo.bernaschi@cnr.it, enrico.mastrostefano@uniroma1.it, vella@di.uniroma1.it.*

Manuscript received 30 Mar. 2017; revised 18 Dec. 2017; accepted 9 Jan. 2018. Date of publication 23 Jan. 2018; date of current version 14 Sept. 2018.
(Corresponding author: Enrico Mastrostefano.)
Recommended for acceptance by A. Kalyanaraman and M. Halappanavar.
For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.
Digital Object Identifier no. 10.1109/TMSCS.2018.2797195

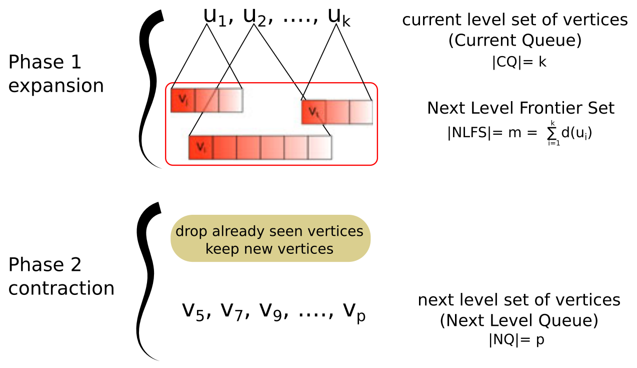


Fig. 1. BFS phases: In the first phase, all neighbors of each vertex in the current queue are inserted in the NLFS. The total number of elements in the NLFS is equal to the sum of the degrees of all the elements in the current queue. In the second phase, already visited vertices are removed from the NLFS. The remaining vertices are inserted in the next level queue.

The contributions of the present paper are manifold:

- an efficient threads-data mapping strategy which allows for achieving close-to-optimal performance regardless of graphs characteristics;
- for the BFS, an enhancement of the single-GPU performance (with respect to [1]) and results on up to 1024 new-generation GPUs;
- a flexible algorithm to compute the BC score in large-scale graphs by exploiting three different levels of parallelism (with new, enhanced, results with respect to [2]).

2 Background and Notation

Let $G = (V, E)$ be a graph representing a network composed by entities (vertices) and relations (edges), respectively. Formally, let $G = (V, E)$ be a undirected and unweighted graph with $n = |V|$ vertices and $m = |E|$ unordered pairs (u, v) such that $u, v \in V$ and $u \neq v$. Since the graph is undirected, we consider (u, v) and (v, u) to be the same edge. The degree of a vertex $deg(v)$ is the number of edges incident to it. A common way to represent a graph is to provide its *adjacency matrix*, a $|V| \times |V|$ matrix of 0s and 1s, where the entry in row i column j is 1 *iff* the edge (i, j) is in the graph. For an undirected graph, the adjacency matrix is symmetric. For sparse graphs, the corresponding matrices are sparse as well and thus are stored in a compressed form (*e.g.*, Compressed Sparse Row format) as a set of lists.

The Breadth First Search finds shortest paths (σ) from a given source vertex, (the *root* of the search tree) to all other vertices, in terms of the number of edges in the paths. Usually in a BFS two values are associated to each vertex v:

- a *distance*, giving the minimum number of edges in any path from the root to v.
- a *predecessor*, i.e., the parent of v along a shortest path from the root.

During the search it is easy to determine whether a vertex has been visited already: a vertex's distance has a special value (e.g., -1) until it has been visited, at which time it gets a numeric value for its distance. At each step, the BFS advances by processing the vertices visited in the previous step looking for connected vertices not yet visited (in the first step only the root node is processed). To keep track of the current set of vertices to be processed, it is possible to use a queue. The queue represents a *current-level set* of vertices, see Fig. 1. For each vertex in the current level, all its neighbors must be visited. The set of all neighbors composes the *Next Level Frontier Set* (NLFS). From the *NLFS* only vertices not yet visited are selected to form the queue for the *next-level set* of vertices. The BFS visit is divided into levels with a distance from the root that increases at each subsequent level.

The Betweenness Centrality score of a vertex v is the sum of the fraction of all-pairs shortest paths of the graph that pass through v. Formally, BC score of a vertex v is defined as a sum of pair-dependencies on v as follows:

$$BC(v) = \sum_{s \neq t \neq v} \delta_{st}(v). \tag{1}$$

The pair-dependency on v of a pair s, t is the ratio $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$. where σ_{st} is the number of shortest paths between vertices s and t and $\sigma_{st}(v)$, is the number of those shortest paths that pass through v with $s, t, v \in V$. An exhaustive evaluation of the BC score requires solving the Single Source Shortest Path problem starting from each vertex.

3 A Fast 2D Breadth-First Search

We described our approach to the Breadth First Search on large scale graphs using clusters of GPU in previous works [1], [3], [4]. Hereafter we recall just its two basic pillars: our solution for the data-threads mapping and our variant of the 2D partitioning technique.

3.1 Active-Edge Parallelism

We resort to a data-GPU threads mapping strategy that does not depend on specific characteristics (e.g., the degree distribution) of the graph. This strategy, we called Active-Edge parallelism, extends the edge-parallel [5] approach by assigning a thread to each outgoing edge from the vertices in the current frontier queue (CQ). In this way, there is no need to inspect each edge in the graph as the original edge-parallelism strategy does. However, it is necessary to count the total number of outgoing edges from the vertices in the frontier and then map each vertex to its neighbors. In detail, the degree of each vertex in the current frontier is stored into a contiguous array CD. A prefix-sum of the CD array is performed afterwards. At the end of the prefix-sum, CD contains the information required to identify the predecessor vertex associated with the i^{th} thread. In order to determine the predecessor, a binary search over the CD array is also required.

A simple example of this approach is illustrated in Fig. 2. This mapping achieves an optimal load balancing by dividing evenly the edges leaving the current frontier among the GPU threads. In general, it requires extra computations that we may minimize as shown in [1].

3.2 2D Partitioning

In a distributed memory implementation, the graph is partitioned among the computing nodes by assigning to each one a subset of the original vertices and edges. The search is performed in parallel, starting from the processor owning

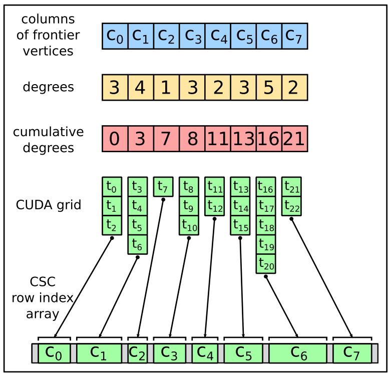


Fig. 2. Example of active-edge parallelism.

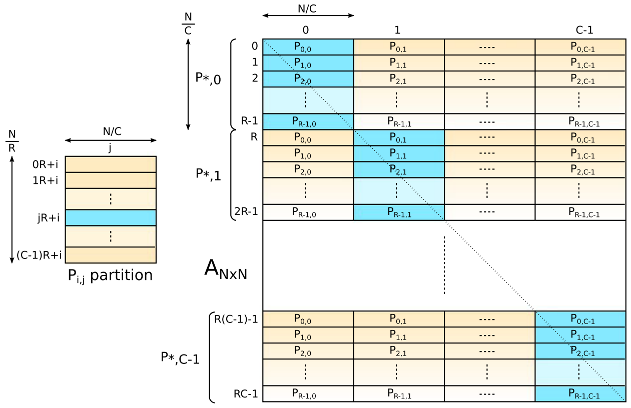


Fig. 3. Two-dimensional partitioning of an adjacency matrix A with an $R \times C$ processor grid. The matrix is divided into C consecutive groups of $R \times C$ blocks of edges along the vertical direction. Each block is a $N/(RC) \times N/C$ sub-matrix of A. Different groups of blocks are colored with different yellow gradients. For each block, the column of processors owning the corresponding vertices (row indexes) are shown in blue. On the left part, it is shown the sequence of blocks, from top to bottom, assigned to the generic processor (p_i, p_j). The colors correspond to the blocks assigned to the processor in each group of blocks.

the root vertex. At each step, processors handling one or more frontier vertices follow the edges connected to them to identify unvisited neighbors. Newly discovered vertices are then exchanged in order to notify their owners and the next iteration begins. The search stops when the connected component containing the root vertex has been completely visited. In [6], the computing nodes are arranged as a logical grid with R rows and C columns and mapped onto the adjacency matrix $A_{N\times N}$ (partitioning it into blocks of edges). The processor grid is mapped once horizontally and C times vertically thus dividing the columns in C blocks and the rows in RC blocks, as shown in Fig. 3.

Processor P_{ij} handles all the edges in the blocks $(mR+i, j)$, with $m = 0, ..., C-1$. Vertices are divided into RC blocks and processor P_{ij} handles the block $jR+i$. Considering the edge lists represented along the columns of the adjacency matrix, this partitioning guarantees that:

i) the edge lists of the vertices handled by each processor are partitioned among the processors in the same grid column;
ii) for each edge, the processor in charge of the destination vertex is in the same grid row of the edge owner.

With such decomposition, each step of the BFS requires two communication phases, called *expand* and *fold*. The first one involves the processors in the same grid column whereas the second those in the same grid row. At the beginning of each step, each processor has its own subset of the frontier set of vertices (initially only the root vertex). The search entails the scanning of the edge lists of all the frontier vertices so, due to property *(i)*, each processor gathers the frontier sets of vertices from the other processors in the same processor-column. For each gathered vertex u, all outgoing edges (u, v_i) are identified and sent to the processors in charge of each v_i. For property *(ii)* this exchange only involves processors in the same processor-row (fold exchange). On the receiving side, edges whose target vertex has already been visited are filtered out. Those that remain have their data updated and are added to the frontier set used for the next level expansion. The search ends when the frontier of each processor is empty, meaning that the whole connected component containing the root vertex has been visited.

The main advantage of the 2D partitioning is a reduction of the number of communications. In a simple 1D partitioning, entire adjacency lists would be assigned to processors so that after each frontier expansion every process may need to communicate with all the others in order to inform them of locally reached vertices. Conversely, with the adopted 2D partitioning, adjacency lists are partitioned, having P computing nodes, among $\sqrt{P}$ nodes (partitioning column) each one handling parts leading to vertices owned by $\sqrt{P}$ nodes (partitioning row). In this way the communication cost reduces from $O(P)$ to $2 \times O(\sqrt{P})$.

To reduce the size of the messages, it is possible to send a bitmap with the bits corresponding to the outgoing vertices set. This is convenient when the size of vertices lists to be sent exceeds, in bits, the number of indices local to the receiving process. Further details can be found in [1].

3.3 Direction-Optimizing

Recently, we improved the single-GPU performance of our BFS by implementing the *direction-optimizing* (DO) variant [7].

That technique carries out the expansion of the current frontier with two equivalent, yet conceptually opposite approaches, the so-called *top-down* (TD) and *bottom-up* (BU) expansions. The top-down approach is the traditional method in which the current frontier (*CQ*) is expanded by following all the edges leaving from its vertices in search of unvisited neighbors. With respect to the predecessor tree produced by the search, the expansion is performed from parents to siblings.

The bottom-up approach, on the other hand, expands the frontier by performing the search in the opposite direction, from siblings to parents. For each unvisited node, its neighbors are searched until one belonging to the current frontier is found, if any. All of the unvisited nodes with a connection in the current frontier are added to the next level frontier set.

It is worthy to note that not only in the top-down method the neighbors search is performed for each vertex in the frontier whereas in the bottom-up it is performed for each unvisited vertex but also, and possibly more important, that, while in the top-down case all of the edges incident to frontier vertices must be followed, since any one may lead to an unvisited vertex, in the bottom-up case the search for a parent can be stopped as soon as an edge leading to the frontier is found. As a matter of fact, that is sufficient to assign the node to the next level frontier.

For a more detailed description of the bottom-up technique see, for example, [7], [8], [9].

The BFS procedure described in 3.2 for a 2D decomposition of the adjacency matrix represents a top-down approach. The corresponding bottom-up version requires to change only the expand phase. As in the top-down case, initially each processor gathers the frontier vertices from the other processors in the same processor-column. After the gather operation, instead of scanning the edge lists of the received vertices (columns of the local matrix shown in the left part of Fig. 3), the processors scan the rows corresponding to unvisited vertices until an edge leading to the gathered frontier is found. Newly discovered vertices are then exchanged across the processor-rows in the fold phase.

Since the local matrix A can be scanned both column-wise and rows-wise, depending on whether the frontier expansion is performed top-down or bottom-up, we store A in two different formats in order to optimize the memory accesses for both cases. The top-down expansion is performed using the Compressed Sparse Column (CSC) format, which provides column data in contiguous memory locations. Analogously, for the bottom-up expansion we use the Compressed Sparse Row (CSR) format.

A direction-optimizing BFS employs both strategies and implements a heuristic to select the most efficient expansion for each search step. Devising a heuristic general enough to be used with different types of graph it is a non-trivial challenge (see, for instance, [10]).

Our code resorts to a simple heuristic based on the ratio between the current frontier size and the number of unvisited vertices, r_f, computed locally by each process. The decision on the type of expansion method to use is taken at every search step by each process, independently of both the methods used in the previous steps and the other processes. At the beginning of each BFS iteration, that ratio is compared to a predefined threshold R_{th}. If $r_f < R_{th}$ then the top-down expansion is used, otherwise the process uses the bottom-up. Clearly, unless the number of vertices assigned to the process in charge of the root vertex is less than $1/R_{th}$, each BFS starts with the top-down expansion.

The value of the threshold R_{th} depends on the graph being searched and thus its optimal value must be explicitly searched before to performing the actual BFS.

4 Betweenness Centrality

The fastest algorithm for calculating BC scores (due to Brandes) has $\mathcal{O}(nm)$ time-complexity and $\mathcal{O}(n+m)$ space-complexity for unweighted graphs [11]. The algorithm exploits the natural sparsity of real-world graphs, by introducing the dependency of a vertex v with respect to a source vertex s:

$$\delta_s(v) = \sum_{w:v\in pred(w)} \frac{\sigma_{sv}}{\sigma_{sw}} \cdot (1 + \delta_s(w)). \tag{2}$$

Formula 1 can be re-defined as sum of dependencies:

$$BC(v) = \sum_{s\neq v} \delta_s(v). \tag{3}$$

After that, it is possible to compute BC scores in $\mathcal{O}(nm)$ on unweighted graphs by:

1) computing σ, that is the number of SSSP of all vertices visited from a root vertex s;
2) summing all dependencies δ from s and update BC score;
3) repeating steps 1) and 2) for each vertex in G.

4.1 3D MGBC

Our Multi-GPU Betweenness Centrality (*MGBC*) algorithm consists in a flexible and efficient parallelization of Brandes' algorithm that exploits three different levels of parallelism:

1) at node-level: CUDA threads work on a subset of edges (fine-grained parallelism on a shared memory system) according to a strategy of data-threads mapping similar to that presented for the BFS.
2) at cluster-level: a set of processors (or accelerators like GPUs) works concurrently following a graph partitioning strategy. At this level, the performance depends on the communication network as well.
3) at subcluster-level: multiple sub-clusters work over replica of the same graph. Each sub-cluster performs the BC procedure on a subset of vertices concurrently to other sub-clusters. This further level of parallelism can be introduced since the betweenness equation is additive.

On a distributed-memory system, the second and third levels enable fine and coarse-grained parallelism. Like Brandes' algorithm, MGBC is composed of three main steps: $i)$ shortest paths counting; $ii)$ dependency accumulation; and $iii)$ update of BC scores. In details, concerning the first step of Brandes' algorithm, we perform a distance BFS to compute the number of shortest-paths that reach any visited vertex from a source vertex s. In the beginning of each step, each processor has its own subset of the frontier. The processors on the same column of the mesh exchange frontier vertices (vertical communication). After this operation, all processors on the same column share the same frontier. During the frontier expansion, new discovered vertices are marked as visited. Their σ values are updated by an atomic operation. The edges belonging to other processors are communicated together with the partial σ score (horizontal communication). At the end, the new frontier and σ values are updated. After shortest path counting, the depth/level array of each discovered vertex (d) is exchanged as well. This operation is performed once for each BC round between shortest-path counting and dependency accumulation phases. In contrast to the 2-D BFS case, during each *fold* phase in the 2-D BC algorithm, the σ values must be sent among processes. This limits the scalability of the BC algorithm with respect to the 2-D BFS building block. Rather than exchanging the predecessors list during the traversal step as usual in a distributed implementation of the Brandes' algorithm, we keep track of the local frontiers computed from each computing node. The combination of the local frontier and the distance array allows building the predecessor/successors information of the BFS tree without paying additional communication costs among processes. As a consequence, the communication cost depends on the number of vertices of the graph (i.e., the distance array) instead of the number of the edges. Furthermore, this solution also allows reducing

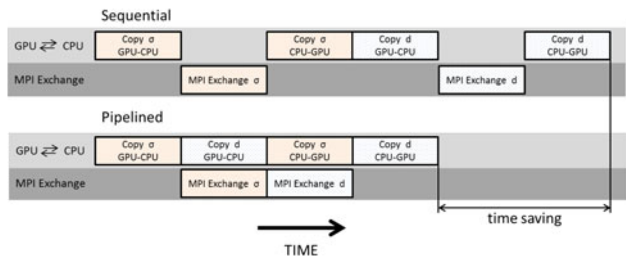


Fig. 4. Overlapping of GPU - CPU data transfer with MPI communication.

both the memory requirements of the local data structures from $\mathcal{O}(m)$ to $\mathcal{O}(n)$ and, as consequence, the number of `read` and `write` operations on GPUs memory.

MGBC accumulates the dependency following the checking successor approach proposed for the first time in [12]. That technique goes backward from the penultimate level towards the root considering the successors of the vertices rather than the predecessors. This has several additional benefits—improved locality on the writes and no need of locks regardless of the data-threads mapping choice. Both the distance information d and the σ of the vertices are exchanged among computing nodes in the same row. The dependency $\delta[w]$ is calculated by the shortest path count $\sigma[v]$ and dependency value $\delta[v]$ of all its successors. Each processor accumulates the local contributions to $\delta[w]$ for those successors for which it holds the edge (w, v). All the local dependency contributions are exchanged and summed by a reduce operation among the processors having the same index column of the mesh afterwards.

The final dependency value $\delta[w]$ is obtained by multiplying the accumulated dependencies over $\sigma[w]$. Finally, $\delta[w]$ values are exchanged among processors on the same row since they are required for the next iteration. An atomic operation is performed to update the local dependency $\delta[w]$.

4.1.1 *Communication Optimization*

The proposed distributed algorithm is amenable to the overlap of MPI communication and CPU-GPU data transfer. Although Nvidia provides several techniques to reduce communication overhead such as GPUDirect RDMA, we adopt a simple overlap mechanism between two consecutive communications, whereby the cost of the communication through the PCI bus can be hidden. In particular, right after the shortest path counting phase, both the distance vector d and σ values are exchanged among processors in the same grid row. Since the computation is totally delegated to GPU, usually two consecutive independent communications comply with the following pattern:

1) synchronous-copy of σ from GPU to CPU;
2) exchange of σ among processors in the same grid row;
3) synchronous-copy of σ from CPU to GPU.
4) synchronous-copy of d from GPU to CPU;
5) exchange of d among processors in the same grid row;
6) synchronous-copy of d from CPU to GPU.

In this naive pattern, data transfer procedure ends after six synchronous steps. However, by exploiting CUDA Asynchronous Copy operations and CUDA Streams the two communications can be completed in four steps (see Fig. 4):

1) asynchronous-copy of σ from GPU to CPU; asynchronous-copy of d from GPU to CPU;
2) exchange of σ among processors in the same grid row;
3) asynchronous-copy of σ from CPU to GPU; exchange of d among processors in the same grid row;
4) asynchronous-copy of d from CPU to GPU.

4.2 Sub-Clustering

A Multi-Source approach for the BC computation offers a significant speed-up on a single-GPU, provided that enough extra-memory (e.g., for the replication of σ and δ arrays) is available [13]. In addition, on distributed systems the replication of data-structures may increase the communication among computing nodes and increase the synchronization requirements. For example, Buluc et al. [14] provides a solution which encapsulates three levels of parallelism: columns of F provide parallelism over starting vertices, columns in M' and rows of F provide parallelism over the vertices in each frontier. Finally, rows of M' encapsulate edge (adjacency) parallelism of each frontier vertex. However all the processors in the mesh are involved in the communication during traversal steps. Therefore, to the best of our knowledge, using the single-GPU Multi-Source approach as a basis for a fully distributed BC algorithm does not appear to be the best option. On the other hand, on distributed systems, a coarse-grained approach enables to obtain a very good speed-up by replicating the data structures among computing nodes in order to work on multiple vertices at the same time. As mentioned above, this approach limits the maximum size of the graph that can be processed (*e.g.,* the Friendster graph cannot be stored in the memory of a single GPU). However, it is possible to obtain a significant improvement of performance by combining fine- and coarse-grained approaches at cluster level abstraction. Within this context, we propose a new solution which combines graph distribution and graph replication on a Multi-GPU system. Although in the present work we employ it for the evaluation of the BC score on Multi-GPUs systems, the approach is more general and can be followed for most problems (i.e., diameter computation, all-pairs-shortest-paths, transitive closure, etc...) that require multiple, independent breadth-first search operations on graphs too large to fit in a single computing node. A set of processors is split into sub-clusters. Each sub-cluster, in turn, is organized as a bi-dimensional grid of processors. Processing nodes in the same sub-cluster work at the fine-grained level: the graph is distributed among the nodes according to a 2-D partitioning, and partial BC values are calculated starting from a subset of vertices. Independent sub-clusters work at the coarse-grained level: the whole graph and additional data structures are replicated in each sub-cluster. In the end, a reduce operation updates the final BC scores. Even if the amount of work in each sub-cluster can be different when processing graphs with multiple connected components, with the sub-clusters solution it is possible to take advantage of both fine- and coarse-grained approaches (see Section 5). Let p be the number of processors available/requested in the cluster, and let fd be the factor of graph distribution (indicating the size of the mesh of the sub-cluster). The factor of replication of the graph (fr) is defined by $fr = \frac{p}{fd}$ and, in our implementation, it determines the number of sub-clusters. A simple example is shown

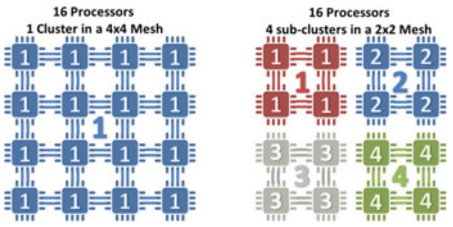


Fig. 5. Sub-clustering. On the left side the configuration ($p = 16$, $fd = 1$, and $fr = 1$) enables to solve SSSP problem by exploiting fine-grained strategy. On the right side, a sub-cluster configuration with $p = 16$, $fd = 4$, and $fr = 4$ exploits fine-grained and coarse-grained parallelism to solve all-pairs shortest path problem.

in Fig. 5. With respect to existing solutions [14], where all p processors are involved on the communication, sub-clustering technique involves only fd processors in a subcluster during traversal steps (except for the final reduction operation). Furthermore, our approach is not bound to a 2-D partitioning, so other partitioning strategies can be adopted. Both the fd and fr factors must be taken into account to achieve the best performance. Concerning practical aspects, we implement this solution by creating a hierarchy among processes managed by different MPI communicators.

5 Experimental Results

In this Section we report the results of our BFS (*bfs2d* in the following) and MGBC codes on multi-GPU systems. We start by introducing the testing environment, then we describe and discuss the performance in different experiments of *bfs2d* and MGBC in Sections 5.3, 5.4 respectively.

5.1 Pascal Architecture

Pascal, as its predecessor Kepler, is an Nvidia architecture based on the CUDA specifications. Each Pascal Stream Multiprocessor (SM) has either 64 or 128 CUDA cores and the same number of registers *per* SM as previous generation GPUs (Maxwell and Kepler), but overall many more registers, since it features far more SMs. Pascal increases performance not only by adding more SMs, but also by making each SM more efficient. Atomic memory operations are important in parallel programming, allowing concurrent threads to correctly perform read-modify-write operations on shared data. Kepler significantly increased the throughput of atomic operations to global memory compared to the earlier Fermi architecture. Pascal further improves atomics by providing a 64 bits atomic add instruction for values in global memory. Previously, 64 bits atomic addition had to be implemented using a compare-and-swap loop, which is generally slower than a native instruction. However, the most significant improvements are in the memory. Pascal features *stacked* memory, a technology which enables multiple layers of DRAM components to be integrated vertically on the package along with the GPU. Pascal is the first GPU to use High Bandwidth Memory 2 (HBM2). HBM2 memory provides much greater bandwidth, more than twice the capacity, compared to current off-package GDDR5. The peak bandwidth is 720 GB/s which is 3 times higher than the Tesla M40 memory bandwidth. Table 1 shows the main features of the GPUs used in our numerical experiments.

TABLE 1
Main Features of the Nvidia GPU Used for the Numerical Experiments

Feature	K20x	K80	P100	Titan X
Architecture	Kepler	Kepler	Pascal	Pascal
N. of SM	14	13	56	28
N. of cores	2688	2496	3584	3584
N. of regs (*per* SM)	65536	65536	65536	65536
GPU clock rate	732 Mhz	824 Mhz	1329 Mhz	1911 Mhz
L2 cache size	1.5MB	1.5MB	4MB	3MB

5.2 Piz Daint Supercomputer

The tests on distributed systems have been carried out on the Piz Daint cluster hosted at the Swiss Center for Scientific Computing (CSCS). Each compute node is equipped with an Intel Xeon E5-2690 (v3 @ 2.60 GHz, 12 cores, 64 GB RAM) and NVIDIA Tesla P100 (16 GB of main memory). The interconnection network is built upon the Aries routing and communication ASIC device, and uses a Dragonfly network topology [15].

5.3 BFS Performances Analysis

We ran a number of tests on both synthetic and real world graphs in order to measure the performance of *bfs2d* and in particular the advantage provided by the recently added bottom-up variant.

Synthetic graphs were generated via the widely used RMAT model whereas real graphs, summarized in Table 2, have been collected from different sources of graph datasets by the authors of *gunrock* that let them be available along with their software for graph analytics [16], [17]. The random generated graph *rgg* has been downloaded from the DIMACS10 challenge [18].

For RMAT graphs having an edge factor variable from 16 to 256, we found that using the bottom-up traversal results in a speedup ranging from 2x to 5x, respectively (Fig. 7). With real world graphs we obtained an average speedup of 1.7x (Fig. 6).

In Fig. 6 we compare the single GPU performance of *bfs2d* with the latest version of *gunrock* [16] and with the

TABLE 2
Features of the Graphs Used to Test the Performance of the *bfs2d* Code

Graph	Vertices	Edges	D
hollywood-2009	1.1M	112.8M	11
indochina-2005	7.4M	302M	26
soc-LiveJournal1	4.8M	85.7M	16
soc-orkut	3M	212.7M	9
road-USA	23.9M	577.1M	6809
rgg-n-24	16.8M	265.1M	2622
rmat_S19_EF16	0.52M	8.3M	6
rmat_S22_EF16	4.19M	67.1M	7
rmat_S19_EF256	0.52M	134.2M	6
rmat_S22_EF256	4.19M	1073.7.1M	7
rmat_S29_EF256	536.8M	137438.95M	6

RMAT graphs are random generated graphs with power-law degree distributions, rgg is a random generated mesh-like graph, and all other graphs are real. ***D*** *denotes the diameter of the graph.*

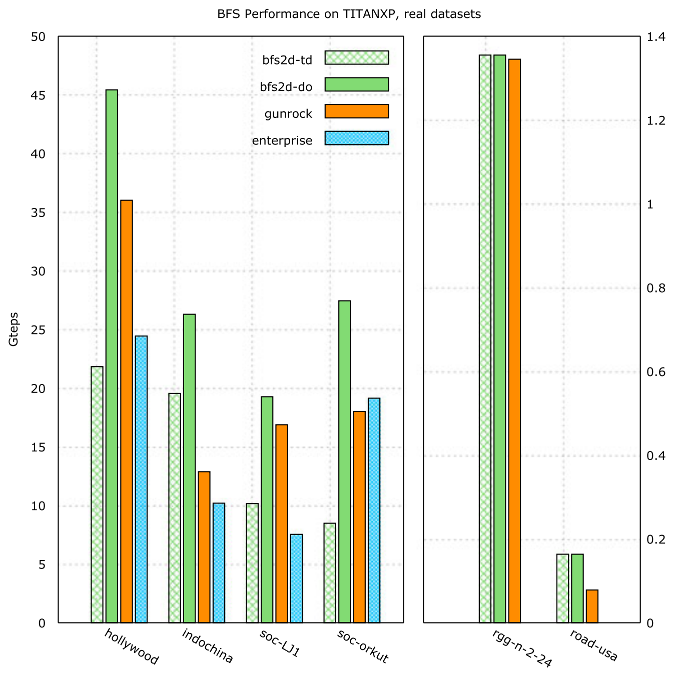


Fig. 6. Single GPU performance of *bfs2d* on graphs from Table 2. The bars show the performance of our code with only the top-down traversal enabled (td) and with the direction-optimization enabled (do). For comparison, we report the performance of *gunrock* and *enterprise* on the same graphs (see the text for further details).

enterprise bfs implementation from [8]. We ran the codes on the same GPU, an Nvidia Titan X Pascal with 12 GB of main memory. For each graph in the dataset, the mean number of Traversed Edges Per Second (TEPS) is computed over 64 BFS operations started from 64 randomly selected root vertices. In order to make the comparison as fair as possible, we used the same sequences for all the codes and, for what concerns *gunrock*, we used the command line options reported in the performance section of the *gunrock* web-site for the Pascal GPU [17]. All the command lines enable the direction-optimizing feature of *gunrock* using different traversal methods and direction thresholds. For *enterprise* we

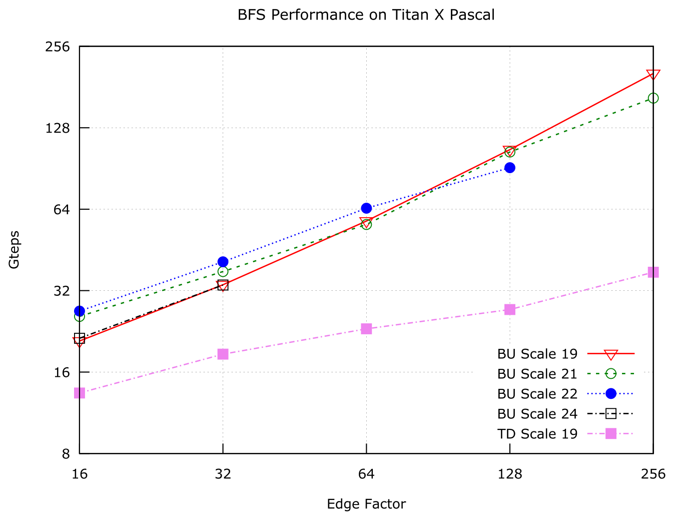


Fig. 7. Performance of *bfs2d* on RMAT graphs with different scales, varying the edge factor. We report the result also for scale 19 using only the top-down traversal mode. The improvement using the bottom-up traversal increases with the edge factor.

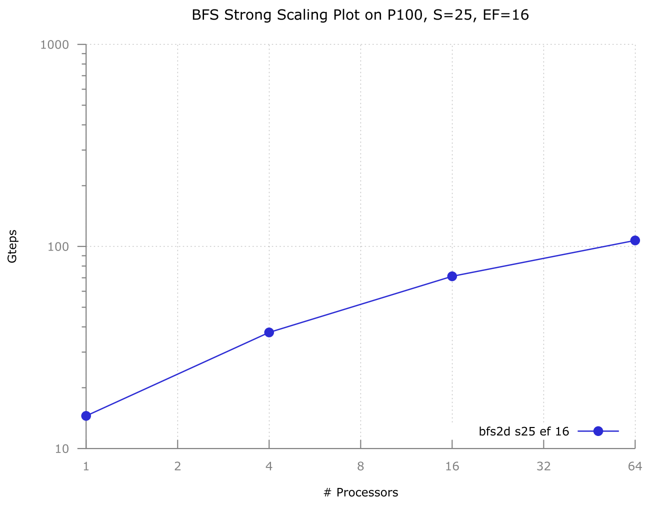


Fig. 8. Strong scaling plot of *bfs2d* on RMAT graphs. The scale of the graph is fixed to 25 and the edge factor to 16. The number of processors varies from 1 to 64. The axis are in log-scale.

followed the instructions provided along with the latest version of the source code available on *github* [19].

The results are reported in terms of directed edges per second, as done by *gunrock* for any type of input graph.

The *bfs2d* code is on average 1.4× faster than *gunrock* and shows better performance on all the real world graphs we tested. It is also worth noting that *bfs2d* computes the predecessors while we were forced to run *gunrock* without the support for predecessor calculation since it fails for some graphs in the dataset.

Actually, on *rgg* and *road-usa* our code does not use the bottom-up traversal. This is not surprising for graphs with a very long diameter and confirms that our heuristic to switch between the two traversal modes, described in Section 3.3, acts correctly in those cases.

In Fig. 7 we explore the performance of *bfs2d* on synthetic graphs with large average degree. For such graphs we expect to find the best performance, the short diameter and the power-law degree distribution permit to take full advantage of the GPU parallelism and the bottom-up technique. Fig. 7 shows the performance of *bfs2d* for different graph size, varying the edge factor. Despite being primarily designed for distributed systems, our code achieves more than 200 GTEPS on a single GPU.

For the sake of completeness, in Fig. 8, we also report the strong scaling plot of *bfs2d* for an RMAT graph with $S = 25$ and $EF = 16$ varying the number of processors from 1 to 64 on a NVDIA P100 GPU. We chose the scale corresponding to the largest graph that fits on the memory of a single GPU.

In a recent work [20] we presented a detailed analysis of the computation and communication phases of our code. Since most of the code is the same, here we present a more brief analysis of the results obtained on the same system, Piz Daint, whose Kepler K20 GPUs have been recently replaced by Pascal P100. Fig. 9 shows a weak-vertex scaling plot of *bfs2d*, whereas Fig. 10 reports, for one node in the cluster, the breakdown of the time spent in computation and communication. The breakdown refers to the same graphs used for the weak-scaling plot (ef 256). Fig. 9 shows a good scaling up to 1024 GPUs both for small values of the edge factor, e.g., 16 and for large values, e.g., 256. From

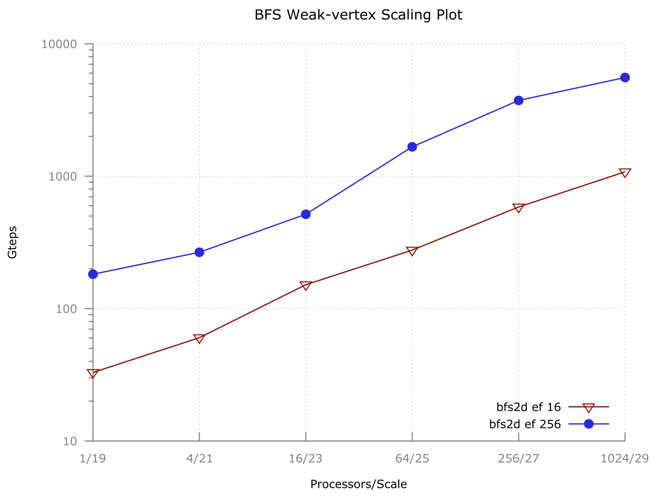


Fig. 9. Weak-vertex scaling of *bfs2d* on RMAT graphs. The experiment starts with scale 19 on 1 processor and then the number of processors and the size of the graph are doubled up to 1024 processors. The performance in GTEPS is reported for two different values of the edge factor: 16 and 256.

Fig. 10, it is apparent that using 1024 GPUs communications become more expensive than computations. This behaviour limits, very likely, the scalability with higher numbers of GPUs, although we did not have the chance to test the code using more GPUs. A direct comparison with the results of our previous work is difficult because we used RMAT graphs with different scales and edge factors. However, while in [20] we achieved a peak performance of 830 GTEPS with 4096 GPUs (330 with 1024 GPUs), in the present work the code has a peak performance of 5.5 TeraTEPS with 1024 GPUs. This improvement results from the combination of advantages given by the direction-optimizing approach and the new Pascal GPUs. To give an insight about the value of those figures, that performance would place the code in the top-ten (7th position) of the Graph500 benchmark according to the November 2016 ranking.

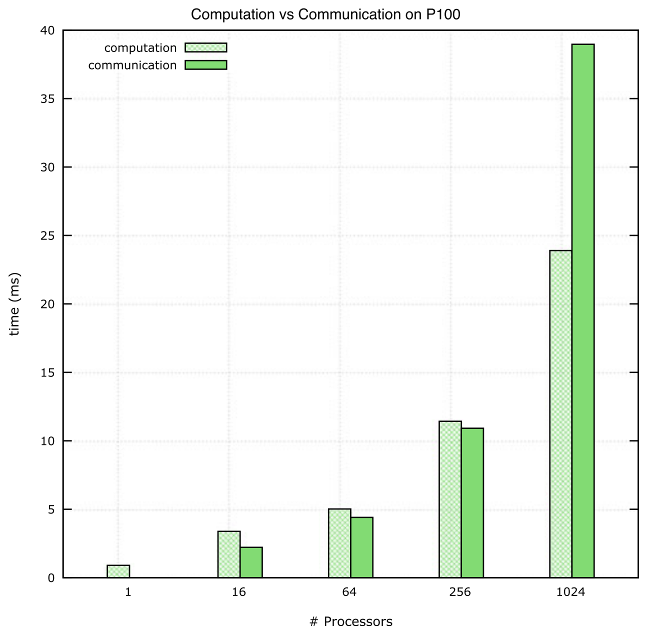


Fig. 10. Breakdown of the time spent in computation and communication of a single processor during one BFS traversal. The graphs are the same used for the weak-vertex scaling plot (Fig. 9, ef 256)

TABLE 3
Features of Real-World Graphs Used for the Experiments of BC Where **D** Denotes the Diameter of the Graph

Graph	Vertices	Edges	D
com-amazon	334,863	925,872	44
web-Google	875,713	5,105,039	21
RoadNet-PA	1,088,092	1,541,898	786
com-youtube	1,134,890	2,987,624	20
soc-Pokec	1,632,803	30,622,564	11
wiki-Talk	2,394,385	5,021,410	9
com-Orkut	3,072,441	117,185,083	9
com-LiveJournal	3,997,962	3,4681,189	17
Friendster	65,608,366	1,806,067,135	32

5.4 MGBC

We present performance results for the two configurations of MGBC (MGBC-2D and MGBC-3D). MGBC-2D implements a bi-dimensional partitioning of the graph [2]. We first evaluate the impact of *active-edge parallelism* over a variety of real-world graphs (see Table 3) [21] on a single GPU. Afterwards, we present strong and weak scaling results varying the fd and fr parameters over both R-MAT [22] and *Friendster* graph. For some experiments, we also report the performance of MGBC using the old configuration of "Piz Daint" based on K20 GPUs.

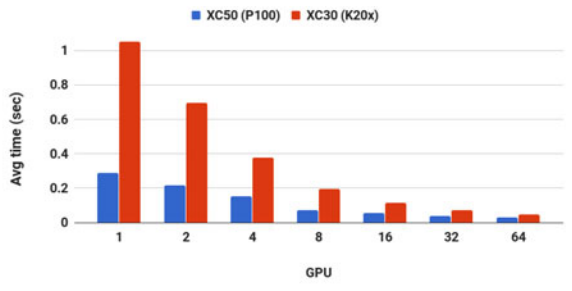
MGBC runs on a single GPU by simply turning off network and related host-device communications. In Table 4, we report the mean time (in seconds) for the Pascal and the Kepler architectures. The mean time is computed over 1,000 randomly selected starting vertices. By comparing P100 against K80 the speed-up achieved ranges from 1.1 over long diameter and small average degree graph (RoadNet-PA) up to 3.4 (Orkut) with an average of 2.3. The limited improvement reported for RoadNet-PA is mainly due to an under-utilization of CUDA threads.

For the computation of the BC score, it is possible to reduce the overhead due to the management of *active-edge parallelism* by removing the prefix-sum in the dependency accumulation. We remark that the dependency accumulation procedure visits the same frontiers of the first phase of Brandes algorithm but in reverse order. By storing the offset array CD during the shortest path counting, we can avoid

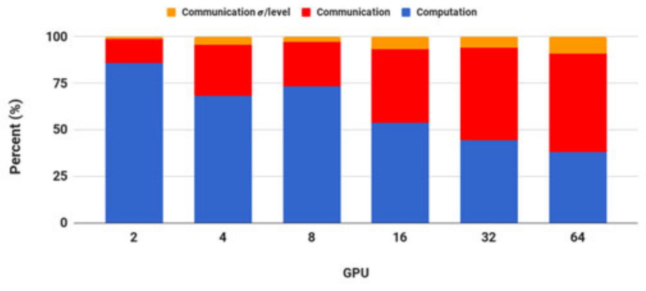
TABLE 4
Single GPU Results

Graph	P100	K80	P100 OPT-OFF	K80 OPT-OFF
com-amazon	**0.005**	0.008	0.007	0.010
web-Google	**0.007**	0.016	0.008	0.017
roadNet-PA	**0.065**	0.069	0.085	0.094
com-youtube	**0.006**	0.013	0.007	0.014
soc-pokec	**0.022**	0.075	0.023	0.077
wiki-Talk	**0.012**	0.025	0.013	0.027
com-orkut	**0.093**	0.315	0.094	0.355
com-LiveJournal	**0.037**	0.101	0.038	0.114
RMAT S23-E32	**0.291**	1.035	0.303	1.038

The values in the columns denote the average time in seconds of a BC round on two different GPU architectures. "OPT-OFF" denotes MGBC implementation with prefix-sum optimization.

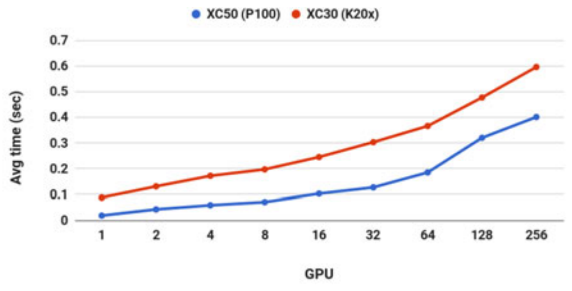


(a) Average time in seconds for the evaluation of the BC score of 1000 randomly selected vertices. The red bar refers to MGBC performance on the old configuration of "Piz Daint" (Cray XC30).

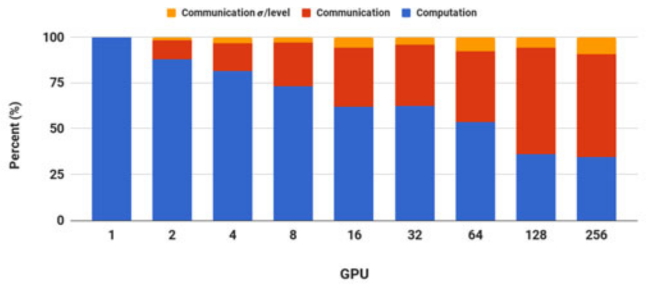


(b) Computation vs Communication cost (in percentage) using the new "Piz Daint" (Cray XC50).

Fig. 11. MGBC-2D strong scaling experiment on R-MAT scale=23 and ef=32.



(a) Average time in seconds for the evaluation of the BC score of 1000 randomly selected vertices. The red line refers to MGBC performance on the old configuration of "Piz Daint" (Cray XC30).



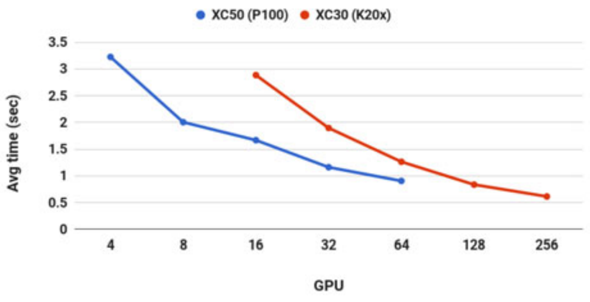
(b) Computation vs Communication cost (in percentage) using the new "Piz Daint" (Cray XC50).

Fig. 12. MGBC-2D weak scaling experiments. The scale of R-MAT ranges from 20 (1 GPU) to 28 (256 GPUs) ef=32.

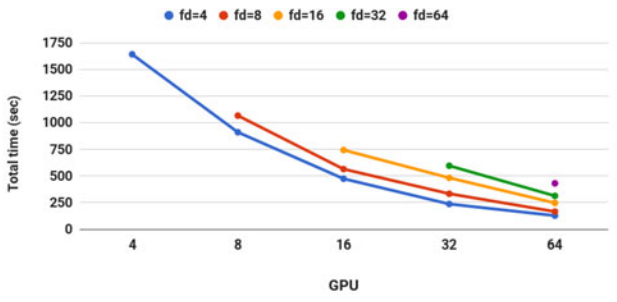
to perform again the prefix-sum. This solution allows reducing the computation time during dependency accumulation by reading CD stored in the previous step. Obviously, this time-saving has an extra memory cost that is, at most, $\mathcal{O}(n)$. Concerning prefix-sum optimization analysis, in general, the improvement is more significant on graphs with long diameter since the prefix-sum is performed for each level and those graphs require more iterations. On the other hand, for R-MAT graphs characterized by short diameter, the prefix-sum is more efficient and achieves the maximum throughput [23]. The experiment on the *Orkut* graph (average degree ~ 38 and diameter 9) represents the case where the prefix-sum is efficient but its cost is relevant. As a matter of fact, the cost of the prefix-sum is more relevant when the algorithm traverses the (middle) levels where most of vertices are found.

On distributed systems, we begin our study by evaluating MGBC-2D performance on strong scaling experiments. The fr parameter is kept equal to 1. In Fig. 11a, we report the mean time for each BC round, while the fd parameter (number of GPUs) ranges from 1 to 64. On the the K20x (red bar), MGBC-2D is able to scale up to 128 GPUs whereas on the P100 (blue bar), albeit it achieves better absolute performance, MGBC-2D scales only up to 32 GPUs since afterwards the communication dominates the computation (see Fig. 11b). In weak scaling experiments, the amount of data is the same for each GPU, the time required to compute BC is not constant. Contrary to the BFS algorithm, where only one vertex is marked as predecessor, during the traversal steps, the evaluation of the BC requires the counting of all shortest paths. When in the current frontier there are more vertices which expand the same successor, the time required to count the shortest paths and to accumulate dependency contributions increases due to atomic operations. In Fig. 12a, we compare two different configurations of Piz Daint by reporting the average time for a BC round. Also in this case, the major improvement is due to the better performance of the P100, therefore the maximum speed-up is achieved with 1 GPU (4.6x of speed-up). When the communication dominates the computation (see Fig. 12b), we observe a smaller difference in terms of performance (only 1.3x with 256 GPU over a R-MAT graph scale 28). However the scalability trend is basically the same. Note that the fine-grained approach does not allow to compute the exact score of BC for large-scale graphs in a reasonable amount of time. For example, the total time required to compute the exact betweenness centrality of *Friendster* graph is about one year by exploiting 256 K20x and 64 P100 respectively, as shown in Fig. 13a. In Fig. 13b, we report the strong scaling of different configurations of MGBC-3D by varying the number of replicas. In details, we compare the performance of different factors of distribution for the *Friendster* graph. The experiment shows that MGBC-3D exhibits a close-to-linear scaling with different configurations. We observe this trend even with more GPUs.[1] As expected, by fixing the total number of GPUs (X-axis), the best performance is achieved by using the smallest fd parameter. For example, by exploiting 64 GPUs MGBC-3D with $fd = 4$ (in blue) and $fr = 16$ is 3.4x times faster than MGBC-3D with $fd = 64$ and $fr = 1$ (violet spot).

1. We evaluted our solution up to 1024 GPUs ($fr = 64$).



(a) MGBC-2D strong scaling. Average time in seconds for the evaluation of the BC score of 1000 randomly selected vertices. The red line refers to MGBC performance on the old configuration of "Piz Daint" (Cray XC30). The expected total time can be extrapolated by multiplying the average time for a BC round and the number of no disconnected vertices.



(b) MGBC-3D strong scaling. The Y-axis shows the total time required to perform 512 BC rounds. The number of replicas (fr parameter) can be derived by dividing the total number of GPUs (reported in the X-axis) and the fd parameter reported for each line. For instance, the blue line shows the strong scaling of MGBC-3D with $fd = 4$ when the number of replicas increases up to 16.

Fig. 13. MGBC strong scaling experiment on a large-scale real-world graph (Friendster). The minimum number of GPU required to store the graph and data-structures is 4 and 16, respectively.

6 Related Works

A number of studies focused on the BFS algorithm both on shared and distributed architectures also including GPUs. Agarwal et al. [24] demonstrated poor scaling due to atomic operations on multi-socket systems. To reduce that cost, the authors proposed a combination of the fine-grained approach (edge partitioning among the sockets) and the accumulation-based approach in edge traversal. In details, each socket atomically updates only the information of the local vertices into a bitmap. They achieved good scaling up to four sockets. On single GPU, several studies addressed the problem related to the data-thread mapping strategy. The easiest approach assign to each thread one element of the BFS queue. On power-law graphs, such approach suffers from thread unbalancing and poor performance [3], [5], [25].

Hong et al. overcame the difficulties due to the vertex-parallelism by adopting a warp centric programming model [26], [27]. In their implementation each warp is responsible of a subset of the vertices in the BFS queue.

The approach proposed by Merrill et al. [28] assigned a chunk of data to a CTA (a CUDA block). The CTA works in parallel to inspect the vertices in its chunk. Furthermore, they used heuristics for avoiding redundant vertex discovery (*warp culling*). A similar approach was efficiently adopted by Gunrock for their direction-optimizing BFS implementation [9].

Beamer et al. [7] proposed the bottom-up approach to efficiently traverse the graph. This technique can be used to speedup the frontier expansion when the frontier is larger than the set of unvisited vertices. In those cases it is more advantageous to process unvisited vertices searching for a connection to the frontier rather than following each edge leaving the frontier. That situation is commonly found when visiting sparse graphs with power-law degree distribution. A combination of the classic top-down and the new bottom-up approaches (often referred as *direction-optimizing*) can be found in several subsequent works on both CPU and GPU [9], [8], [10]. Subsequent works also aim at finding heuristics to detect the optimal switching point between the two approaches.

By now, several graph analytics solutions are available [8], [29], [30], [31], [32]. Medusa [30] is a GPU-based graph processing framework based on the Pregel programming model [29] which allows to exploit more than one GPU in a box by inserting specific primitives in a simple C/C++ code. The CuSha [31] framework introduced new data structures in order to alleviate the burden of uncoalesced global memory data accesses. It employs G-Shard, an alternative format to the classic CSR data structure. The authors also proposed a novel approach, named Concatenated Windows, that increases CUDA threads utilization for very large graphs.

Liu et al. [8] designed an efficient GPU implementation of the bottom-up technique combining a workload balancing method with an original utilization of different GPU memories to mitigate random accesses. They also developed a switching technique (from top-down to bottom-up) based on the number of hub vertices in the frontier.

Shi et al. [32] proposed a new lock-free asynchronous framework, called Frog, with the purpose of reducing the synchronization cost of BSP model-based frameworks. Large scale graphs are partioned by using an approximated graph coloring algorithm, so that in each partition there are enough vertices to sustain a high level of concurrency.

On distributed memory systems several works based on an algebraic approach have been proposed [33], [34], [35], [36]. Satish et al. [37] implemented a distributed BFS with 1-D partitioning that shows good scaling with up to 1024 nodes. They described a technique to postpone the exchange of predecessors at the end of the traversal step. A similar technique was also reported and validated in other works [4], [38]. Ueno et al. [35] presented a hybrid CPU-GPU implementation of the Graph500 benchmark, using the 2D partitioning proposed by Yoo et al. [6]. Their implementation uses the technique introduced by Merrill et al. [28] to create the edge frontier. Furthermore, in order to reduce the size of the messages, they used a novel compression technique. Finally, they also implemented a sophisticated method to overlap communication and computation in order to reduce the working memory size of the GPUs.

Edmonds et al. provided the first hybrid-parallel 1D BFS implementation that uses active messages [39].

Petrini et al. [40] implemented a distributed-memory parallelization of BFS for the BlueGene architectures. Their results show that the combination of the underlaying architecture and the System Programming Interface (SPI) allows achieving significant performance on R-MAT graphs. They also reported that the SPI implementation outperforms the MPI one by a factor of 5. Bisson et al. [1] developed a fast 2D BFS implementation which exploits Nvidia Kepler capabilities. Their code achieved 830 billion edges per second on an R-MAT graph by exploiting 4096 GPUs.

To the best of our knowledge, the fastest distributed implementation runs on the K-Supercomputer and it is described in [41].

There are several parallel implementations for the computation of betweenness centrality on shared-memory systems. Madduri et al. [12] maintain successors instead of predecessors in the dependency accumulation step. In that way, the dependency accumulation procedure can start from one depth-level closer to the root vertex of the BFS tree and generally it does not require atomic operations. Prountzos and Pingali [42] developed a BC formulation based on the operator formulation and the Galois abstraction [43]. Green and Bader [44] reduced space complexity of structures local to a thread from $\mathcal{O}(m)$ to $\mathcal{O}(n)$. Tan et al.[45] described a new data structure for eliminating conflicts among threads in order to avoid atomic operations. Jia et al. [5] evaluated *vertex-parallel* and *edge-parallel* for Betweenness Centrality computation on GPU. Several authors proposed different strategies in order to exploit the advantages of both the methods [13], [46], [47]. Sarıyüce et al., introduced the vertex virtualization technique based on a relabeling of the data structure (e.g., CSR, Compressed Sparse Row) [46] [13]. That technique replaces a high-degree vertex v with $n_v = \lceil \mathtt{adj(v)} \rceil / \Delta$ virtual vertices having at most Δ neighbours. The vertex virtualization technique is not very effective for graphs with low average degree. Moreover, it requires a careful tuning of its parameters. The authors also proposed a coarse-grained approach in which a single GPU executes multiple BFS at the same time with an increase of memory requirements.

On distributed systems, the coarse-grained parallelism requires replicating the input graph and additional data structures so that each compute node can compute in parallel the BC score of a subset of vertices. At the end, a collective reduce updates final BC scores. McLaughlin and Bader [47] combined vertex- and edge-parallelism on GPUs and offer nearly perfect scaling. However, this approach does not work for graphs that do not fit in the memory of one node.

Edmonds et al. [48] developed a space efficient distributed algorithm. Buluc et al. [14] use 2D graph partitioning for BC. Contrary to their approach we exploit coarse-grained parallelism in distributed systems in order to reduce the inter-node synchronization. Their solution solved the exact BC computation exploiting a Multi-Source BFS algorithm for the shortest path counting based on the linear algebra approach [49]. A performance comparison of 1D and 2D partitioning for the Betweenness Centrality was given by Bernaschi et al. [50]. For graphs with n vertices and average degree k, Solomonik et al. [51] showed that on p processors, their 3D algorithm requires a factor of $p^{1/3}$ less communication than known alternatives when $k = n/p^{2/3}$.

Gunrock library [52] also provides an implementation of Brandes' algorithm on a single Multi-GPU computing node. Their BC implementation is 2.5 faster than the single GPU version by exploiting 6 GPUs and 1-D partitioning [53].

7 Conclusions

We described our approach to the solution of two of the most relevant problems in graph analysis: the Breadth First Search and Betweenness Centrality measure. We resort to a suitable combination of parallel programming paradigms: CUDA for the single GPU and MPI when multiple systems are available or absolutely necessary due to the size of the graphs under study.

The levels of performance achieved by the two codes are among the best reported in recent literature for single GPU and are unmatched on large scale systems, since most of other GPU-based solutions do not feature full MPI support. That choice limits their scaling to the number of GPUs that fit in a single computing node.

Our BFS and BC codes are available for download at [54] and [55], respectively.

For the future, we expect to work on multiple directions: for the BFS we would like to improve the heuristic to switch between the top-down and the bottom-up variant and to evaluate the advantages that technologies like GPUDirect Async can provide for reducing the overhead of the communication among GPUs. As to the BC, we are already adapting MGBC to the evaluation of *approximated* BC [56], [57] for extremely-large-scale graphs. Finally, we are going to explore the chance of using our BFS as a building block for the solution of other graphs problems.

Acknowledgments

The authors thank the Swiss Center for Scientific Computing (CSCS) for the access to their "Piz Daint" system.

References

[1] M. Bisson, M. Bernaschi, and E. Mastrostefano, "Parallel distributed breadth first search on the kepler architecture," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 7, pp. 2091–2102, Jul. 2016.

[2] M. Bernaschi, G. Carbone, and F. Vella, "Scalable betweenness centrality on multi-gpu systems," in *Proc. ACM Int. Conf. Comput. Frontiers*, 2016, pp. 29–36.

[3] E. Mastrostefano and M. Bernaschi, "Efficient breadth first search on multi-GPU systems," *J. Parallel Distrib. Comput.*, vol. 73, no. 9, pp. 1292–1305, Sep. 2013.

[4] M. Bernaschi, G. Carbone, E. Mastrostefano, and F. Vella, "Solutions to the st-connectivity problem using a GPU-based distributed BFS," *J. Parallel Distrib. Comput.*, vol. 76, pp. 145–153, 2015.

[5] Y. Jia, V. Lu, J. Hoberock, M. Garland, and J. C. Hart, "Edge vs. node parallelism for graph centrality metrics," *GPU Comput. Gems: Jade Ed.*, pp. 15–28, 2011.

[6] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, "A scalable distributed parallel breadth-first search algorithm on bluegene/l," in *Proc. ACM/IEEE Conf. Supercomput.*, 2005, Art. no. 25.

[7] S. Beamer, K. Asanović, and D. Patterson, "Direction-optimizing breadth-first search," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2012, pp. 12:1–12:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=2388996.2389013

[8] H. Liu and H. H. Huang, "Enterprise: Breadth-first graph traversal on GPUs," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2015, pp. 68:1–68:12. [Online]. Available: http://doi.acm.org/10.1145/2807591.2807594

[9] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *Proc. 21st ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2016, Art. no. 11.

[10] Y. You, D. Bader, and M. M. Dehnavi, "Designing a heuristic cross-architecture combination for breadth-first search," in *Proc. 43rd Int. Conf Parallel Process.*, 2014, pp. 70–79.

[11] U. Brandes, "A faster algorithm for betweenness centrality*," *J. Math. Sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[12] K. Madduri, D. Ediger, K. Jiang, D. A. Bader, and D. Chavarria-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," in *Proc. IEEE Int. Symp. Parallel Distrib. Process.*, 2009, pp. 1–8.

[13] A. E. Sariyüce, K. Kaya, E. Saule, and U. V. Çatalyürek, "Regularizing graph centrality computations," *J. Parallel Distrib. Comput.*, vol. 76, no. C, pp. 106–119, 2015.

[14] A. Buluç and J. R. Gilbert, "The combinatorial BLAS: Design, implementation, and applications," *Int. J. High Perform. Comput. Appl.*, vol. 25, pp. 496–509, 2011.

[15] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *Proc. Int. Symp. Comput. Archit.*, Jun. 2008, pp. 77–88.

[16] Y. Wang, et al., "Gunrock: GPU graph analytics," *CoRR*, vol. abs/1701.01170, 2017. [Online]. Available: http://arxiv.org/abs/1701.01170

[17] gunrock, gunrock: high-performance graph primitives for the GPU, 2012. [Online]. Available: http://gunrock.github.io/gunrock/doc/latest/index.html

[18] DIMACS, 10th dimacs implementation challenge - graph partitioning and graph clustering, 2012. [Online]. Available: http://www.cc.gatech.edu/dimacs10/

[19] enterprise, Enterprise: Breadth-first graph traversal on GPUs. sc'15, 2015. [Online]. Available: https://github.com/iHeartGraph/Enterprise

[20] M. Bisson, M. Bernaschi, and E. Mastrostefano, "Parallel distributed breadth first search on the kepler architecture," *IEEE Trans. Parallel Distrib. Syst.*, vol. 27, no. 7, pp. 2091–2102, Jul. 2016.

[21] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," Nov. 2015. [Online]. Available: http://snap.stanford.edu/data

[22] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-MAT: A recursive model for graph mining." in *Proc. SIAM Int. Conf. Data Mining*, 2004, vol. 4, pp. 442–446.

[23] B. Merry, "A performance comparison of sort and scan libraries for GPUs," *Parallel Process. Lett.*, vol. 25, no. 4, 2015, Art. no. 1550007.

[24] V. Agarwal, F. Petrini, D. Pasetto, and D. Bader, "Scalable Graph Exploration on Multicore Processors," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2010, pp. 1 –11. [Online]. Available: http://dx.doi.org/10.1109/SC.2010.46

[25] A. McLaughlin and D. A. Bader, "Revisiting edge and node parallelism for dynamic gpu graph analytics," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops*, 2014, pp. 1396–1406.

[26] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Accelerating CUDA graph algorithms at maximum warp," in *ACM SIGPLAN Notices*, vol. 46, no. 8, pp. 267–276, 2011.

[27] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient parallel graph exploration on multi-core cpu and GPU," in *Proc. Int. Conf. Parallel Archit. Compilation Tech.*, 2011, pp. 78–88.

[28] D. Merrill, M. Garland, and A. Grimshaw, "Scalable GPU graph traversal," in *Proc. 17th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2012, pp. 117–128.

[29] G. Malewicz, et al., "Pregel: A system for large-scale graph processing," in *Proc. ACM SIGMOD Int. Conf. Manage. Data*, 2010, pp. 135–146. [Online]. Available: http://doi.acm.org/10.1145/1807167.1807184

[30] J. Zhong and B. He, "Medusa: Simplified graph processing on GPUs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 25, no. 6, pp. 1543–1552, 2014.

[31] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan, "Cusha: vertex-centric graph processing on GPUs," in *Proc. 23rd Int. Symp. High-Perform. Parallel Distrib. Comput.*, 2014, pp. 239–252.

[32] X. Shi, et al., "Frog: Asynchronous graph processing on GPU with hybrid coloring model," Huazhong Univ. Sci. Technol., Wuhan, China, Tech. Rep. HUST-CGCL-TR-402, 2015.

[33] A. Buluc and K. Madduri, "Parallel breadth-first search on distributed memory systems," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2011, Art. no. 65.

[34] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. R. Choudhury, and Y. Sabharwal, "Breaking the speed and scalability barriers for graph exploration on distributed-memory machines," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2012, pp. 1–12.

[35] K. Ueno and T. Suzumura, "Parallel distributed breadth first search on GPU," in *Proc. 20th Int. Conf. High Perform. Comput.*, 2013, pp. 314–323.

[36] A. Buluç, S. Beamer, K. Madduri, K. Asanovic, and D. A. Patterson, "Distributed-memory breadth-first search on massive graphs," *CoRR*, vol. abs/1705.04590, 2017, http://arxiv.org/abs/1705.04590

[37] N. Satish, C. Kim, J. Chhugani, and P. Dubey, "Large-scale energy-efficient graph traversal: A path to efficient data-intensive supercomputing," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2012, pp. 14:1–14:11.

[38] M. Bernaschi, G. Carbone, E. Mastrostefano, M. Bisson, and M. Fatica, "Enhanced GPU-based distributed breadth first search," in *Proc. 12th ACM Int. Conf. Comput. Frontiers*, 2015, pp. 10:1–10:8.

[39] N. Edmonds, J. Willcock, T. Hoefler, and A. Lumsdaine, "Design of a large-scale hybrid-parallel graph library," presented at the *Int. Conf. High Perform. Comput., Student Res. Symp.*, Goa, India, 2010.

[40] F. Checconi and F. Petrini, "Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines," in *Proc. IEEE 28th Int. Parallel Distrib. Process. Symp.*, 2014, pp. 425–434. [Online]. Available: http://dx.doi.org/10.1109/IPDPS.2014.52

[41] K. Ueno, T. Suzumura, N. Maruyama, K. Fujisawa, and S. Matsuoka, "Extreme scale breadth-first search on supercomputers," in *Proc. IEEE Int. Conf. Big Data*, Dec. 2016, pp. 1040–1047.

[42] D. Prountzos and K. Pingali, "Betweenness centrality: Algorithms and implementations," *ACM SIGPLAN Notices*, vol. 48, no. 8, pp. 35–46, 2013.

[43] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and L. P. Chew, "Optimistic parallelism requires abstractions," *ACM SIGPLAN Notices*, vol. 42, no. 6, pp. 211–222, 2007.

[44] O. Green and D. A. Bader, "Faster betweenness centrality based on data structure experimentation," *Procedia Comput. Sci.*, vol. 18, pp. 399–408, 2013.

[45] G. Tan, D. Tu, and N. Sun, "A parallel algorithm for computing betweenness centrality," in *Proc. Int. Conf. Parallel Process.*, 2009, pp. 340–347.

[46] A. E. Sariyüce, K. Kaya, E. Saule, and U. V. Çatalyürek, "Betweenness centrality on GPUs and heterogeneous architectures," in *Proc. 6th Workshop Gen. Purpose Processor Using Graph. Process. Units*, 2013, pp. 76–85.

[47] A. McLaughlin and D. A. Bader, "Scalable and high performance betweenness centrality on the GPU," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, 2014, pp. 572–583.

[48] N. Edmonds, T. Hoefler, and A. Lumsdaine, "A space-efficient parallel algorithm for computing betweenness centrality in distributed memory," in *Proc. Int. Conf. High Perform. Comput.*, 2010, pp. 1–10.

[49] J. Kepner and J. Gilbert, *Graph Algorithms in the Language of Linear Algebra*, vol. 22. SIAM, 2011, doi: 10.1137/1.9780898719918.

[50] M. Bernaschi, G. Carbone, and F. Vella, "Betweenness centrality on multi-GPU systems," in *Proc. 5th Workshop Irregular Appl.: Archit. Algorithms*, 2015, pp. 12:1–12:4.

[51] E. Solomonik, M. Besta, F. Vella, and T. Hoefler, "Scaling betweenness centrality using communication-efficient sparse matrix multiplication," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, pp. 47:1–47:14, no. 47, 2017, doi: 10.1145/3126908.3126971.

[52] Y. Pan, Y. Wang, Y. Wu, C. Yang, and J. D. Owens, "Multi-GPU graph analytics," *CoRR*, vol. abs/1504.04804, 2015.

[53] Y. Wang, A. Davidson, Y. Pan, Y. Wu, A. Riffel, and J. D. Owens, "Gunrock: A high-performance graph processing library on the GPU," in *Proc. 20th ACM SIGPLAN Symp. Principles Practice Parallel Program.*, 2015, pp. 265–266.

[54] (Jan. 2018). [Online]. Available: https://gitlab.com/maurob/Bfs2D

[55] (Jan. 2018). [Online]. Available: https://bitbucket.org/fvella/mgbc

[56] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail, "Approximating betweenness centrality," in *Proc. Int. Workshop Algorithms Models-Web-Graph*, 2007, pp. 124–137.

[57] R. Geisberger, P. Sanders, and D. Schultes, "Better approximation of betweenness centrality," in *Proc. Meeting Algorithm Eng. Experiments*, 2008, pp. 90–100.

Massimo Bernaschi has been with IBM for 10 years working in high performance computing. Currently, he is with the National Research Council of Italy (CNR) as chief technology officer of the Institute for Computing Applications. He is also an adjunct professor of computer science with the “Sapienza” University in Rome.

Mauro Bisson received the degree and PhD degree in computer science from the Sapienza University of Rome, Italy. He is a developer technology engineer in the High Performance Computing and Benchmarks Team, Nvidia. Prior to joining Nvidia, he worked as a postdoc applying high performance computing to different fields like computational physics, cryptography, and graph processing. He has been a finalist in the ACM Gordon Bell prize multiple times and received an honorable mention in 2011.

Enrico Mastrostefano received the degree in physics and the PhD degree in computer science from the Sapienza University of Rome. His research interests include numerical simulation and high-performance computing.

Flavio Vella received the PhD degree in computer science from the University of Perugia. Currently, he is a researcher at Dividiti. He has also been an intern at the Nvidia Corporation and an academic guest at ETH Zurich. His interests include GPU computing, parallel algorithms, and computational logic.