Joint Usage/Research Center for Interdisciplinary Large-scale Information Infrastructures
Final Report for JHPCN Joint Research of FY 2017, May 2018

jh170049-ISJ

Theory and Practice of Vector Processing for Data and Memory Centric Applications

Hiroaki Kobayashi （Graduate School of Information Sciences, Tohoku University）

Abstract
Since vector data processing is one of the key technologies for the next-generation supercomputing system, it is strongly required to clarify the capability of vector data processing. This project conducts the research on vector data processing potential that can be used both for development of new HPC systems and for the development of algorithms for extremely large-scale numerical problems. To this end, the research group of Lomonosov Moscow State University (MSU) that has experience with graph problems and the research group of Tohoku University that has experience with program development for vector supercomputers work together to clarify the potential of vector data processing on graph problems. The MSU group has developed various graph algorithms for an ultra-high scale that contains more than 10^9 vertices and edges. The research group of Tohoku University has knowhow to exploit the potential of vector supercomputers. By the international collaboration of the two groups, the theory and practice of vector data processing for ultra-high scale graph problems aims to be clarified. In this report, by the performance analysis of four graph problems on SX-ACE, the potential of the vector data processing to the graph problems is shown.

1. Basic Information

(1) Collaborating JHPCN Centers

Cyberscience Center, Tohoku University

(2) Research Areas

- ☐ Very large-scale numerical computation
- ☐ Very large-scale data processing
- ☐ Very large capacity network technology
- ■ Very large-scale information systems

(3) Roles of Project Members

This collaborative work is undertaken by Lomonosov Moscow State University (MSU) and Tohoku University. The researchers of MSU mainly focus on algorithm study and development of graph problems that can effectively utilize vector data processing. The researchers of Tohoku University conduct performance analysis and optimizations for obtaining aspects for development of next generation vector computer architectures.

Lomonosov Moscow State University (Russia)

- ☐ Vladimir Voevodin (Algorithm Development)
- ☐ Dmitry Nikitenko (Performance analysis and statistics)
- ☐ Alexander Antonov (Informational structure analysis)
- ☐ Alexey Teplov (Scalability and performance analysis)
- ☐ Ilya Afanasyev (Code design and code optimization)

Tohoku University (Japan)

- ☐ Hiroaki Kobayashi (Code optimization and performance analysis)
- ☐ Hiroyuki Takizawa (Code optimization and performance analysis)
- ☐ Akihiro Musa (Code design and code optimization)
- ☐ Ryusuke Egawa (Code optimization and performance analysis)
- ☐ Kazuhiko Komatsu (Code design and code optimization)

2. Purpose and Significance of the Research

The overall goal of this project is the deep research on vector processing potential

on large-scale numerical methods. The obtained knowledge of this project can be useful not only for developments of new vector supercomputing systems, but also for developments of extremely large-scale numerical solutions.

To this end, this project focuses on a graph problem that is one of the important extremely large-scale numerical solutions. A graph problem is well known as the basis for solution of the wide range of applied and scientific challenges such as communicational and transport network optimization, social network analysis, and web data analysis. To satisfy the demands of these challenges, a graph problem with more than 10^9 vertices and edges should be solved. The Russian research group tries to solve such huge-scale graph problems by effectively utilizing the vector data processing on vector supercomputing systems in this project. The Russian research group considers the efficient algorithm that is suitable for the vector data processing and implements the algorithm on vector supercomputing systems.

Furthermore, by collaborating with the Tohoku university research group that has a long experience with operation and development of vector supercomputing systems, the vector-friendly graph algorithm is further optimized and tuned so as to fully exploit the potential of vector supercomputing systems.

Through this collaborative work of both groups about theory and practice, these research efforts become public to share valuable results and knowledge. For example, as a part of the AlgoWiki project that is mainly developed and maintained in the Russian group, the valuable experimental data is open to be made commonly and widely available. These results are also positively utilized for development of other scientific applications as well as better understanding the requirements for the next generation vector processor and memory architecture.

The effective use of vector data processing becomes more essential for the future design of applications as vector data processing is widely used in x86 processors, IBM Power processors as well as vector processors. Therefore, the deep study on vector data processing is important for both supercomputer design and large-scale applications.

3. Significance as a JHPCN Joint Research Project

From the scientific aspects, this project can show the effectiveness of vector data processing in large-scale graph algorithms. The development in the vector-friendly graph algorithm is important. Thus, the JHPCN Joint Research Project is an excellent opportunity to conduct our joint interdisciplinary research between the Russian and Japan teams.

From the computational aspects, Russia has different supercomputing systems from Japan. Regarding vector supercomputing systems, currently, there are no vector systems in Russia. Thus, the JHPCN project helps to investigate and develop a graph algorithm that is suitable for vector data processing on vector supercomputing systems.

4. Outline of the Research Achievements up to FY2016

As this year is the first year, there are no research achievements up to FY2016.

5. Details of FY2017 Research Achievements

In the period of FY2017, we have conducted performance evaluation of four graph problems using an Intel Xeon Gold processor, an Intel Xeon Phi Knights Landing processor. Then, we have been working on deep performance and bottleneck analysis of graph algorithms on a vector processor, especially NEC SX-ACE.

5.1 Performance evaluation of Single Source Shortest Path (SSSP) problem

5.1.1 Problem description of SSSP

An undirected graph $G = (V, E)$ with vertices $V = (v_1, v_2, \dots, v_n)$ and edges $E = (e_1, e_2, \dots, e_m)$ is given. Each edge $e \in E$ has a weight value $w(e)$. The path between vertices u and v is defined as edges sequence $\pi_{u,v} = (e_1, \dots, e_k)$, beginning in vertex u and ending in vertex v, so each edge follows another one. The path length can be defined as $w(\pi_{u,v}) = \sum_{i=1}^{k} w(e_i)$. A path $\pi^*_{u,v}$, which has a minimal possible length between vertices u and v, is called the shortest path: $d(u, v) = w(\pi^*_{u,v}) = \min w(\pi_{u,v})$.

Depending on the choice of a vertex pair between which the search is performed, the shortest paths problem can be formulated in three different ways:

- **SSSP** (single source shortest paths) computes the shortest paths from a single selected source vertex.
- **APSP** (all pairs shortest paths) computes the shortest paths between all pairs of graph vertices.
- **SPSP** (some pairs shortest paths) computes the shortest paths between some pre-selected pairs of vertices.

In this project, the SSSP problem is examined, since it is the simplest and most basic for other problems: for example, the APSP problem for large-scale graphs can be solved by repeated calls of the SSSP operations for each source vertex, since traditional algorithms, such as Floyd-Warshal, cannot be applied because of the high memory costs.

5.1.2 Algorithm description

The SSSP problem can be solved with three traditional algorithms: Dijkstra, Bellman-Ford, and Delta-stepping.

- **Dijkstra's algorithm** is designed to solve the problem in graphs with edges, having only non-negative weights. Variation of the algorithm, implemented with Fibonacci heap has the asymptotically fastest time $O(|E| + |V| \log |V|)$. The computations of this algorithm include sequential traversal of vertices, starting from the source vertex, while putting adjacent vertices to the stack (heap) to be processed later. As a result, the algorithm can be executed only sequentially.
- **The Bellman-Ford algorithm** is designed to solve the problem in graphs, including edges with negative weights. The computational kernel of the algorithm consists of a few iterations, which require traverses of all graph edges; the array of distances is computed from the data of each edge. The computations continue until there are no changes in the distance array. The algorithm has sequential

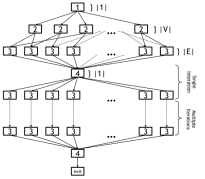

Figure 5.1 Informational graph of Bellman-Ford algorithm

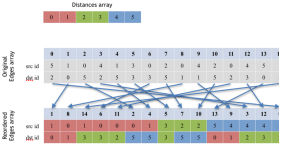

Figure 5.2 Graph edges reordering example for graph with 5 vertices and 16 edges

complexity equal to $O(p|E|)$, where p is the maximum possible length of the shortest path from the source vertex to any other. As a consequence, the worst-case complexity is equal to $O(|V||E|)$, however, for many real-world graphs, the algorithm is finished in a much smaller number of steps. Moreover, the algorithm has a significant parallel potential: its parallel complexity is equal to $O\left(p\frac{|E|}{N}\right)$, where N is the number of processors used.

- **The delta-stepping algorithm** is designed to solve the single source shortest paths problem using novel data-structures called *buckets*. It can be viewed as a generalization of the Bellman-Ford algorithm, aiming to improve its complexity in a *average* sense. For random directed graphs with $\frac{d}{|V|}$ and uniformly distributed edge weights, the algorithm completes computations in $O\left(\frac{(\log|V|)\hat{}3}{(\log\log|V|)}\right)$.

5.1.3 Implementation details of SSSP

First of all, it is very important to select the most suitable algorithm for the target NEC SX architecture. Since Dijkstra's algorithm is naturally sequential and requires complex data structures as well as the Delta-stepping algorithm, those algorithms are not implemented on NEC SX architecture. On the other hand, the Bellman-Ford algorithm does not require complex data structure processing. Moreover, each iteration of this algorithm is a parallel traversal of all graph edges, where parallel processing and vectorization can be applied.

During the selection of the most suitable algorithm for particular parallel architecture, parallel algorithm properties have to be studied. For this purpose, informational graphs, introduced in [1], can be used. Figure 5.1 demonstrates informational graphs of the Bellman-Ford algorithm.

Table 5.1. System configurations.

Model name	Intel Xeon Gold 6126	Intel Xeon Phi 7230	NEC SX-ACE
Socket	1	1	1
Cores per socket	12	64	4
CPU MHz	2601	1297	1000
L1d cache	32K	32K	1M
L2 cache	1024K	1024K	-
L3 cache:	19712K	-	-

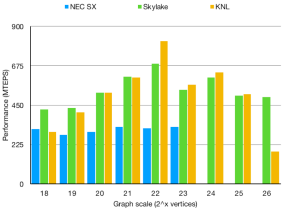

Figure 5.3 Performance (in MTEPS) of multi core implementations of the Bellman-Ford algorithm.

Parallel operations as shown in named 3 in Figure 5.1 correspond to independent updates of graph edges on each iteration, while operations in named 2 correspond to parallel initialization of distances array. Both those operations can be successfully parallelized and vectorized.

Before implementing the chosen algorithm, it is important to determine the storage data format for input graphs. For the Bellman-Ford algorithm, the most suitable format is an edges list, where each edge is stored as a triple {vertex-start, vertex-end, edge's weight}; all edges are stored in a single array in any order. Moreover, this format allows simpler vectorization and possible sorting-optimization, which aims to improve data locality. Thus, this format is going to be used for all implementations in this research project.

To achieve data locality improvement on per-edge updates (indirected memory accesses, parallel operations on informational graph), the main optimization used is graph edges reordering. This optimization significantly improves memory access pattern, which allows the algorithm to achieve much higher performance, since the data with indirect memory accesses are placed more locally and stored in caches for the longer period of time. The reordering is implemented in the following way: an array of distances is divided into segments (red, green and blue colors on Figure 5.2), whose sizes are equal to the size of the lowest level cache on the target architecture. Then, the edges are placed into the array in the following way: from the beginning of the array, an edge, whose source vertices belong to the first segment of distances array, is sorted. Then, the second and the third edges are sorted in order. Edges with the same segment number are sorted with the similar strategy, applied to their destination vertices.

Vectorization itself is achieved with help of *#pragma cdir nodep* directive and *sxc++* compiler.

5.1.4 Evaluation of the Bellman-Ford algorithm on NEC SX, Intel KNL, and Skylake systems

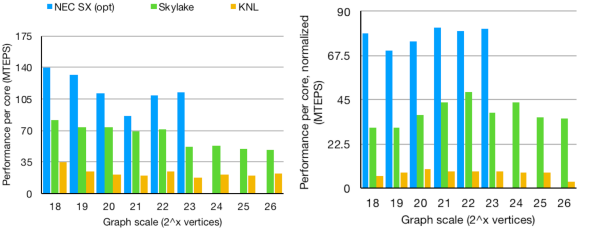

Figure 5.4 Performance (in MTEPS) of single core implementations of the Bellman-Ford algorithm.

This section describes the comparisons of performances of the Bellman-Ford algorithm among three different architectures: NEC SX-ACE, Intel Knight Landing (KNL), and Intel Skylake.

All results presented in this report have been obtained on two supercomputers: NEC SX-ACE in Tohoku University, and Lomonosov-2 in Lomonosov Moscow State University. Lomonosov-2 has several partitions with different configurations, including nodes with Intel Xeon Gold 6126 and Intel Xeon Phi 7230. The specifications of those systems are presented in Table 5.1. In general, the KNL has the highest number of lightweight cores among the reviewed architectures, in the meantime with the smallest amount of cache memory. Moreover, in current configuration on Lomonosov-2, KNL has high-bandwidth MCDRAM memory, which can be used as a large last-level cache on a variety of problems.

In order to compare different graph algorithms implementations between different architectures, a performance metric is widely used. When the algorithm processes a fixed graph with a total amount of |E| edges in T seconds, the performance can be calculated in this way.

$$performance(inTEPS) = \frac{|E|}{T} = \frac{edgescount}{executiontime}$$

Thus, the performance metric indicates the amount of Traversed Edges Per Second (TEPS) of a fixed graph during algorithm execution time. This metric can be calculated for both single-core and multicore implementations; Figure 5.3 demonstrates performance comparison between multicore implementations for three different architectures.

Since all three architectures have different numbers of cores, it is interesting to compare performance values obtained on a single core of each target processor. For these purposes we use 2 metrics: *performance per core*, which is a performance in TEPS metric, obtained on a single core of each

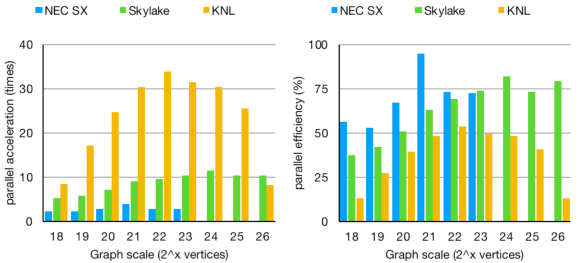

Figure 5.5 Performance (in MTEPS) of single core implementations of the Bellman-Ford algorithm.

target processor, and *performance per core normalized*, which is usual performance in TEPS obtained on maximum number of cores on a target processor, divided by the number of cores used during computations (4 for NEC SX, 12 for Skylake, 64 for Intel KNL). In general, a *performance per core* metric represents how well an algorithm utilizes single core computational capabilities of the target system, while a *performance per core normalized* metric takes into an account how well all the cores work together on a multicore system.

Figure 5.4 demonstrates performance per core comparison between implementations for three different reviewed architectures. These figures clearly show that the performance of NEC SX is comparable to or higher than those of Skylake and KNL in the case of the single core performance. This is because the algorithm implemented is suitable for the vector architecture. Thus, the average vector length and vector operation ratio, which are important metrics for the vector architecture, are very high of 99.62% and 255.9, respectively. Therefore, NEC SX could achieve such performance.

In order to further evaluate parallel efficiency of the developed implementations, we use another 2 widely used metrics: parallel efficiency and parallel acceleration, which can be calculated using the following formulas:

$$parallel\ acceleration = \frac{sequential\ time}{parallel\ time} = \frac{parallel\ performance}{sequential\ performance}$$

$$parallel\ efficiency = \frac{\frac{sequential\ time}{parallel\ time}}{cores\ number} * 100 = \frac{\frac{parallel\ performance}{sequential\ performance}}{cores\ number}$$

Figure 5.5 demonstrates the parallel

acceleration and parallel efficiency derived by these equations. The parallel efficiencies of NEC SX is higher than or comparable to those of Skylake. More investigation is required when the graph scale is more than 24 in the case of NEC SX.

5.2 Performance evaluation of BFS problem

5.2.1 Problem description of BFS

An undirected graph $G = (V, E)$ with vertices $V = (v_1, v_2, \dots, v_n)$ and edges $E = (e_1, e_2, \dots, e_m)$ is given. The path between vertices u and v is defined as a set arcs$\pi_{u,v} = (e_1, \dots, e_k)$, beginning in vertex u and ending in vertex v, so each edge follows another one. A length $d(u, v)$ of path $\pi_{u,v}$ between u and v vertices is the number of its arcs (edges). The breadth-first search finds the shortest $d(u, v)$ from a specified vertex u to all the other nodes. (1)

There are several possible modifications of the BFS problem:

- it is required only to check which vertices are reachable from specified source vertex; (2)
- it is required to find out «parent» vertex for each one. (3)

In this report, we have investigated and implemented the basic version (1) of the problems, such as SCC and transitive closure. The BFS employs the version (2).

5.2.2 Algorithms for BFS

BFS is a fundamental problem of graph processing, so there are a lot of algorithms that have been designed so far.

- **Frontier-based BFS** is a classical algorithm for solving the BFS problem on processors. The algorithm is based around storing vertices, reached on the current step in a separate data-structure called *frontier*. Also, there have been many implementations for multi-core CPUs and GPU.
- **Direction-optimizing BFS** is a novel approach to BFS, widely used nowadays. This approach uses two different *bottom-up* and *top-down* stages; the top-down stage is usually corresponding to the basic frontier-based approach, while the bottom-up stage is a reverse direction search. Combining these two approaches allows one to check significantly fewer edges, which results in significant accelerations obtained [2][3].

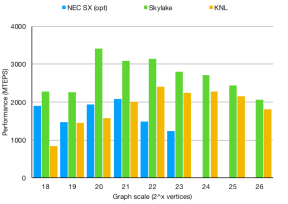

Figure 5.6 Performance (in MTEPS) of multi core implementations of the Bellman-Ford-based BFS algorithm for NEC SX, Intel KNL, and Skylake architectures.

- **SpMV-based BFS** is based on sparse-matrix dense-vector (SpMV) multiplications, to which the BFS problem can be reduced. A implementation of this method for vector architectures (Intel KNL) is described in [4], where authors claim that even though this approach has great computational complexity, it is still more suitable for modern vector architectures. It is shown that this approach demonstrates comparable results to the state-of-art implementations.
- **Bellman-ford based BFS** has approximately the same computational complexity as the SpMV-based BFS, but has different data-structures used. The idea of this approach is very similar to the

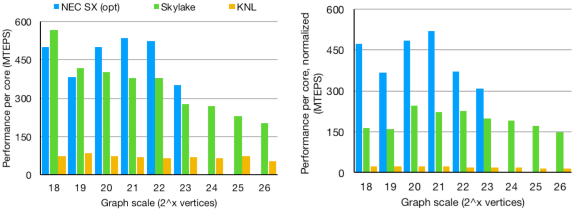

Figure 5.7 Performance (in MTEPS) of single core implementations of the Bellman-Ford-based BFS algorithm for NEC SX, Intel KNL, and Skylake architectures.

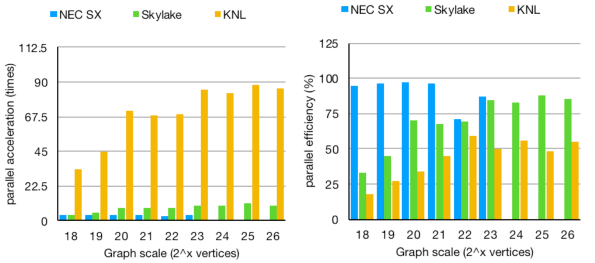

Figure 5.8 Parallel acceleration and parallel efficiency comparison for NEC SX, Intel KNL, and Skylake architectures.

Bellman-Ford shortest paths algorithm, where weight-distances updates are replaced with reachability or integer distances arrays.

5.2.3 Implementation details of BFS

In this report, the Bellman-Ford based BFS is implemented. It has very similar potential for vectorization and concurrency. The #pragma cdir *vprefetch* directive, the #pragma cdir *vovertake* directive, the #pragma cdir *vob* directive, and manual loop unrolling are used to effectively issue the vector gather/scatter instructions by overtaking other vector instructions.

5.2.4 Evaluation of the Bellman-Ford based BFS on NEC SX, Intel KNL, and Skylake systems

Similar to the evaluation of SSSP, Figure 5.6 demonstrates the performance comparison between multicore implementations for three different architectures. Figure 5.7 demonstrates performance per core comparison. Figure 5.8 demonstrates the parallel acceleration and parallel efficiency. From these figures, the parallel efficiencies of NEC SX are higher than those of the others, and the high efficiency can be obtained even when the graph scale is small. A high vector operation ratio of 99.62% and a high average vector length of 255.9 contribute to the high efficiency in the case of the single core performance.

5.3 Performance evaluation of the SCC problem

5.3.1 Problem description of SCC

A directed graph $G = (V, E)$ with vertices $V = (v_1, v_2, \ldots, v_n)$ and edges $E = (e_1, e_2, \ldots, e_m)$ is given. Edges may not have any data assigned (so graphs without edges weights are discussed in current section). A strongly connected component (SCC) of a directed graph G is a strongly connected subgraph, which is maximal within the following property: no additional edges or vertices from G can be included in the subgraph without breaking its property of being strongly connected.

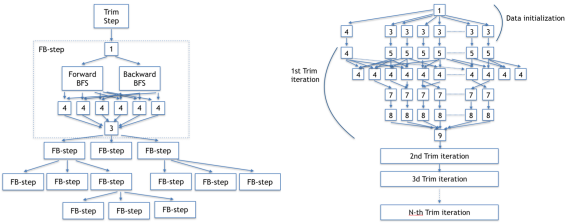

Figure 5.9 Informational graph of Forward-Backward-Trim algorithm: top-level (left), trim-step (right).

5.3.2 Algorithm description of SCC

SCCs in the BFS can be found with one of the following algorithms.

- **Tarjan's algorithm** is based on a single depth first search (DFS) and uses $O(|E|)$ operations. Due to the fact that the algorithm is based on the DFS, only a sequential implementation is possible.
- **The DCSC algorithm** (Divide and Conquer Strong Components), or FB (Forward-backward) is based on the BFS and requires $O(|E| * \log(|E|))$ operations. This algorithm is initially designed for parallel implementations: in each step, it finds a single strongly connected component and allocates the component up to three subgraph, each of which may contain other strongly connected components, and, as a result, this algorithm can be processed in parallel.
- **Variations of the DCSC algorithm** are proposed such as Coloring and FB with step-trim. These modified versions of the DCSC algorithm are described in detail in papers [5][6].

For obvious reasons, Tarjan's algorithm is not suitable for solving a problem on parallel architectures, since it is based on the depth first search as well as complex data structures (stack and queue) processing, which cannot be implemented efficiently on GPUs.

A large number of papers such as [7] have already investigated different variation of the DCSC algorithms. Different variations can be more or less effective for different types of graphs; the paper [7] concluded that the forward-backward-trim algorithm is the most efficient way to process graphs.

5.3.3 Implementation details of SCC

During the selection of the most suitable algorithm for particular parallel architectures, parallel algorithm properties have to be studied. For this purpose, informational graphs, introduced in [1], can be used.

Figure 5.9 demonstrates informational graph of the Forward-Backward-Trim algorithm. As one can see from the informational graph, the Forward-Backward-Trim algorithm is designed in the following way: on the first step (trim step), the removal of the strongly connected components of size 1 is performed. After that, in each step, the algorithm finds one nontrivial strongly connected component (FB-step) and allocates the component up to three subgraphs, each of which contains other components, and, more importantly, this algorithm can be processed in parallel. This step heavily relies on breadth-first search to find all vertices, which can be reached from the selected pivot vertex, and all vertices, from which pivot vertex can be reached. Thus, this algorithm has two levels of parallelism: *BFS level* and *parallel subgraphs handling level*, which are a big advantage for parallel target architectures.

It is important to describe that both the Trim and FB steps can be successfully vectorized, since they both have a lot of independent operations on bottom level as shown in boxes 3, 4, 5, 7, 8, and inside BFS in Figure 5.9.

As a result, the parallel Forward-backward-Trim algorithm is implemented in the following way. First, in order to avoid building a reverse (transposed) graph, which is necessary for efficient backward search implementation, a graph is converted to the storage format of an edge list. To further increase the performance, the input edges list can be pre-sorted as described in Section 5.1.3. The approach greatly improves data locality and cache usage. Moreover, this optimization also significantly improves the trim step efficiency, since it has the similar memory access pattern.

By the first analysis of the performance of SCC on NEC SX, as the tendency of the behavior is similar to the BFS, the similar optimizations are applied to the Bellman-Ford based BFS such as the insertions of directives and manual unrolling in order to enhance the overtake of vector gather/scatter instructions.

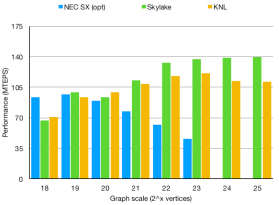

Figure 5.10 Performance (in MTEPS) of multi core implementations of Forward-Backward algorithm for NEC SX, Intel KNL and Skylake architectures.

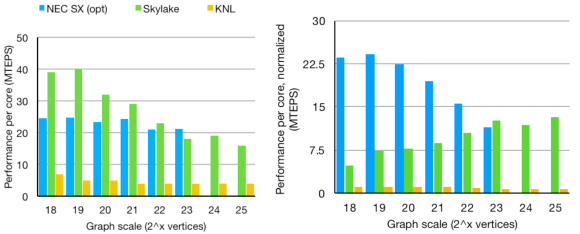

Figure 5.11 Performance (in MTEPS) of single core implementations of the Forward-Backward algorithm for NEC SX, Intel KNL, and Skylake architectures.

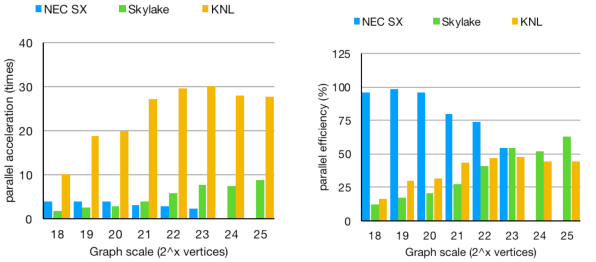

Figure 5.12 Parallel acceleration and parallel efficiency comparison for NEC SX, Intel KNL, and Skylake architectures.

5.3.4 Evaluation of the Forward-Backward algorithm on NEC, KNL, and Skylake systems

Figure 5.10 demonstrates the performance comparison between multicore implementations for three different architectures. Figure 5.11 demonstrates performance per core comparison. Figure 5.12 demonstrates the parallel acceleration and parallel efficiency. These figures show that the performance of NEC SX is good when the graph scale is small such as 18, 19, and 20. However, as the graph scale increase, the performance and efficiency decrease. In the case of KNL, the performance and efficiency are stable and in the case of Skylake, the performance and efficiency increase. More analysis and optimizations are necessary for NEC SX for the Forward-Backward algorithm, especially in the case of the larger graph scale.

5.4 Performance evaluation of Transitive Closure problem

5.4.1 Problem description of Transitive Closure

A directed graph $G = (V, E)$ with vertices $V = (v_1, v_2, \dots, v_n)$ and edges $E = (e_1, e_2, \dots, e_m)$ is given. The path between vertices u and v is defined as edges sequence $\pi_{u,v} = (e_1, \dots, e_k)$, beginning in vertex u and ending in vertex v, so each edge follows another one. Vertex v is reachable from vertex u, if at least a single path $P(u, v)$ between vertices u and v exists (every vertex is considered reachable from itself).

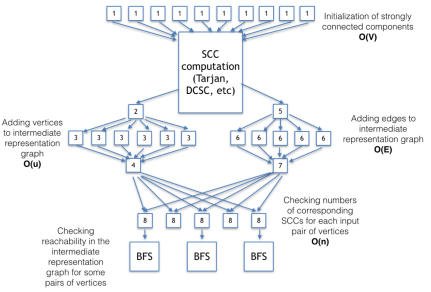

Figure 5.13 Informational graph of Purdom's algorithm.

Computing the transitive closure of graph $G = (V, E)$ means obtaining graph $G^+ = (V, E^+)$, where an edge $E(v, w)$ from G belongs to E^+ if and only if vertex w is reachable form vertex v in graph G. As a result, the transitive closure problem solution requires the $|V|^2$ storage space. Thus, the memory capacity of a node of modern computers is not enough even for medium-sized graphs (starting with around 2^{20} vertices). For this reason, the generalization of the transitive closure problem is used: only the specified pairs of vertices (u_1, v_1), (u_2, v_2), …, (u_n, v_n) are checked to belong to the transitive closure. The number of the pairs required to check n becomes an additional flexible algorithm parameter: varying it may greatly affect the overall algorithm performance.

5.4.2 Algorithm of Transitive Closure

The transitive closure computation problem in directed graph G can be solved using three different traditional approaches, described below.

1. The transitive closure computation can be reduced to the shortest paths computation in a corresponding graph with identical weights. Consequently, it can be solved with the Floyd-Warshall algorithm, introduced in [3] and [4]. This algorithm has the $O(|V|^3)$ computational complexity, and historically is the first developed algorithm for the transitive closure problem solution [5]. An important property of the Floyd-Warshall algorithm is $O(|V|^2)$ memory requirement for computations, which immediately reduces its applicability only to small-scale graphs.

Table 5.2 The comparison of sizes between original input graphs and corresponding intermediate representation graphs.

Count of vertices in the original graph	Distances array size in the original graph	Count of vertices in the intermediate representation graph	Distances array size in the intermediate representation graph	Graph type
1m	1 MB	14k	13 KB	RMAT[12]
8m	4 MB	659k	0.6 MB	RMAT
33m	35 MB	3.4m	3.2MB	RMAT
1m	1 MB	46k	44KB	SSCA2[13]
8m	4 MB	366k	357KB	SSCA2
33m	35 MB	1.45m	1.3MB	SSCA2

2. The transitive closure can be obtained using several multiple BFS, executed from each vertex of the graph. The BFS algorithm has been first described in [6]; the algorithm allows checking reachability between the selected source vertex and other vertices of the graph with only $O(|E|+|V|)$ operations required (using a queue-based implementation approach). As a result, the full transitive closure computation requires $O(|V| * (|V| + |E|))$ operations
3. Among the reviewed approaches, the most optimal computational complexity has Purdom's algorithm, introduced in [7]. Purdom's algorithm is based on the following idea: the transitive closure computation for graph G can be reduced to the transitive closure computation for graph G^-, obtained from graph G by collapsing G's strongly connected components into G^- vertices. The described approach provides $O(|E|+u|V|)$ computational complexity, where u is the number of edges in graph G^-. The provided estimate is based on the assumption that asymptotically optimal Tartan's algorithm ($O(|E|)$) is used for strongly connected components computation; if another algorithm, such as the DCSC is used, computational complexity can be different.

5.4.3 Implementation details of Transitive Closure

Based on computational complexity estimates, the most suitable approach to solve the transitive closure problem is Purdom's algorithm. However, during the selection of the most suitable algorithm for particular parallel architectures, the properties of the parallel algorithm have to be studied. Figure 5.13 demonstrates an informational graph of Purdom's algorithm. The presented graph is rather complicated and therefore includes two subgraphs, each one corresponding to an important algorithm building-block: the BFS and SCC computations.

Thus, the developed algorithm can be divided into 4 separate stages (steps):

1. Detecting strongly connected components in input graph G,
2. Creating intermediate representation

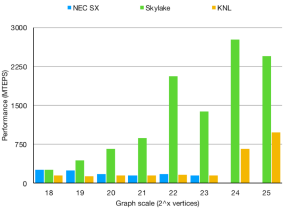

Figure 5.14 Performance (in MTEPS) of multi core implementations of the Purdom's algorithm for NEC SX, Intel KNL, and Skylake architectures.

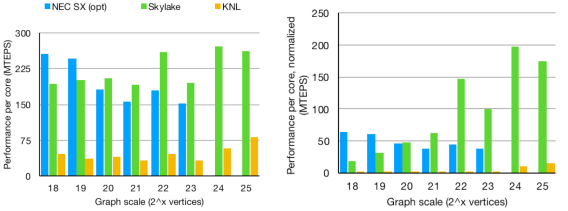

Figure 5.15 Performance (in MTEPS) of single core implementations of the Purdom's algorithm for NEC SX, Intel KNL, and Skylake architectures.

graph G^{-},

3. Computing an answer for all vertex pairs, related to the same strongly connected component,
4. Computing an answer for the rest pairs, using parallel BFS in the intermediate representation graph G^{-}.

If required, an intermediate representation graph can be saved to hard drive after stage 1. Then, the transitive closure can be found more efficiently due to no repeated SCC computation.

The edges sorting is not required for intermediate representation graph G^{-}, since usually these graphs have a significantly smaller number of vertices compared to the original input graphs (while it is still used for the original graph G in order to improve the SCC step performance). As a result, corresponding reachability arrays of intermediate representation graphs usually fit into caches. Table 5.2 demonstrates comparison between sizes of original input

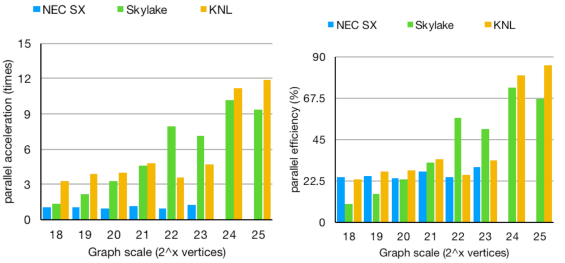

Figure 5.16 Parallel acceleration and parallel efficiency comparison for NEC SX, Intel KNL, and Skylake architectures.

graphs and graphs of corresponding intermediate representations.

In the loops, the container functions are utilized and these functions prevent vectorization. By avoiding using the functions, the compiler could vectorize the code.

5.4.4 Evaluation of Puron's algorithm on NEC, KNL, and Skylake systems

Figure 5.14 demonstrates the performance comparison between multicore implementations for three different architectures. Figure 5.15 demonstrates performance per core comparison. Figure 5.16 demonstrates the parallel acceleration and parallel efficiency. These figures show that the performances of NEC SX and KNL are not high compared to Skylake. In this algorithm, the deep analysis and further optimization are mandatory to exploit the potential of NEC SX-ACE.

References

[1] Voevodin, V.V. Parallel Computing. 608p. BHV, St. Petersburg (2002). (in Russian)

[2] Beamer, Scott, Krste Asanovi□, and David Patterson. "Direction-optimizing breadth-first search." *Scientific Programming* 21.3-4 (2013).

[3] Zhong, Jianlong, and Bingsheng He. "Medusa: Simplified graph processing on GPUs." *IEEE Transactions on Parallel and Distributed Systems* 25.6 (2014).

[4] Besta, Maciej, et al. "SlimSell: A Vectorizable Graph Representation for Breadth-First Search." *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*. IEEE, 2017.

[5] Fleischer, Lisa K, Bruce Hendrickson, and Ali Pınar. «On Identifying Strongly Connected Components in Parallel». In Lecture Notes in Computer Science, Volume 1800, Springer, 2000, pp. 505–11.

[6] Hong, Sungpack, Nicole C Rodia, and Kunle Olukotun. «On Fast Parallel Detection of Strongly Connected Components (SCC) in Small-World Graphs», Proceeedings of SC'13, 1–11, New York, New York, USA: ACM Press, 2013.

[7] Ji˙r ı Barnat, Petr Bauch, Lubos˙ Brim, and Milan C˙es˙ka. «Computing Strongly Connected Components in Parallel on CUDA». Faculty of Informatics, Masaryk University, Botanicka 68a, 60200 Brno, Czech Republic.

6. Progress of FY2017 and Future Prospects

As we planned, four graph algorithms have been implemented on NEC SX-ACE and conducted performance analysis by considering the characteristics of each algorithms. Through the initial analysis of the first versions of implementation, the optimization candidates are clarified. Then, optimizations for NEC SX-ACE have been applied to these four graph algorithms. By appropriate optimizations, the potential of the vector data processing to the graph problems is clarified.

For future work, further optimizations for Transitive Closure, which is one of the graph algorithms, need to be performed. Furthermore, as there are other graph algorithms that are not implemented on a vector architecture, these graph algorithms are the target to find more vector-friendly algorithms.

7. List of Publications and Presentations

(1) Journal Papers

(2) Conference Papers

1. Ilya Afanasyev, Vladimir Voevodin, “The comparison of large-scale graph processing algorithms implementation methods for Intel KNL and NVIDIA GPU”, large-scale graph processing Workshop, Sep. 2017.
2. Ryusuke Egawa, Kazuhiko Komatsu, Yoko Isobe, Toshihiro Kato, Souya Fujimoto, Hiroyuki Takizawa, Akihiro Musa and Hiroaki Kobayashi, “Performance and Power Analysis of SX-ACE using HP-X Benchmark Programs,” Proceedings of IEEE International Conference on Cluster Computing 2017, pp. 693-700, Sep. 2017.
3. Hiroyuki Takizawa, Thorsten Reimann, Kazuhiko Komatsu, Takashi Soga, Ryusuke Egawa, Akihiro Musa and Hiroaki Kobayashi, “Vectorization-aware Loop Optimization with User-defined Code Transformations”, Proceedings of IEEE International Conference on Cluster Computing 2017, pp. 685 – 692, Sep. 2017.
4. Hiroyuki Takizawa, Kenta Yamaguchi, Takashi Soga, Thorsten Reimann, Kazuhiko Komatsu, Ryusuke Egawa, Akihiro Musa and Hiroaki Kobayashi, "Migrating an old vector code to modern vector machines," accepted for presentation at The 30th International Conference on Parallel Computational Fluid Dynamics, 2018.
5. Hiroyuki Takizawa, Thorsten Reimann, Kazuhiko Komatsu, Takashi Soga, Ryusuke Egawa, Akihiro Musa, and Hiroaki Kobayashi, "Expressing the differences in code optimizations between Intel Knights Landing and NEC SX-ACE processors," accepted for presentation at the 13th World Congress on Computational Mechanics / 2nd Pan American Congress on Computational Mechanics, 2018.
6. Afanasyev Ilya, “An Efficient Implementation of the Transitive Closure Problem on Intel KNL Architecture,” the 3rd Ural Workshop on Parallel, Distributed, and Cloud Computing for Young Scientists, 2017, □. 10-19.
7. Afanasyev I., Voevodin Vl, “The Comparison of Large-Scale Graph Processing Algorithms Implementation

Methods for Intel KNL and NVIDIA," Supercomputing. RuSCDays 2017, series "Communications in Computer and Information Science", Springer, Vol. 793, p. 80-94.

(3) Oral Presentations

1. Ilya Afanasyev, "Efficient Implementation of the Transitive Closure Problem on Intel KNL Architecture", 3rd Ural Workshop on Parallel, Distributed, and Cloud Computing for Young Scientists (Ural-PDC 2017), Oct. 2017.
2. Hiroaki Kobayashi, "Two-Year Experiences with Vector Supercomputer SX-ACE and Design Space Exploration of the Next Generation Vector System", Keynote Speech in Russian Supercomputing Days 2017, Sep. 2017.
3. Alexander Antonov, Jack Dongarra, Dmitry Nikitenko, Pavel Shvets, Vladimir Voevodin, Changes in Architectures Require Changes in Implementation, not in Algorithms," Russian Supercomputing Days, Sep. 2017.
4. Vladimir Voevodin , The AlgoWiki Project and Challenges of the Well-Known Area," Invited talk at the 3rd International High Performance Computing Forum, Guanzhou, China, Sept. 2017.
5. Vladimir Voevodin., "The AlgoWiki Project: An Algorithmic Pillar of Exascale Computing," Invited talk at ATIP workshop "International exascale and next-generation computing programs", Nov. 2017.
6. Vladimir Voevodin, "Algorithms, Computing Platforms and Unlimited Freedom of Comparison," Invited talk at the 27th Workshop on Sustained Simulation Performance, March 2018
7. Vadim Voevodin, "HPC software for mass analysis of parallel applications efficiency," Invited talk at the 27th Workshop on Sustained Simulation Performance, March 2018.

(4) Others