DOI: 10.14529/jsfi210206

Distributed Graph Algorithms for Multiple Vector Engines of NEC SX-Aurora TSUBASA Systems

Ilya V. Afanasyev[1,2], *Vadim V. Voevodin*[1,2], *Kazuhiko Komatsu*[3], *Hiroaki Kobayashi*[3]

This paper describes the world-first attempt to develop distributed graph algorithm implementations, aimed for modern NEC SX-Aurora TSUBASA vector systems. Such systems are equipped with up to eight powerful vector engines, which are capable to significantly accelerate graph processing and simultaneously increase the scale of processed input graphs. This paper describes distributed implementations of three widely-used graph algorithms: Page Rank (PR), Bellman-Ford Single Source Shortest Paths (further referred as SSSP) and Hyperlink-Induced Topic Search (HITS), evaluating their performance and scalability on Aurora 8 system. In this paper we describe graph partitioning strategies, communication strategies, programming models and single-VE optimizations used in these implementations. The developed implementations achieve 40, 6.6 and 1.3 GTEPS performance on PR, SSSP and HITS algorithm on 8 vector engines, at the same time achieving up to 1.5x, 2x and 2.5x acceleration on 2, 4 and 8 vector engines of Aurora 8 systems. Finally, this paper describes an approach to incorporate distributed graph processing support into our previously developed Vector Graph Library (VGL) framework – a novel framework for graph analytics on NEC SX-Aurora TSUBASA architecture.

Keywords: vector computers, graph algorithms, graph framework, VGL, optimisation.

Introduction

Developing efficient graph algorithm implementations is an extremely important problem of modern computer science, since graphs are frequently used in various real-world applications, such as social network and web-graph analysis, navigation, and many others. Due to the fact that graph algorithms belong to the data-intensive class (since they typically load from memory a large amount of data, at the same time performing almost no floating point arithmetic), modern architectures with high-bandwidth memory potentially allow solving many graph problems significantly faster compared to modern multicore CPUs. Among other supercomputer architectures, NEC SX-Aurora TSUBASA vector processors [26, 36] are equipped with high-bandwidth memory, what makes them a very promising architecture for graph processing.

In our prior research, we have proposed a Vector Graph Library (VGL) [3, 4, 6] – a novel high-performance graph-processing framework, which is so far the only known graph processing library for the NEC SX-Aurora TSUBASA architecture. As we have previously demonstrated, VGL is capable to outperform both modern Intel multicore CPUs and NVIDIA GPUs on several graph algorithms.

However, at the current moment VGL supports graph processing only within a single vector engine of NEC SX-Aurora TSUBASA system. At the same time, modern NEC SX-Aurora TSUBASA systems, such as the A300-8 (Aurora8) [1], consist of up to 8 vector engines, connected to a single vector host. Such systems are similar to well-known multi-GPU systems, such as DGX or DGX-2, thus in the following paper, we will refer to them as "multi-VE" systems. Implementing graph processing only within a single vector engine has two important drawbacks. First, (1) memory of single vector engine is limited to 48 GB, thus large-scale graphs can not be

[1]Moscow Center of Fundamental and Applied Mathematics, Moscow, Russia
[2]Research Computing Center, Lomonosov Moscow State University, Moscow, Russia
[3]Tohoku University, Sendai, Miyagi, Japan

processed (without using out-of-core processing techniques, which have been shown to be rather slow on NVIDIA GPUs [15]). Second, (2) eight vector engines of the Aurora8 system working in parallel are capable of significantly accelerating many graph algorithms.

Similar to developing efficient vector graph algorithm implementations for a single vector engine of NEC SX-Aurora TSUBASA, approaches to developing efficient multi-VE implementations are not studied well enough at the moment of this writing. Due to a certain similarity of multi-VE to multi-GPU systems, some existing graph partitioning or inter-GPU communication methods can be used, but this requires a detailed verification due to the differences between vector engines and GPUs. Examples of such differences include using different graph storage formats and optimization techniques within single-GPU or single-VE, different properties of host-device interconnect, different programming models used for developing distributed multi-VE and multi-GPU implementations, and so on.

In this paper, we develop multi-VE implementations of three widely-used graph problems: Page Rank, Single Source Shortest Paths and Hyperlink-Induced Topic Search. We discuss in details which graph partitioning strategies, communication strategies, programming model, and single-VE optimizations have been applied to these implementations in order to achieve good scalability and performance. The performance and scalability of the developed implementations have been evaluated on Aurora 8 system, installed in Tohoku university. Finally, we discuss how programming multi-VE systems can be implemented in VGL framework[4].

1. NEC SX-Aurora TSUBASA Architecture

NEC has developed vector computing systems called SX series since SX-2 released in 1983 to SX-9 [33], and SX-ACE [13]. So far, many optimizations have been conducted on NEC SX series not only for the benchmark programs [11, 25] but also for various applications such as HPC simulation codes [12, 14, 23, 24] and graph algorithms [5, 7].

The latest vector computer NEC SX-Aurora TSUBASA [26, 36] with dedicated vector processors is the primary target architecture of graph algorithm implementations, proposed in this paper. NEC SX-Aurora TSUBASA has been developed according to the design concepts of vector supercomputer based on the long-term experiences and novel innovations to achieve higher sustained performance and higher usability. Different from the previous generations of the SX vector supercomputer series, the system architecture of SX-Aurora TSUBASA mainly consists of vector engines (VE), equipped with a vector processor and a vector host (VH) of an x86 node.

The VE is used as a primary processor for executing applications, while the VH is used as a secondary processor for managing the VE and executing a basic operating system (OS) functions that are offloaded from the VE.

1.1. Vector Engine

The VE has eight powerful vector cores. As each core provides 537.6 GFlop/s of single-precision performance at 1.40 GHz frequency, the peak performance of the VE reaches 4.3 TFlop/s.

Each SX-Aurora vector core consists of three components: scalar processing unit (SPU), vector processing unit (VPU), and memory subsystem. Most computations are performed by VPUs, while SPUs provide the functionality of a typical CPU. Since SX-Aurora is not just a

[4]VGL is available for free download at vgl.parallel.ru

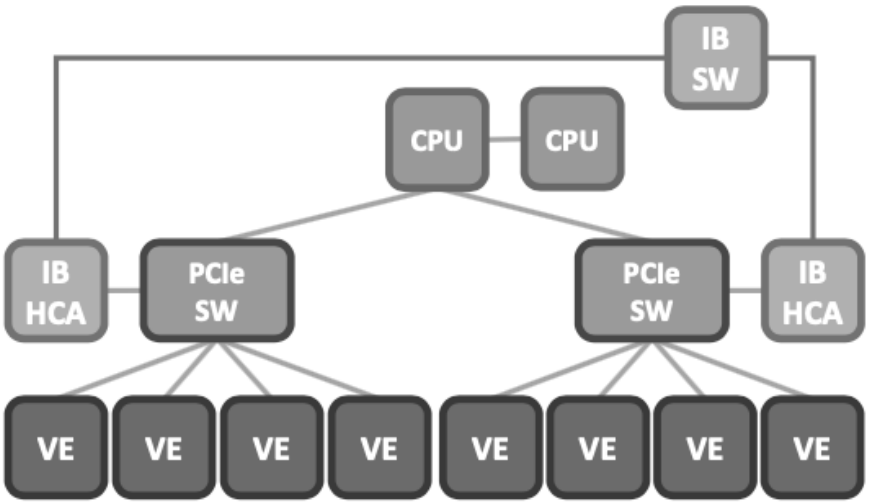

Figure 1. SX-Aurora TSUBASA A300-8

typical accelerator, but rather a self-sufficient processor, SPUs are designed to provide relatively high performance on scalar computations. VPU of each vector core has its own relatively simple instruction pipeline aimed at decoding and reordering vector instructions incoming from SPU. Decoded instructions are executed on vector-parallel pipelines (VPP). In order to store the results of intermediate calculations, each vector core is equipped with 64 vector registers with a total register capacity equal to 128 KB. Each register is designed to store a vector of 256 double-precision elements (DP). On the memory subsystem side, six HBM modules in the vector processor can deliver the 1.22 TB/s memory bandwidth with up to 48 GB total capacity.

1.2. SX-Aurora TSUBASA A300-8 System

Figure 1 is the SX-Aurora TSUBASA A300-8 model. One *VH node* consists of one Vector Host (*VH*) and eight Vector Engines (*VEs*). Four VEs are grouped into a *VE island*. Each VE island and one Infiniband EDR host channel adapter (HCA) is connected to PCI Express switch. Two PCI Express switch is connected into one of CPUs.

As shown in the figure, there are multiple hierarchies of communications. As the communication bandwidth is different among network hierarchies, it is necessary to consider the network hierarchies to exploit the potential of multiple VEs.

1.3. Programming Aurora Systems

Parallel programs for the NEC SX-Aurora Vector Engines are implemented using the OpenMP programming model, while vectorization is performed by the NEC compiler: a developer inserts compiler-specific directives, which help the compiler to perform automatic vectorization. When utilizing multiple Vector Engines of Aurora systems is required, MPI parallelization has to be implemented. At each vector engine an MPI process is started, while data transfers are implemented via MPI send, recv, gather, scatter, bcast and other functions.

2. State of the Art

In this section we will describe previously conducted research, related both to developing distributed graph algorithms and implementing graph algorithms on NEC SX-Aurora TSUBASA vector system.

2.1. Distributed Graph Algorithm Implementations

As already mentioned, NEX SX-Aurora TSUBASA systems equipped with multiple vector engines have multiple similarities to modern multi-GPU system. Developers of Gunrock [35] framework, which targets both single and multi-GPU, have recently published a comprehensive survey on approaches used for developing multi-GPU implementations [32].

There are multiple large scale distributed memory based graph processing systems for CPU clusters, such as GraphX [18], Pregel [28] or Giraph [20]. However, communication models and methods of those systems will very likely be under heavy pressures when local computation is much faster, as in the case when GPU or SX-Aurora vector engines are involved.

Regarding GPU, the most well-known multi-GPU frameworks are Enterprise [27], Medusa [37], Gunrock [35] and NVGRAPH [2]. These frameworks use important implementation techniques, referred through our paper: graph partitioning strategies (how graph vertices and edges are distributed among processors), such as 1D, 2D, Metis [21] partitioning, vertex duplication strategies (how graph remote graph vertices are stored), such as duplicate-1-hop, duplicate-all, and communication strategies (which information to transfer between GPUs), such as broadcast or selective-communicate. In addition, in these framework different programming techniques are used for inter-GPU communication, such as MPI, unified memory, peer-to-peer access.

2.2. VGL (Vector Graph Library) Framework

As previously mentioned in the introduction, we are primary developers of Vector Graph Library (VGL) – a novel graph processing framework, designed to operate on NEC SX-Aurora TSUBASA vector system. A detailed description of VGL is provided in [4]. VGL is designed to implement iterative graph algorithms, and thus uses Bulk synchronous parallel(BSP) [34] model. All graph algorithms in VGL are represented as a sequence of 4 computational abstractions: Advance, Compute, Reduce and Generate New Frontier abstractions (GNF). The Advance abstraction is responsible for processing graph edges, Compute and Reduce – graph vertices, while GNF allows to create working subsets of vertices, which can be processed by other abstractions. These abstractions operate over Graph, Frontier, VerticesArray and EdgesArray data-structures. VGL includes a large variety of optimizations required to operate efficiently within a single vector engine of NEC SX-Aurora TSUBASA, such as parallel workload balancing, using vector instructions with the maximum vector length, improving LLC usage, and many others. In addition, VGL provides an architecture-independent API, which allows it to support computations on different architectures, such as modern NVIDIA GPUs and multicore CPUs without having to change implemented algorithms. This is achieved by object oriented programming, which allows to develop different implementations of each computational abstraction, however, providing identical interfaces for each target architecture [3].

2.3. VGL Graph Storage Format

A central point of VGL framework is an optimized graph storage format, called VectCSR. This format is described in details in [4]. VectCSR is based on a combination of CSR (Compressed Sparse Row) and Sell-C-Sigma [17] formats. In the following paragraph, we will provide a brief description of VectCSR format, since understanding it is frequently required in the following subsections of the paper.

Vect CSR graph storage format scheme is provided in Fig. 3. Its main idea is based on splitting graph vertices into 3 groups based on their outgoing (or incoming) degrees. Vertices with "large degree" ($> 256 * 8$) are processed using all cores of vector engine, each vertex with "medium degree" – using a single vector core, while a group of 256 "small degree" vertices is processed collectively by a single vector core. In addition, edges of "small-degree" vertices are stored in memory in 2 different representations: CSR, which allows to process only a few vertices in sparse algorithms (for example final iterations of BFS), and representation similar to Sell-C-Sigma format, where all edges are reordered in a way shown in Fig. 3. In order to simplify splitting vertices in such groups, as well as to improve memory access pattern for power-law graphs, graph vertices are preliminary sorted based on their degree, and renumbered afterwards.

According to our experience [4], using VectCSR Graph Storage format is mandatory for achieving high performance of graph processing on a single vector engine. This is mostly related to the fact that CSR format does not allow to process graph vertices with a degree lower than vector length efficiently. At the same time VectCSR format provides enough flexibility to allow processing specific subsets of graph vertices and their adjacent edges, which is required in many sparse graph algorithms, such as BFS. Our experiments during VGL development demonstrated that using VectCSR format provides approximately 4–5 times acceleration compared to using traditional CSR format on various synthetic and real-world graphs. This forces us to investigate graph partitioning strategies taking into account the requirement of storing graph in VectCSR-like format within a single vector engine.

3. Evaluating MPI Benchmarks on Aurora8 System

In the beginning of our research, we wanted to estimate transfer speed and scalability of two communication patterns frequently used in graph processing: send/receive point-to-point and all_gather.

3.1. MPI Ping-Pong

MPI ping pong benchmark can be used to evaluate transfer speed of send/receive point-to-point communication. "Ping-pong" benchmark is based on performing a sequence of MPI_Send and MPI_Recv operation between two MPI processes. This communication pattern is used in graph algorithm when MPI processes are doing cycle exchange in order to update some vertices arrays, as will be described further in the paper. We executed "ping-pong" benchmark on MPI processes, attached to different Vector Engines of Aurora8 system (Fig. 2). As shown in Fig. 2, the transfer bandwidth slightly decreases in the case when communicating processes are attached to different switches of aurora8 system (e.g., vector engines 0 and 7). To achieve close to theoretical peak bandwidth values, sending messages of ≥ 16 MB size is needed. Finally, point-to-point communication bandwidth is relatively low due to the fact that different vector

engines are communicating through PCI express, one-directional bandwidth of which does not exceed 12 GB/s. On the contrary, multi-GPU systems equipped with NVLINK 3.0 and A100 GPU are capable of transferring data with 300 GB/s one-directional bandwidth. This means that graph partitioning and communication strategies have to be chosen much more carefully for vector engines, since MPI transfers between different Vector Engines can quickly become a significant bottleneck.

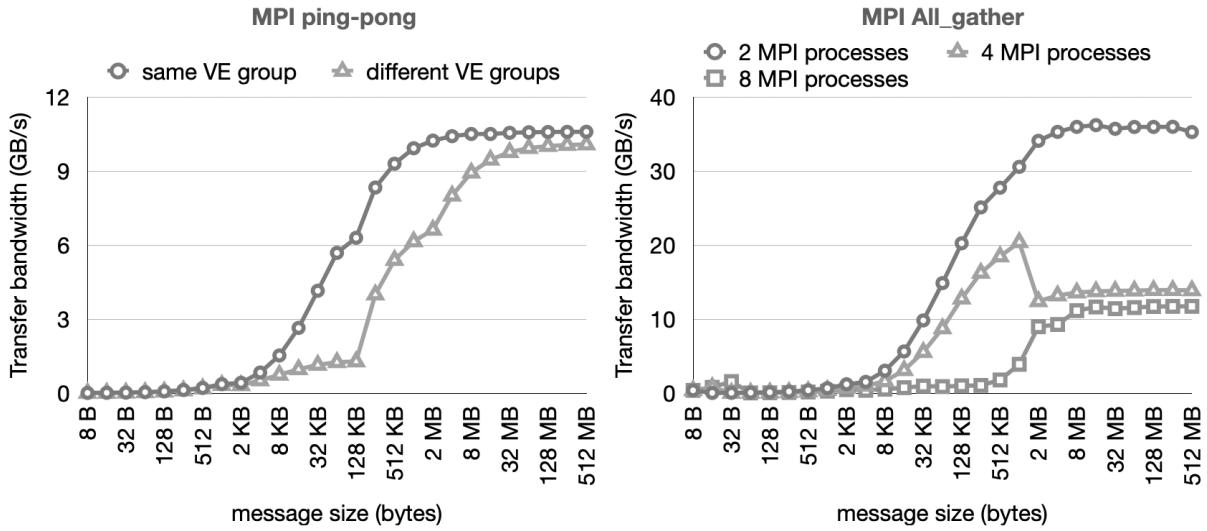

Figure 2. The transfer bandwidth of MPI ping-pong (left) and all_gather (right) benchmarks on Aurora8 system

3.2. MPI_Allgather

Next we evaluated transfer bandwidth of MPI_Allgather operation, which together with MPI_Allgatherv will be frequently used in our implementations. Transfer bandwidth of MPI_Allgather is calculated as $|N|/time$, where $|N|$ is the size of array, which is distributed among $|P|$ MPI processes. As shown in Fig. 2, MPI_Allgather communication between two adjacent vector engines (0/1, 1/2, etc.) can be up to two times faster, compared to four and eight. This is explained by the (1) increase of communication rate (amount of data transferred between all processes) and (2) that while communication of 2 and 4 vector engines is handled by one PCIe SW, but two PCIe SW are used for 8 vector engine communication. In the context of developing graph algorithms this means that the developed implementations will demonstrate the best scaling among two vector engines when MPI_Allgather is used.

4. Implementing Multi-VE Graph Algorithms

4.1. Deciding on Graph Partitioning Strategy

The first important thing to take into an account is how graph vertices and edges need to be partitioned among MPI processes. When selecting graph partitioning strategy, the following factors need to be considered:

1. each MPI process should process approximately equal amount of graph edges;
2. edge processing rate should be approximately equal among different MPI processes;

3. the amount of communications (number of vertices, information about which is required to be sent to other processors) should be minimized;
4. graph processing inside a single vector engine should include optimizations, implemented in VGL (primary VectCSR format should be supported);
5. graph partitioning process should not be very complex and should be vectorizable in order to allow graph partitioning right on vector engines (to avoid exchanging data with vector host).

Thus, we considered three possible graph partitioning strategies between different MPI processes.

Successive distribution of VectCSR edges between MPI processes

First, let us denote the total amount of graph edges as $|E|$, total amount of graph vertices as $|V|$, and total amount of MPI process (and thus vector engines, because each MPI process is bound to a separate vector engine) as $|P|$. The most basic partitioning strategy, which can be applied to VectCSR format is illustrated in Fig. 3 (left), where each MPI process stores a successive region of $|E|/|P|$ edges. This approach has the following **advantages**:

1. such partitioning allows efficient graph processing inside single vector engine, since no reordering of vertices data or strided memory accesses to vertices and edges arrays is required (unlike in other approaches, which we will discuss further in the section);
2. each MPI process storing approximately equal amount of edges.

However, this approach also has several crucial **disadvantages**:

1. as shown in our previous paper [4], edge processing rate among different groups of vertices in VectCSR format ("large-degree", "medium-degree", "small-degree") can be different for some algorithms and input graphs. For example, on RMAT [10] graphs and shortest paths algorithm "small-degree" vertices are processed with a rate of 518 GB/s sustained bandwidth rate, while group of "medium-degree" vertices – with 350 GB/s sustained bandwidth rate. This causes uneven load balancing between different vector engines, since each engine processes equal amount of $|E|/|P|$ edges, but with different processing rate.
2. vector extension, required for fast processing of vertices with low-degree, is stored together with CSR representation. Thus several MPI processes (MPI Proc 3 in Fig. 3), which store information about vertices with low degree, are forced to store approximately twice the amount of edges, which leads to a significant difference in memory consumption between different processes.
3. finally, for many scale-free graphs with uneven distribution of vertex degrees, each MPI process does not necessary store $|V|/|P|$ vertices. In the case when communication between processes is based on sending "local" vertices, communication time will be different for each MPI process resulting in a significant imbalance of time spent on communication.

Distribution of VectCSR edges between MPI processes inside separate vertex groups Disadvantages (1) and (2) of previously discussed partitioning strategy can be relatively easily solved by doing partitioning inside each vertex group ("small-degree", "medium-degree", etc.) instead of the whole graph, as shown in Fig. 3 (right). According to our experiments, edge processing rates are roughly equal inside different parts of vector groups, while vector extension is distributed among all working processes, instead of only last ones. However, disadvantage (3) still exists, since low rank processes store information about lower number of vertices in scale-free graphs. In order to solve this issue to some extent, we reversed distribution of vertices

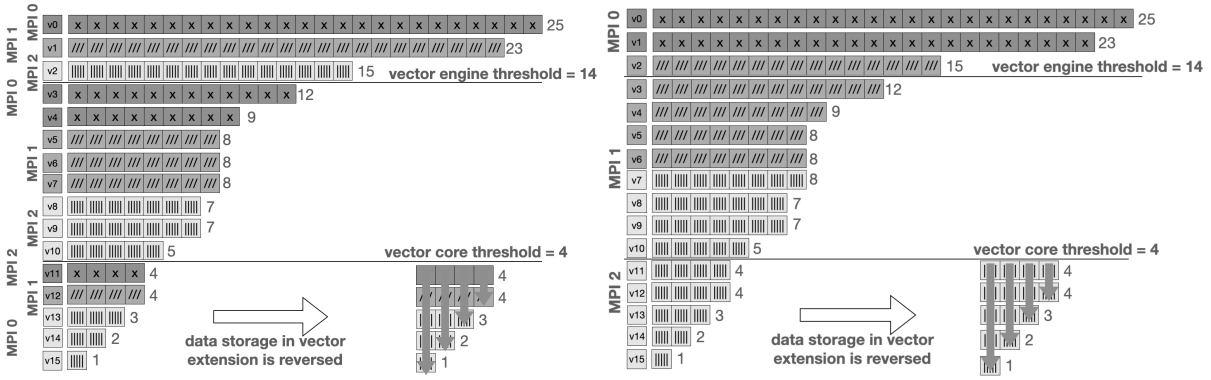

Figure 3. Two strategies of distributing VectCSR graph between MPI processes: spliting VectCSR edges between MPI processes successively and splitting graph edges inside each vector group. Three MPI processes used in this example: MPI process 0 stores graph edges are marked as "x", MPI process 1 – as "///", MPI process 2 – as "|||"

inside vector core and collective groups, as shown in Fig. 3 (right): low rank processes store high-degree vertices of vector core group, and small-degree vertices of collective group.

Specific distribution of graph vertices between processes Despite previously discussed graph partitioning strategies lack significant disadvantages, strategies which allow better load-balancing and lower amount of communications between MPI processes exist. These strategies are based on distributing graph vertices and edges between different processor according to some vertex partitioning function. However, a more complex graph storage format should be used in order to incorporate VectCSR format into such approaches, due to each vector engine storing non-consecutive vertices. This format (illustrated in Fig. 4) will be further referred to as ShardedCSR format. Vertices in ShardedCSR graph are distributed between different shards (segments) using a specific partitioning function, some of which will be described in the further paragraphs. In our implementation, each vertex belongs only to a single shard, however, this can be relatively easily changed.

Vertex partitioning function is a crucial factor, which defines quality of load-balancing and the amount of communication in ShardedCSR graph. A particularly good strategy, used in Gunrock, is random partitioning of vertices between GPUs (vector engines in our case). Each vertex is assigned with an equal probability to a particular GPU, together with all its neighbouring edges. If the size of processed graph is large, this approach results in $|E|/|P|$ edges and $|V|/|P|$ vertices being stored by each VE.

Another well-known approach is using graph partitioning systems, which are designed to minimize the amount of communications between different MPI processes, such as Metis [21]. Metis provides multilevel recursive-bisection, multilevel k-way, and multi-constraint partitioning schemes. Unfortunately, Metis can not be executed on vector engines, and thus requires doing preliminary graph partitioning on vector host. According to our experiments this approach is rather disadvantageous, since (1) graph partitioning time on CPU is huge and (2) it is accompanied by a large transfer time graph VH to VE through PCIe, which results in very significant pre-processing overheads, in many situations larger than distributed graph processing time. At the same time, previously mentioned methods allow vectorized graph partitioning and preprocessing directly on vector engines, which allows them to significantly outperform Metis-based approach.

Conclusions Thus, further in the paper we will use two **most promising** graph partitioning strategies: (1) Distribution of VectCSR edges between vector engines inside separate vertex groups and (2) random distribution of graph vertices between vector engines.

4.2. Deciding on Communication Strategy

Communication strategy is another important trait of distributed graph algorithm implementations. All the discussed graph partitioning strategies are based on splitting graph edges between MPI processes to make sure that all the required for the computations edges are stored locally (on the process itself). Thus, MPI processes need to exchange **only information about vertices** (for example current distances in shortest paths, rank values in page rank, etc). Information about graph vertices can be exchanged in several different ways, depending on the properties of algorithm and graph partitioning schemes. The comparative characteristics of multiple communication strategies, such as additional space required for communication buffers, communication cost (the amount of data transferred between all processes) and pre-/post-processing complexity are provided in Tab. 1. Pre-/post-processing complexity is the total number of operations, required to prepare data for MPI communication (for example, copy information about scattered vertices into exchange buffers), and to receive it afterwards. The communication cost provided in Tab. 1 takes into account only the amount of data exchanged, however, the actual communication time additionally depends on the performance of MPI communication functions, shown in Fig. 2.

Table 1. The comparative characteristics of communication strategies. $|V|$ – number of vertices in graph, $|N|$ – number of MPI processes, $|K_i|$ – number of updated vertices on each MPI process

Strategy	**Additional space for communication buffers**	**Communication cost**	**Pre-/post-processing complexity**
(1)all-to-all (bcast based)	$\|V\| * \|N\|$	$\|V\| * \|N\|^2$	$\|V\| * \|N\|$
(2)all-to-all (cycle exchange based)	$\|V\|$	$\|V\| * \|N\| * log\|N\|$	$\|V\| * log\|N\|$
(3)all-to-all recently changed only	$\|V\|$	$(\sum_{i=1}^{\|N\|} K_i) * \|N\| * log\|N\|$	$(\sum_{i=1}^{\|N\|} K_i) * log\|N\|$
(4)all_gather	0	$\|V\| * \|N\|$	0
(5)all_gather recently changed only	$\|V\|$	$(\sum_{i=1}^{\|N\|} K_i) * \|N\|$	$K_i + \sum_{i=1}^{\|N\|} K_i$

All-to-all This communication scheme involves each MPI process broadcasting vertices array of size $|V|$ to all other processes. This can be achieved by using 2 communication algorithms. (1) Each MPI process broadcasts information about $|V|$ graph vertices to all other processes, while receiving information about $|V| * |N|$ vertices. After communication each process updates its private vertices arrays using provided in the algorithm update criteria, such as computing minimal distances to each vertex or calculating a sum of ranks for each vertex, obtained on different processes. In Tab. 1 this strategy is referred to as “all-to-all bcast based”. Another

possible strategy (2) requires each MPI process to send information about $|V|$ graph vertices to $rank+1$ process and simultaneously receive information about $|V|$ vertices from $rank-1$ process. After communication, each process updates local vertices arrays using update criteria and the received information. After that, newly calculated information is sent to $rank+2$ process, and so on using $log|N|$ steps. In Tab. 1 this strategy is referred to as **"all-to-all cycle exchange based"**. This approach requires significantly less additional space and has lower communication cost compared to "all-to-all bcast based" strategy (1).

All-to-all recently changed only This strategy is based on slightly modified all-to-all strategy. However, information only about $|K|$ vertices, values of which have been changed during the last algorithm iteration (for example only recently updated distances or ranks) is sent to other processes. This reduces the amount of transferred data depending on the algorithm properties, however, at the cost of increased pre-processing and post-processing, since each process is now required to generate lists of recently updated vertices before communication (using copy_if or parallel prefix sum algorithm), and afterwards to place received data to the correct places of local vertices arrays using scatter operation. Similarly to "all-to-all" strategy, cycle exchange communication pattern can be used instead of broadcast here.

All_gather Two previously discussed strategies can be used both for pull- and push-based [9] algorithms. In pull-based algorithms each MPI process updates only its locally stored vertices, while for push-based algorithm any graph vertices can be updated. Since pull-based algorithms update only their local vertices during computations, it is possible to use communication strategy based on MPI_all_gather operation, when each process broadcasts only its local vertices.

All_gather recently changed only This strategy is aimed to further reduce communication cost of all_gather strategy, similar to "all-to-all recently changed only". Each process generates a list of recently updated local vertices, and then uses all_gatherv operation to communicate lists of potentially smaller size, but, similarly, at the cost of additional pre- and post-processing.

Conclusions Table 1 demonstrates that each strategy can potentially be useful for different graph algorithms, due different strategies having different trade-offs: larger communicating cost or pre-/post-processing complexity, etc. In addition, all these strategies can be implemented on vector engines, since they do not require complex data structure and support all the required optimizations, applied withing a single VE. In the following section we implemented all these strategies for single source shortest paths algorithm, comparing execution time, performance and scalability of each approach, and afterwards selecting the most fitting approach for NEC SX-Aurora TSUBASA architecture.

4.3. Bellman-Ford Shortest Paths Algorithm

The single-source shortest paths problem involves finding paths between a given source vertex and all other graph vertices, such that all weights on the path between source and destination vertices are minimized. Multiple parallel shortest paths algorithms exist, including delta-stepping [29] and Bellman-Ford [16], the latter being implemented in VGL, and thus being a subject of our research. Two different variations of Bellman-Ford algorithm exist: push-based and pull-based. Pull-based variation updates distance to each vertex based on the distances to vertices, connected via incoming edges to the processed vertex. Push-based algorithm propagates distance of current vertex to its neighbouring vertices via outgoing edges. Both these variations are suitable for implementation within a single vector engine [4].

We implemented the following variations of shortest paths algorithms based on vectCSR partitioning inside separate vector groups:

1. push-based algorithm using “all-to-all” communication strategy;
2. push-based algorithm using “all-to-all recently-changed only” communication strategy;
3. pull-based algorithm using “all-to-all recently-changed only” communication strategy;
4. pull-based algorithm using “all-gather” communication strategy;
5. pull-based algorithm using “all-to-all recently-changed only” communication strategy.

In addition, we implemented two variations of algorithm based on ShardedCSR graph storage format and random vertex partitioning:

1. pull-based algorithm using “all-gather” communication strategy;
2. push-based algorithm using “all-to-all recently-changed only” communication strategy.

Scalability of these implementations is evaluated in Fig. 4. Here (as well as in the following section) synthetic RMAT [10] of scale 25 with edge factor 32 are used as input data. Thus such graph contains 33 million vertices and 1 billion edges. RMAT graphs are scale-free, which means that they have power law distribution of vertex degrees. Thus, they successfully model social networks and web graphs, which are used in various application fields. As could be expected, scalability of all-to-all implementations is very limited due to a very large amount of transferred data coupled with low bandwidth of PCIe interconnect.

The highest performance and best scalability has been achieved when using VectCSR partitioning inside separate vertices groups together with “push all to all recently changed” and “pull all gather” communication policies. However, scalability of both these implementations is still far away from the linear: only 2 and 2.6 times acceleration is achieved when using 8 vector engines. In order to further investigate these issues, we have collected profiling data, which is provided in Tab. 2 and Tab. 3. Table 2 provides information about main computational components of the developed implementations, such as execution time spent inside Advance (which processes graph edges) and Compute (which processes graph vertices) VGL abstractions, MPI pre-process and post-process functions (when sent data is prepared and received data is analysed), and MPI communication time (of MPI_send, MPI_recv, MPI_allgather, MPI_bcast functions). Table 2 demonstrates that while time, spent on processing of graph edges (Advance) scales nearly linearly, the primary reason of limited scalability of both implementations is the increase of MPI communications and MPI pre-/post-processing time. While on single VE no time is required on these activities, 16 % of program execution time is spent on communication and pre-/post-processing on 2 VE, 42 % on 4 VE and 61 % on 8 VE for push-based algorithm. Similar situation can be observed on pull-based algorithm. As demonstrated in Tab. 3, time spent on communications is roughly equal between different MPI processes.

On the final note, we would like to discuss reasons standing behind poor scalability of implementations, based on ShardedCSR format. Despite the fact it provides more equal distribution of graph edges and vertices between different processes, this format requires reordering of vertices arrays of size $|V|$, after obtaining information from other processes, since vertices inside each sharded can be sorted differently to support VectCSR format inside each shard. Despite the fact that this reordering has $O|V|$ complexity, its efficiency is lower compared to edge processing in advance, which has $O(|E|/|N|)|$ complexity. With the increase of number of MPI processes used ($|N|$), and taking into the account that for many real-world graphs $|E| = C * |V|$, with $|C|$ being a constant in 8–64 range, such reordering has a comparable time with Advance execution time, and the scalability of ShardedCSR implementaion being limited by Amdahl’s law [19]. A

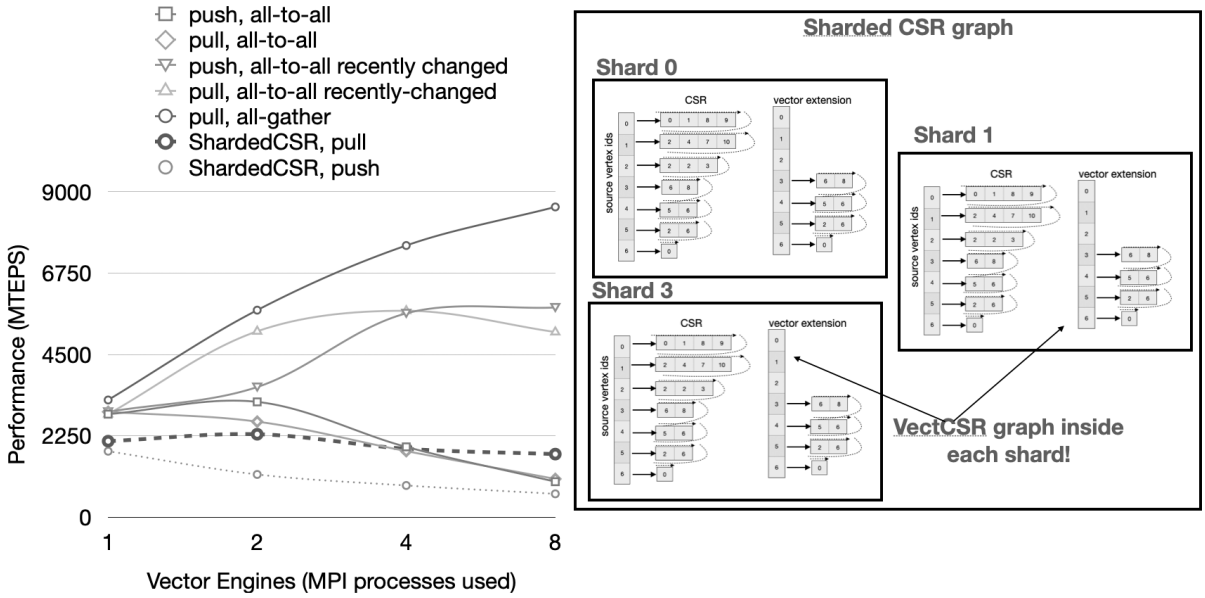

Figure 4. Scalabilty of the shortest paths algorithms (left), ShardedCSR graph storage format scheme (right). ShardedCSR graph consists of different shards (0, 1, 2 in this example). Each shard is a subgraph in VectCSR format, demonstrated in details in Fig. 3, which includes CSR and vector extension

Table 2. Main computational components of shortest paths implementations on RMAT graph of scale 25, multiple algorithm iterations

Activity	**SSSP, push, 1 VE**	**SSSP, push, 2 VE**	**SSSP, push, 4 VE**	**SSSP, push, 8 VE**	**SSSP, pull, 1 VE**	**SSSP, pull, 2 VE**	**SSSP, pull, 4 VE**	**SSSP, pull, 8 VE**
Advance	3540 s	1999 s	1122 s	602 s	3795 s	1467 s	737 s	387 s
Compute	23 s	31 s	29 s	32 s	28 s	30 s	32 s	32 s
MPI communication	-	81 s	224 s	139 s	-	294 s	407 s	774 s
MPI preprocess and postprocess	-	324 s	598 s	834 s	-	40 s	17 s	21 s

possible solution to this problem is using CSR format instead of VectCSR inside each shard, which removes the requirement of reordering vertices after each communication, however, these increase Advance time in 3–4 times for various graph algorithms, which is an unacceptable trade off.

4.4. Page Rank Algorithm

The page rank [31] algorithm assigns a numerical weighting to each element of a hyperlinked set of documents, (for example, web-graph), with the purpose of quantifying its relative importance within the set. Similarly to shortest paths, page rank algorithm has pull-based or push-based variations. Push-based algorithm requires using atomicAdd operations (since 2 vertices processed in parallel can possibly try to update the same adjacent vertex). Despite vector

Table 3. Profiling of shortest paths implementations using Ftrace tool

Algorithm and MPI rank	ELAPSED time	COMM TIME	COMM TIME /ELAPSED time	AVER.LEN
sssp pull, rank 0	2.229 s	1.392 s	62 %	4 MB
sssp pull, rank 1	2.227 s	1.300 s	58 %	4.1 MB
sssp pull, rank 2	2.227 s	1.047 s	47 %	4.5 M
sssp pull, rank 3	2.226 s	0.423 s	19 %	7 MB
sssp push, rank 0	2.039 s	0.340 s	16 %	3.1 MB
sssp push, rank 1	2.037 s	0.373 s	12 %	3.1 MB
sssp push, rank 2	2.037 s	0.333 s	11 %	3.0 MB
sssp push, rank 3	2.036 s	0.248 s	6 %	3.0 MB

engines having support of atomic operations, code including them can not be vectorized, thus pull-based variation must be used on NEC SX-Aurora TSUBASA.

Based on the research provided for shortest paths algorithm, for page rank algorithm we implemented vectCSR partitioning inside separate vector groups and all_gather communication model. Scalability of the developed implementation is provided in Section 5.

4.5. HITS

Hyperlink-Induced Topic Search (HITS) is also a link analysis algorithm that rates Web pages [22], however, quite differently compared to page rank. The main idea of HITS algorithm is based on each graph vertex having authority and hub scores. Both scores are consequently updated on each iteration; authority score of each node is updated based on incoming edges, while hub score – based on incoming. This means that when using VectCSR graph storage format, two reorderings of vertices arrays are required on each iteration, when traversal direction is changed. According to the profiling data the reordering process on large graphs (vertices arrays of which do not fit into LLC cache) take up to 45 % of program execution time, while the remaining 55 % are spent on Advance (processing edges), Compute and Reduce (communications are excluded). At the same time increasing the number of vector engines reduces Advance time linearly, but does not reduce reordering time at all, since each process is required to reorder $|V|$ vertices no matter how many vector engines are used. Thus scalability of such implementation is limited due to Amdahl's law, similarly to shortest paths implementation when using shardedCSR graph partitioning. Scalability of the developed HITS implementation is provided in Fig. 5 and Tab. 4.

The provided analysis allows to make a conclusion that VectCSR-based format is not suitable for distributed graph processing when algorithm frequently switches traversal direction, such as HITS or direction-optimizing BFS [8].

4.6. Implementing Distributed Graph Processing Support in VGL

The provided research allowed us to include support of distributed graph processing inside VGL[5]. This has been achieved by implementing a special method exchange_vertices_array, which is aimed to update vertices on each vector engine according to remote data and is called after each graph algorithm iteration. In addition, we slightly modified VectCSR and ShardedCSR graph storage formats in order to support distributed graph storage, as well as inner representation of Advance abstraction. However, since all computational and data abstractions have their interfaces unchanged, many implemented in VGL algorithms can be turned into distributed versions by simply adding exchange_vertices_array calls into the correct places.

5. Evaluating Scalability of All Developed Implementations

Figure 5 and Tab. 4 demonstrate scalability of the developed implementations of page rank, HITS and shortest paths (pull and push) algorithms. The performance comparison of VGL-based implementations working on a single vector engine with their GPU and Intel CPU counterparts can be found in [4] or at VGL website[6] in the "performance" section. In general, for SSSP and PR algorithms VGL-based implementations are up to 14 times faster compared to Ligra, Galois and GAPBS multicore CPU frameworks and libraries, and up to 3 times faster compared to Gunrock and NVGRAPH implementations for various synthetic and real-world graphs [4]. Thus, in this section we will only evaluate MPI scalability of the developed implementations.

During these experiments we used 1, 2, 4 and 8 vector engines of Aurora 8 systems. When using 2 and 4 vector engines, MPI processes were bound to the adjacent vector engines in the same PCIe SW group. The main performance characteristics used is TEPS (Traversed Edges Per Second) [30], equal to the amount of graph edges processed divided by algorithm execution time. For page rank and HITS algorithms the performance is calculated for a single iteration.

Table 4. Scalabilty of all the developed distributed graph algorithm implementations

Algorithm	1 VE	2 VE	4 VE	8 VE
page rank	1	1.53	2.0	2.38
sssp (push)	1	1.41	1.75	1.82
sssp (pull)	1	1.74	1.97	2.34
hits	1	0.89	1.00	1.07

Finally, we can compare scalability of our implementations with multi-GPU scalability of Gunrock [35]. According to the research [32] on SSSP problem Gunrock achieves 1.5 and 2.92 times acceleration when using 2 GPUs and 6 GPUs, while on PR Gunrock achieves 2.8 and 4.1 times acceleration on 2 and 6 GPU. Comparing these values to Tab. 4 indicates that VGL achieves lower scalability, which is explained by differences in bandwidth of interconnect used in modern GPUs (NVLINK, 80–300 GB/s) and NEC SX-Aurora TSUBASA (PCIe, 12 GB/s).

[5]Distributed graph processing is currently available in VGL on NEC-MPI git branch.
[6]https://vgl.parallel.ru/performance.html

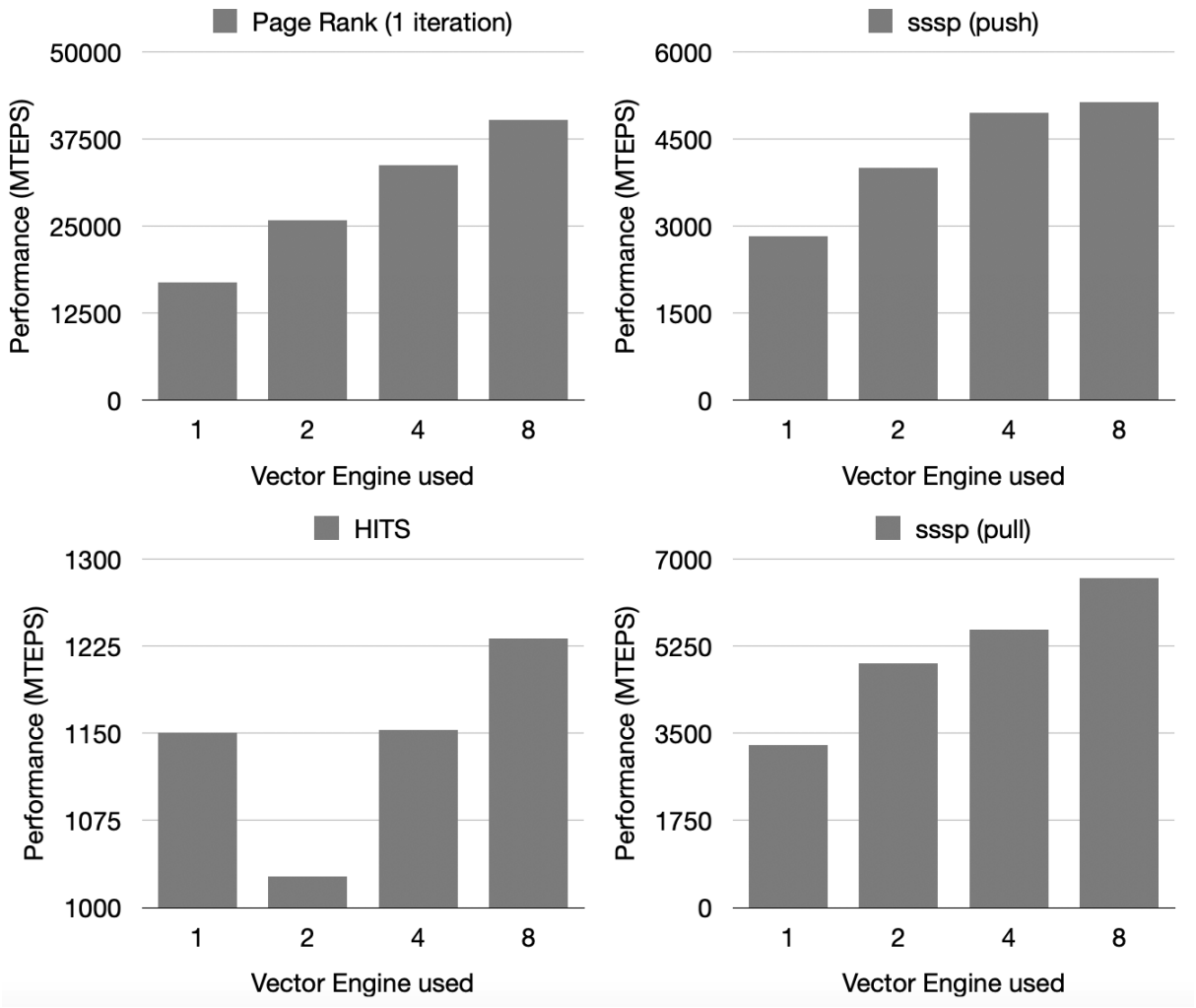

Figure 5. Scalabilty of the developed distributed implementations of page rank, HITS and shortest paths (pull and push) graph algorithms

We hope that future generations of NEC SX-Aurora TSUBASA architecture will be based on higher bandwidth interconnect, which can greatly improve scalability of our implementations.

Conclusion and Future Plans

In this paper we have proposed world first distributed graph algorithms for modern NEC SX-Aurora TSUBASA systems. In this paper we have discussed multiple attributes of these implementations: graph storage format, graph partitioning schemes, communication strategies, optimizations within single vector engine. Our experiments demonstrated that partitioning of VectCSR graph storage format inside separate vertex groups coupled with all_gather and all-to-all recently changed only strategies communication provide the best scalability. Our implementations achieve 40, 6.6 and 1.3 GTEPS performance on PR, SSSP and HITS algorithm on 8 vector engines, at the same time achieving up to 1.5x, 2x and 2.5x acceleration on 2, 4 and 8 vector engines of Aurora 8 systems. In addition, the developed distributed implementations allowed us to process 8 times larger graphs using Aurora 8 system at the same time obtaining reasonable (up to 2.6 times acceleration), which is crucial in many real-world applications.

Finally, using the example of HITs algorithm we have demonstrated that algorithms, which require switching between push and pull traversal on each iteration do not scale well with the proposed approaches on Aurora systems. Solving this problem is an important direction of our future work. Our other future plans include developing distributed versions of graph algorithms, which require working with sparse frontiers of active vertices, such as breadth-first search.

Acknowledgements

The research is carried out using the equipment of the shared research facilities of HPC computing resources at Lomonosov Moscow State University and the computational resources of Cyberscience Center at Tohoku University.

The reported study presented in all sections excluding 5 was funded by RFBR and JSPS according to the research project No. 21-57-50002 and Grant number JPJSBP120214801. The work presented in Section 5 is supported by Russian Ministry of Science and Higher Education, agreement No. 075-15-2019-1621.

References

1. NEC SX-Aurora TSUBASA A300-8. https://www.nec.com/en/global/solutions/hpc/sx/A300-8.html, accessed: 2021-04-06

2. NVGRAPH. https://developer.nvidia.com/nvgraph, accessed: 2021-06-28

3. Afanasyev, I.V.: Developing an architecture-independent graph framework for modern vector processors and NVIDIA GPUs. Supercomputing Frontiers and Innovations 7(4), 49–61 (2021). https://doi.org/10.14529/jsfi200404

4. Afanasyev, I.V., Voevodin, V.V., Komatsu, K., Kobayashi, H.: VGL: a high-performance graph processing framework for the NEC SX-Aurora TSUBASA vector architecture. The Journal of Supercomputing 77(8), 8694–8715 (2021). https://doi.org/10.1007/s11227-020-03564-9

5. Afanasyev, I.V., Voevodin, V.V., Komatsu, K., Kobayashi, H., *et al.*: Developing efficient implementations of Bellman–Ford and Forward-Backward graph algorithms for NEC SX-ACE. Supercomputing Frontiers and Innovations 5(3), 65–69 (2018). https://doi.org/10.14529/jsfi180311

6. Afanasyev, I.V., Voevodin, V.V., Komatsu, K., Kobayashi, H., *et al.*: Analysis of relationship between SIMD-processing features used in NVIDIA GPUs and NEC SX-Aurora TSUBASA vector processors. In: International Conference on Parallel Computing Technologies. Lecture Notes in Computer Science, vol. 11657, pp. 125–139. Springer (2019). https://doi.org/10.1007/978-3-030-25636-4_10

7. Afanasyev, I.V., Voevodin, V.V., Komatsu, K., Kobayashi, H., *et al.*: Developing efficient implementations of shortest paths and page rank algorithms for NEC SX-Aurora TSUBASA

architecture. Lobachevskii Journal of Mathematics 40(11), 1753–1762 (2019). https://doi.org/10.1134/S1995080219110039

8. Beamer, S., Asanović, K., Patterson, D.: Direction-optimizing breadth-first search. Scientific Programming 21(3-4), 137–148 (2013). https://doi.org/10.1109/SC.2012.50

9. Besta, M., Podstawski, M., Groner, L., *et al.*: To push or to pull: On reducing communication and synchronization in graph computations. In: Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing. pp. 93–104. ACM (2017). https://doi.org/10.1145/3078597.3078616

10. Chakrabarti, D., Zhan, Y., Faloutsos, C.: R-MAT: A recursive model for graph mining. In: Proceedings of the 2004 SIAM International Conference on Data Mining. pp. 442–446. SIAM (2004). https://doi.org/10.1137/1.9781611972740.43

11. Egawa, R., Komatsu, K., Isobe, Y., *et al.*: Performance and power analysis of SX-ACE using HP-X benchmark programs. In: 2017 IEEE International Conference on Cluster Computing (CLUSTER). pp. 693–700. IEEE Computer Society (2017). https://doi.org/10.1109/CLUSTER.2017.65

12. Egawa, R., Komatsu, K., Kobayashi, H.: Designing an HPC refactoring catalog toward the exa-scale computing era. In: Resch, M.M., Bez, W., Focht, E., Kobayashi, H., Patel, N. (eds.) Sustained Simulation Performance 2014. pp. 91–98. Springer (2015). https://doi.org/10.1007/978-3-319-10626-7_8

13. Egawa, R., Komatsu, K., Kobayashi, H., *et al.*: Potential of a modern vector supercomputer for practical applications: performance evaluation of SX-ACE. The Journal of Supercomputing 73(9), 3948–3976 (2017). https://doi.org/10.1007/s11227-017-1993-y

14. Egawa, R., Komatsu, K., Takizawa, H.: Designing an open database of system-aware code optimizations. In: 2017 Fifth International Symposium on Computing and Networking (CANDAR). pp. 369–374. IEEE Computer Society (2017). https://doi.org/10.1109/CANDAR.2017.102

15. Gharaibeh, A., Reza, T., Santos-Neto, E., *et al.*: Efficient large-scale graph processing on hybrid CPU and GPU systems. arXiv preprint arXiv:1312.3018 (2013)

16. Goldberg, A., Radzik, T.: A heuristic improvement of the Bellman-Ford algorithm. Tech. rep., Stanford Univ CA Dept of Computer Science (1993)

17. Gómez, C., Casas, M., Mantovani, F., Focht, E.: Optimizing sparse matrix-vector multiplication in NEC SX-Aurora Vector Engine. Tech. rep., Technical Report, Barcelona Supercomputing Center (2020)

18. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: Graphx: Graph processing in a distributed dataflow framework. In: 11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14). pp. 599–613 (2014)

19. Gustafson, J.L.: Reevaluating Amdahl's law. Communications of the ACM 31(5), 532–533 (1988). https://doi.org/10.1145/42411.42415

20. Han, M., Daudjee, K.: Giraph unchained: Barrierless asynchronous parallel execution in pregel-like graph processing systems. Proceedings of the VLDB Endowment 8(9), 950–961 (2015). https://doi.org/10.14778/2777598.2777604

21. Karypis, G., Kumar, V.: Metis: A software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices (1997), https://hdl.handle.net/11299/215346

22. Kleinberg, J.M., Kumar, R., Raghavan, P., *et al.*: The web as a graph: Measurements, models, and methods. In: International Computing and Combinatorics Conference. Lecture Notes in Computer Science, vol. 1627, pp. 1–17. Springer (1999). https://doi.org/10.1007/3-540-48686-0_1

23. Komatsu, K., Egawa, R., Hirasawa, S., Takizawa, H., Itakura, K., Kobayashi, H.: Migration of an atmospheric simulation code to an OpenACC platform using the Xevolver framework. In: 2015 Third International Symposium on Computing and Networking (CANDAR). pp. 515–520. IEEE Computer Society (2015). https://doi.org/10.1109/CANDAR.2015.102

24. Komatsu, K., Egawa, R., Hirasawa, S., Takizawa, H., Itakura, K., Kobayashi, H.: Translation of large-scale simulation codes for an OpenACC platform using the Xevolver framework. International Journal of Networking and Computing 6(2), 167–180 (2016). https://doi.org/10.15803/ijnc.6.2_167

25. Komatsu, K., Egawa, R., Isobe, Y., *et al.*: An approach to the highest efficiency of the HPCG benchmark on the SX-ACE supercomputer. In: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis (SC15), Poster. pp. 1–2 (2015)

26. Komatsu, K., Watanabe, O., Musa, A., *et al.*: Performance evaluation of a vector supercomputer SX-Aurora TSUBASA. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, Dallas, TX, USA, Nov. 11-16, 2018. pp. 54:1–54:12. SC '18, IEEE (2018). https://doi.org/10.1109/SC.2018.00057

27. Liu, H., Huang, H.H.: Enterprise: breadth-first graph traversal on GPUs. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 1–12. ACM (2015). https://doi.org/10.1145/2807591.2807594

28. Malewicz, G., Austern, M.H., Bik, A.J., *et al.*: Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. pp. 135–146. ACM (2010). https://doi.org/10.1145/1807167.1807184

29. Meyer, U., Sanders, P.: δ-stepping: a parallelizable shortest path algorithm. Journal of Algorithms 49(1), 114–152 (2003). https://doi.org/10.1016/S0196-6774(03)00076-2

30. Murphy, R.C., Wheeler, K.B., Barrett, B.W., Ang, J.A.: Introducing the graph 500. Cray Users Group (CUG) 19, 45–74 (2010)

31. Page, L., Brin, S., Motwani, R., Winograd, T.: The pagerank citation ranking: Bringing order to the web. Tech. rep., Stanford InfoLab (1999)

32. Pan, Y.: Multi-GPU Graph Processing. Ph.D. thesis, University of California, Davis (2019)

33. Soga, T., Musa, A., Okabe, K., *et al.*: Performance of SOR methods on modern vector and scalar processors. Computers & Fluids 45(1), 215–221 (2011). https://doi.org/10.1016/j.compfluid.2010.12.024

34. Tiskin, A.: The design and analysis of bulk-synchronous parallel algorithms. Ph.D. thesis, Citeseer (1998)

35. Wang, Y., Davidson, A., Pan, Y., *et al.*: Gunrock: A high-performance graph processing library on the GPU. In: Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. pp. 1–12. ACM (2016). https://doi.org/10.1145/2851141.2851145

36. Yamada, Y., Momose, S.: Vector engine processor of NEC brand-new supercomputer SX-Aurora TSUBASA. In: Intenational symposium on High Performance Chips (Hot Chips2018) (2018)

37. Zhong, J., He, B.: Medusa: Simplified graph processing on GPUs. IEEE Transactions on Parallel and Distributed Systems 25(6), 1543–1552 (2013). https://doi.org/10.1109/TPDS.2013.111