Journal of the Operations Research Society of Japan
Vol. 54, No. 4, December 2011, pp. 259–280
© The Operations Research Society of Japan

NETAL: HIGH-PERFORMANCE IMPLEMENTATION OF NETWORK ANALYSIS LIBRARY CONSIDERING COMPUTER MEMORY HIERARCHY

Yuichiro Yasui Katsuki Fujisawa *Chuo University*
Kazushige Goto *Microsoft Corporation*
Naoyuki Kamiyama *Kyushu University*
Mizuyo Takamatsu *Chuo University*

(Received August 4, 2011; Revised October 31, 2011)

Abstract The use of network analysis has increased in various fields. In particular, a lot of attention has been paid to centrality metrics using shortest paths, which require a comparatively smaller amount of computation, and the global characteristic features within the network. While theoretical and experimental progress has enabled greater control over networks, large amounts of computation required for dealing with large-scale networks is a major hurdle. This research is on high-performance network analysis considering the memory hierarchy in a computer; it targets extremely important kernel types called *shortest paths* and *centrality*. Our implementation, called NETAL (NETwork Analysis Library), can achieve high efficiency in parallel processing using many-core processors such as the AMD Opteron 6174, which has the NUMA architecture. We demonstrated through tests on real-world networks that NETAL is faster than previous implementations. In the *all-pairs shortest paths* for the weighted graph `USA-road-d.NY.gr` (n =264K, m =734K), our implementation solved the *shortest path distance labels* in 44.4 seconds and the *shortest paths* with multiple predecessors in 411.2 seconds. Compared with the 9th DIMACS benchmark solver, our implementation is, respectively, 302.7 times and 32.7 times faster. NETAL succeeded in solving the *shortest path distance labels* for the `USA-road-d.USA.gr` (n =24M, m =58M) without preprocessing in 7.75 days. Numerical results showed that our implementation performance was 432.4 times that of the Δ-stepping algorithm and 228.9 times that of the 9th DIMACS benchmark solver. Furthermore, while GraphCT took 18 hours to compute the *betweenness* of `web-BerkStan`, our implementation computed *multiple centrality metrics* (*closeness*, *graph*, *stress*, and *betweenness*) simultaneously within 1 hour. A performance increase of 2.4-3.7 times compared with R-MAT graph was confirmed for SSCA#2.

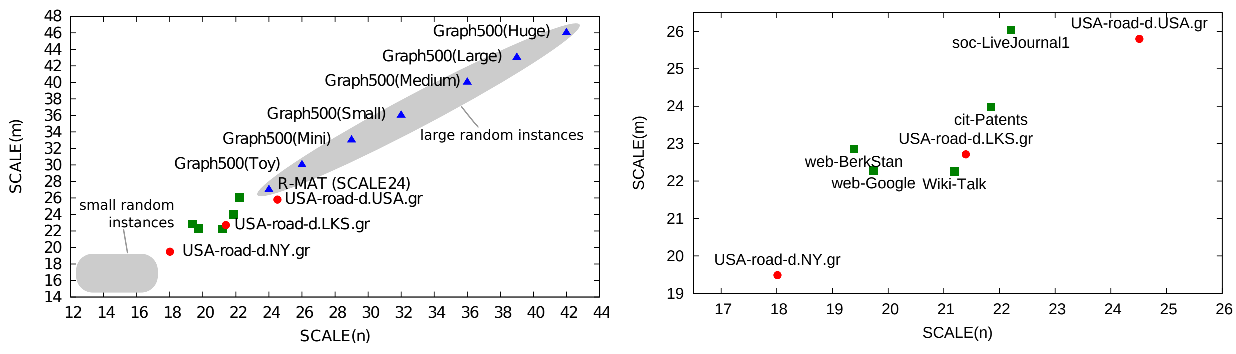
Keywords: Computer, graph theory, information technologies, optimization, shortest paths

1. Introduction

Theoretical progress in mathematics and advances in information technology have enabled us to control networks of a much larger scale than before. In particular, the extremely large scale of current networks used in various fields (healthcare, social-networks, intelligence gathering, systems biology, electric power grid, modeling, and simulation), has prompted studies of centrality using shortest paths, to reduce the amount of computations involved in controls. The well-known centrality metrics that use shortest paths are *closeness* [20], *graph* [14], *stress* [21], and *betweenness* [12]. In particular, *betweenness* is often used for graph analysis and clustering in networks that do not have coordinates (see, e.g., [6]). Although centrality metrics using shortest paths use fewer computations than other centrality metrics do, computation volumes similar to those of *all-pairs shortest paths* (APSP) are required to guarantee precision and accuracy. Even in the case of the Brandes' algorithm [1, 2], which efficiently uses *betweenness*, the process of finding the shortest paths is a bottleneck in terms of performance.

1.1. Related work

To date, various methods for finding the shortest path, such as Dijkstra's algorithm [10] have been studied. During the 9th DIMACS Implementation Challenge, a lively debate was conducted regarding high-performance *point-to-point shortest path* (P2PSP) computations, in which both a source node s and a destination node t are known, for a US road network with n=24M nodes and m=58M arcs. Because P2PSP computation requires several hours of advance preprocessing, it is not easy to apply its results to a centrality computation. Implementations using GPGPU [15, 19] and the *multi-source label-correcting algorithm* (MSLC) [22] (an extension of Dijkstra's algorithm) have been proposed for APSP computations of graphs at scales of several thousands to hundreds of thousands ($n = [2^{12}, 2^{17}]$, Figure 1 (a)). These algorithms use parallel computation after dividing APSP into n *single-source shortest path* (SSSP) parts and n/β *multi-source shortest paths* (MSSP) parts, where β is the number of source nodes. The *level-synchronized parallel breadth-first search* (LS-BFS), which does not consider a weighted graph, and the Δ-*stepping algorithm* (DS) [4, 17], which considers a weighted graph, are distributed memory algorithms for large-scale graphs with several billion nodes that are too big to handle on one computer ($n = [2^{26}, 2^{42}]$, Figure 1 (a)). However, the relevant computing environment uses parallel computers with distributed memory clusters and massively multithreaded computers with a shared memory. Furthermore, at Super Computing 2010, computer features and their evaluation were proposed through a graph search function named Graph500 List [24]. The ranking of Graph500 List is determined by the *traversed edges per second* (TEPS) ratio of the BFS in the Kronecker graph, which possesses a scale-free feature generated by the Kronecker product.



(a) real-world/random graph instance size (b) real-world instance size

Figure 1: (a) scale of real-world/random graph, (b) scale of real-world graph. The x-axis is the number of nodes on a log scale n, and y-axis is the number of arcs on a log scale m.

1.2. Our contributions

We propose an efficient multithreaded computation for general computing environments to calculate the shortest paths on typically sized graphs ($n = [2^{18}, 2^{25}]$, Figure 1 (b)). Our strategy is to divide the APSP into BFS, SSSP, and MSSP components using certain constraints (unweighted/weighted, distance labels/multiple predecessors) and to process these components in parallel. We implemented three strategies, i.e., n-BFS, n-Dijkstra, and n/β-MSLC for APSP, and two strategies, i.e., n-BFS, n-Dijkstra for centrality, and named the overall implementation NETAL (NETwork Analysis Library). The n-BFS and n-Dijkstra parts obtain all alternative shortest paths that are equal in length on the unweighted graph by using BFS (multipathBFS) and Dijkstra's algorithm with a binary heap (multipathSSSP), and our n/β-MSLC part computes the shortest path distance labels by using the MSLC

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

(distanceMSSP). Numerical experiments comparing ours with the previous implementations show that NETAL is very powerful. A brief summary of our contributions is as follows;

- Efficient multithreaded computation of APSP (or many BFSs, SSSPs, MSSPs) considering the computer memory hierarchy
  - Our strategy is to divide APSP into BFS, SSSP, and MSSP parts by using certain constraints (unweighted/weighted, distance labels/multiple predecessors), and to process these parts in parallel.
  - Our affinity-setting that considers the NUMA architecture ties each process to each processor core and each local memory.
- Performance evaluation
  1. Several experiments were conducted on evaluations speedup, conflicts between threads in parallel computation, affinity-setting, variances of execution times.
  2. Performance evaluation of two important applications: APSP and centrality
     - APSP computation use n-BFS, n-Dijkstra, and n/β-MSLC and was 17.9 times faster than MLB (the 9th DIMACS reference solver) and 302.7 times faster than the Δ-stepping algorithm. We obtained the shortest path distance labels of the US road network by using n/β-MSLC without preprocessing in 7.75 days.
     - The multiple centrality (*closeness*, *graph*, *stress*, and *betweenness*) computation used n-BFS and n-Dijkstra and was 8.6-33.3 times and 2.4-3.8 times faster than GraphCT and SSCA#2, respectively.

2. Shortest Paths

First, we will explain the shortest path problems and centrality using shortest paths. The shortest path problems can have several different constraints, i.e., unweighted or weighted, single/multiple predecessor(s), or distance labels only.

2.1. Shortest path problems

The foundational problems are the *breadth-first search* (BFS) and *single-source shortest path* (SSSP) problem. These problems are to find shortest paths from a single source node to all other nodes in given graph. In the **SSSP** problem, one is given a directed graph $G = (V, E)$, a non-negative arc weight function $\ell(e)(e \in E)$, and a source node s, and the goal is to find shortest paths from s to all other nodes $v \in V$ in the graph. The shortest paths so obtained are represented by the distance labels $d_G(s, v)$ and predecessors $\pi_s(v)$ for each node $v \in V$. If each predecessor has a single node $\pi_s(v) \in V \cup \emptyset$, we call this problem *singlepathSSSP*; otherwise, i.e., $\pi_s(v) = \{u \in V : (u, v) \in E, d_G(s, v) = d_G(s, u) + \ell(u, v)\}$, we call it *multipathSSSP*. If the shortest paths do not include predecessors, we call it *distanceSSSP*. In contrast, in the **BFS**, one is given only the directed graph $G = (V, E)$ and source node s, and the outputs are the distance labels (number of hops) $d_G(s, v)$ and predecessors $\pi_s(v)$ of the shortest paths from s to all other nodes $v \in V$. Similarly for the SSSP, we classify BFS into *singlepathBFS*, *multipathBFS*, and *distanceBFS* types. To compute SSSP several times on a graph, these types can be expressed as a *multi-source shortest paths* (MSSP) problem. In **MSSP** problem, one is given a directed graph $G = (V, E)$, a non-negative arc weight function $\ell(e)(e \in E)$, and a set of β source nodes $V_S = \{s_0, s_1, ..., s_{\beta-1}\}$, and the goal is to find the shortest paths for each source node. The so obtained shortest paths are represented by the distance labels $d_G(s, v)$ and predecessors $\pi_s(v)$ from each source node $s \in V_S$ to each node $v \in V$. Similarly to the SSSP and BFS, we classify MSSP into *singlepathMSSP*, *multipathMSSP*, and *distanceMSSP* types. If source nodes are adjacent (or nearly so), we can compute more efficiently by using the multi-source label-correcting algorithm (MSLC)

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

instead of each independent SSSP. The *all-pairs shortest paths* (APSP) problem is equal to n BFS, n SSSP problems, or n/β MSSP problems. In **APSP** problem, one is given a directed graph $G = (V, E)$, and a non-negative arc weight function $\ell(e)(e \in E)$, and the goal is to find the shortest path distance labels $d_G(u, v)$ and predecessors $\pi_u(v)$ of the shortest paths between all node pairs $(u, v), u, v \in V$. As is the case with SSSP and BFS, we can classify APSP into *singlepathAPSP*, *multipathAPSP*, and *distanceAPSP* types.

2.2. Labeling algorithm for BFS, SSSP, and MSSP

It is possible to classify algorithms for solving the shortest path problems into label-setting or label-correcting algorithms. Both these labeling algorithms update the temporary distance labels $d(s, v)$ initialized by $d(s, v) = \infty$ from the source node $s \in V$ to each node $v \in V$. The search starts from the given source node s, and when inequality $d(s, w) > d(s, v) + \ell(v, w)$ is satisfied for the distance labels for nodes w connected to the selected nodes v for each iteration, the search is updated to $d(s, w) \leftarrow d(s, v) + \ell(v, w)$. The iteration continues until there are no more candidate nodes. After the algorithm terminates, the temporary distance label $d(s, v)$ is equal to the shortest path distance label $d_G(s, v)$. Using label-setting algorithms, the node selected in each iteration is the unexplored node with the minimum distance label. As a result, each node is selected at most once. In contrast, in the case of label-correcting algorithms, the selected node is not necessarily the minimum distance label. As a result, the same node might be selected multiple times. Label-correcting algorithms are easier to parallelize than label-setting algorithms. Table 1 summarizes the features of label-setting and label-correcting algorithms in the case of the 256×256 grid graph `Square.16.0.gr` (number of nodes $n = 65536$, number of arcs $m = 261120$) used in the 9th DIMACS challenge. The weight of each arc is a random number in the range $[0, 1024]$ (maximum arc weight $C = 1024$ and maximum degree of $d = 4$). The listed complexity is the amount of computation required for each problem (BFS, SSSP, and MSSP), and the CPU time is the time required for sequential execution of APSP on a Intel Xeon X5460 using a single thread.

Table 1: Sequential performance (CPU time in seconds) of APSP (`Square.16.0.gr`) computation attained by each algorithm on Intel Xeon X5460

implementation	algorithm	complexity	CPU time
BFS (breadth-first search)			
BFS [25]	naive BFS	$O(m)$	530.99 s
LS-BFS [4]	level-synchronized parallel BFS	$O(m)$	117.10 s
Label-setting algorithm (Dijkstra's algorithm) for SSSP			
DIKQ [9]	naive Dijkstra's algorithm	$O(n^2)$	12610.28 s
DIKH [9]	k-heap ($k = 4$)	$O(m \log_k n)$	961.89 s
DIKB [9]	Dial's algorithm	$O(m + nC)$	771.02 s
DIKBD [9]	double buckets	$O(m + n\sqrt{C})$	875.69 s
DIKF [9]	Fibonacci heaps	$O(m + n \log n)$	2310.56 s
MLB [13]	multi-level buckets	$O(m + n \log C)$	869.23 s
Label-correcting algorithm for SSSP			
BFM [9]	naive Bellman-Ford-Moore	$O(nm)$	3050.17 s
DS [17]	Δ-stepping algorithm	$O(dn)$	1244.08 s
Label-correcting algorithm for MSSP (β source nodes)			
MSLC [22]	multi-source label correcting algorithm ($\beta = 128$)	$O(\beta m \log n)$	118.02 s

Label-setting algorithm

The variations of Dijkstra's algorithm, i.e., DIKQ and DIKH, DIKB, DIKBD, DIKF, and MLB, that use a priority queue, are all label-setting algorithms (Table 1). Cherkassky, Gold-

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

berg, and Radzik [9] performed theoretical and experimental evaluations of DIKQ, DIKH, DIKB, DIKDB, and DIKF and showed DIKDB to be effective for most graph instances. As shown in Table 1, it is not guaranteed that the computation volume is a good estimate of the actual performance. Goldberg [13] devised MLB that is a simple high-performance priority-queue based on a RAM model. This algorithm was chosen as the 9th DIMACS reference implementation [25].

Label-correcting algorithm

BFM, DS, and MSLC are label-correcting algorithms. Madduri and Bader *et al.* [17] devised DS, which is a parallel algorithm for large-scale sparse graphs with small diameters. However, only shared memory-type special computing environments such as Cray XMT and Cray MTA-2 support threaded parallel computations, and it is difficult to obtain this functionality with in general computers. Yanagisawa [22] devised MSLC for MSSP. It is based on Dijkstra's algorithm, and it computes the MSSP for the set of β adjacent (or somewhat close) source nodes $V_S = \{s_0, s_1, ..., s_{\beta-1}\}$. With MSLC, because each node $v \in V$ has a distance label $d(s, v)$ for each source node $s \in V_S$, multiple distance labels $d(s, w)$ can be updated simultaneously for the nodes $w \in V, (v, w) \in E$ connected to the selected node v. In such cases, it is possible to apply a priority queue to the set of candidate nodes in the same manner as that of the Dijkstra's algorithm, and the priority key can use the minimum distance label $k(v) = \min \{d(s, v) : s \in V_S\}$ for the node v from the set of source nodes V_S in question. As it is a natural extension of Dijkstra's algorithm, it has the same computation volume as Dijkstra's algorithm. Yanagisawa confirmed a performance increase of 3.6-3.9 times in comparison with a single thread MSLC on an Intel Xeon X5460 executing a four-threaded parallel computation. Because the required memory volume is large when $\beta = 128$ and there is a large set of source nodes, it is difficult to use MSLC on large-scale graphs, as demonstrated by the fact that it uses a 32-bit floating number and might result in an inaccurate solution.

2.3. All-pairs shortest paths

Applications such as computing centrality metrics are APSPs; however, an APSP is not directly computable for the instance sizes that are the subject of this research as they require large amounts of memory. From the results of single-thread computations shown in Table 1, we can conjecture that turning APSP into BFS, SSSP, and MSSP threaded parallel computations would be a simple and effective method. However, compared with simple sequential computing, it is extremely likely that performance will deteriorate because the computational resource conflicts between threads will be large. We propose a parallel computation method that avoids such conflicts for the three sub-problems - multipathBFS (Algorithm 1), multipathSSSP (Algorithm 2), and distanceMSSP (Algorithm 3). We call the implementations of three sub problems as *n-BFS*, *n-Dijkstra*, or n/β*-MSLC*, where n is the number of nodes and β is the number of source nodes. In n-Dijkstra, each SSSP is solved by using Dijkstra's algorithm with a binary heap. In n/β-MSLC, each MSSP with β source nodes is solved by using the MSLC with a binary heap.

2.4. Centrality

The centrality of each node is defined by using shortest paths as follows: First, for the directed graph $G = (V, E)$ and $s, t \in V$, denote $d_G(s, t)$ as the shortest (s, t)-path distance label. Let σ_{st} be the number of shortest (s, t)-paths, and let $\sigma_{st}(v)$ be the number of shortest (s, t)-paths on which v lies. If $s = t$, let $\sigma_{st} = 1$, and if $v \in \{s, t\}$, let $\sigma_{st}(v) = 0$. The centrality metrics using shortest paths are called *closeness* C_C, *graph* C_G, *stress* C_S, and *betweenness* C_B (Table 2). If a graph has an arc weight function $\ell(e), e \in E$, the

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

Algorithm 1 multipathBFS(G, s)

```
Input: directed graph $G = (V, E)$, source node $s$
Output: shortest path distance labels $d_G(s, v)$, number of shortest paths $\sigma_s(v)$, predecessors $\pi_s(v)$,
    $v \in V$; stack $\mathcal{S}$
 1: $\mathcal{S} \leftarrow$ empty stack
 2: $\pi_s(w) \leftarrow$ empty list, $w \in V$
 3: $\sigma(s,t) \leftarrow 0, t \in V$;  $\sigma(s,t) \leftarrow 1$
 4: $d(s,t) \leftarrow -1, t \in V$;  $d(s,s) \leftarrow 0$
 5: $Q \leftarrow$ empty queue;  enqueue $s \rightarrow Q$
 6: while $Q \neq \emptyset$ do
 7:    dequeue $v \leftarrow Q$; push $v \rightarrow \mathcal{S}$
 8:    for all $w : (v, w) \in E$ do
 9:       if $d(s,w) < 0$ then  $d(s,w) \leftarrow d(s,v) + 1$;  enqueue $w \leftarrow Q$
10:       if $d(s,w) = d(s,v) + 1$ then  $\sigma(s,w) \leftarrow \sigma(s,w) + \sigma(s,v)$;  append $v \rightarrow \pi_s(w)$
11:    end for
12: end while
13: $d_G(s,t) \leftarrow d(s,t), t \in V$
```

Algorithm 2 multipathSSSP(G, ℓ, s) - Dijkstra's algorithm with priority queue

```
Input: directed graph $G = (V, E)$, non-negative arc weighted function $\ell(e) \in E$, source node $s$
Output: shortest path distance labels $d_G(s, v)$, number of shortest paths $\sigma_s(v)$, predecessors $\pi_s(v)$,
    $v \in V$; stack $\mathcal{S}$
 1: $\mathcal{S} \leftarrow$ empty stack
 2: $\pi_s(w) \leftarrow$ empty list, $w \in V$
 3: $\sigma(s,t) \leftarrow 0$, $t \in V$;  $\sigma(s,s) \leftarrow 1$
 4: $d(s,t) \leftarrow \infty$, $t \in V$;  $d(s,s) \leftarrow 0$
 5: $Q \leftarrow$ empty priority queue;  insert $s \rightarrow Q$
 6: while $Q \neq \emptyset$ do
 7:    extract $v \leftarrow Q$ with minimum $d(s,v)$;  push $v \rightarrow \mathcal{S}$
 8:    for all $w : (v, w) \in E$ do
 9:       if $d(s,w) > d(s,v) + \ell(v,w)$ then
10:          $d(s,w) \leftarrow d(s,v) + \ell(v,w)$;  insert $w \rightarrow Q$;  $\pi_s(w) \leftarrow$ empty list
11:       end if
12:       if $d(s,w) = d(s,v) + \ell(v,w)$ then  $\sigma(s,w) \leftarrow \sigma(s,w) + \sigma(s,v)$;  append $v \rightarrow \pi_s(w)$
13:    end for
14: end while
15: $d_G(s,t) \leftarrow d(s,t), t \in V$
```

corresponding centrality metrics are called weighted.

Table 2: Centrality metrics using shortest paths

closeness [20]	$C_C(v) = \dfrac{1}{\sum_{t \in V} d_G(v,t)}$	*graph* [14]	$C_G(v) = \dfrac{1}{\max_{t \in V} d_G(v,t)}$
stress [21]	$C_S(v) = \sum_{s \neq v \neq t \in V} \sigma_{st}(v)$	*betweenness* [12]	$C_B(v) = \sum_{s \neq v \neq t \in V} \dfrac{\sigma_{st}(v)}{\sigma_{st}}$

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

Algorithm 3 distanceMSSP(G, ℓ, V_S) - multi-source label correcting algorithm (MSLC)

```
Input: directed graph $G = (V, E)$, non-negative arc weighted function $\ell(e) \in E$, set of source nodes $V_S = \{s_0, s_1, ..., s_{\beta-1}\}$
Output: shortest path distance labels $d_G(s, v), s \in V_S, t \in V$
 1: $d(v,t) \leftarrow \infty, v \in V \backslash V_S, t \in V$;  $d(s,t) \leftarrow 0, s \in V_S, t \in V$
 2: $Q \leftarrow$ empty priority queue;  insert $V_S \rightarrow Q$
 3: while $Q \neq \emptyset$ do
 4:     extract $v \leftarrow Q$ with minimum $k(v)$
 5:     for all $w : (v, w) \in E$ do
 6:         $updated \leftarrow false$
 7:         for all $s \in V_S$ do
 8:             if $d(s,w) > d(s,v) + \ell(v,w)$ then  $d(s,w) = d(s,v) + \ell(v,w)$;  $updated \leftarrow true$
 9:         end for
10:         if $updated = true$ then  $k(w) \leftarrow \min_{s \in V_S} d(s,w)$;  insert $w \rightarrow Q$ with $k(w)$
11:     end for
12: end while
13: $d_G(s,t) \leftarrow d(s,t), s \in V_S, t \in V$
```

Brandes' algorithm for *betweenness*

Brandes proposed in [1, 2] an efficient algorithm for *betweenness* (Algorithm 4). This algorithm repeatedly computes a **shortest path phase** (line 2) and an **update phase** (line 3-7) for each node. The shortest path phase computes the predecessor list $\pi_s(v)$ and the number of shortest paths $\sigma(s, v)$ for each node $v \in V$ from the specified source s, and the update phase updates the centrality index $C_B(v)$ for each node v in the order in which nodes are separated from the source by using a FILO stack $\mathcal{S}$ of the selected node history. The computational complexity of BFS in which the arc weight is not considered is $O(m)$, and whereas the complexity of Dijkstra's algorithm with a binary heap in which arc weight is considered is $O(m \log n)$. The computation volume of the update phase is $O(m)$. Therefore, as the computation volume is $O(nm)$ for the centrality computation when arc weight is not considered, and $O(nm \log n)$ when arc weight is considered, the overall performance dependent on selected algorithm for the shortest path phase. Bader et al. [3] proposed a simple random sampling method for this algorithm. This method only computes the iterations for all nodes (line.1) for a subset V' randomly sampled from the set of nodes V and obtains an approximate index $C'_B(v)$ for the gained betweenness index as $C'_B(v) \leftarrow \frac{|V|}{|V'|} \cdot C_B(v), \forall v \in V$.

Algorithm 4 Brandes' algorithm for (weighted) *betweenness* C_B

```
Input: directed graph $G = (V, E)$, (non-negative arc weighted function $\ell(e) \in E$)
Output: $C_B(v), \forall v \in V$ (initialize to 0)
 1: for $s \in V$ do
 2:     $\sigma, \pi_s, \mathcal{S} \leftarrow$ multipathBFS$(G, s)$ (multipathSSSP$(G, s, \ell)$)
 3:     while $\mathcal{S} \neq \emptyset$ do
 4:         pop $w \leftarrow \mathcal{S}$
 5:         for $v \in \pi_s(w)$ do  $\delta_B(v) \leftarrow \delta_B(v) + \frac{\sigma(s,v)}{\sigma(s,w)} \cdot (1 + \delta_B(w))$
 6:         if $w \neq s$ then  $C_B(w) \leftarrow C_B(w) + \delta_B(w)$
 7:     end while
 8: end for
```

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

Multiple centrality metrics computation

Because the shortest path phase requires the most computation time, the obtained shortest path should be used to calculate centrality metrics more. Here, we computes simultaneously C_C, C_G, C_S, and C_B.

Algorithm 5 Multiple centrality metrics computation for (weighted) C_C, C_G, C_S, C_B

```
Input: directed graph $G = (V, E)$, (non-negative arc weighted function $\ell(e) \in E$)
Output: $C_C(v)$, $C_G(v)$, $C_S(v)$, $C_B(v)$, $\forall v \in V$ (initialize to 0)
1: for $s \in V$ parallel do
2:     $d_G, \sigma, \pi_s, \mathcal{S} \leftarrow$ multipathBFS($G, s$) (multipathSSSP($G, s, \ell$))
3:     $C_C(s) \leftarrow \frac{1}{\sum_{t \in V} d_G(s,t)}$; $C_G(s) \leftarrow \frac{1}{\max_{t \in V} d_G(s,t)}$
4:     while $\mathcal{S} \neq \emptyset$ do
5:         pop $w \leftarrow \mathcal{S}$
6:         for $v \in \pi_s(w)$ do $\delta_S(v) \leftarrow (1 + \delta_S(w))$; $\delta_B(v) \leftarrow \delta_B(v) + \frac{\sigma(s,v)}{\sigma(s,w)} \cdot (1 + \delta_B(w))$
7:         if $w \neq s$ then $C_S(w) \leftarrow C_S(w) + \sigma(s,w) \cdot \delta_S(w)$; $C_B(w) \leftarrow C_B(w) + \delta_B(w)$
8:     end while
9: end for
```

3. Fast Implementation Based on Computer Memory Hierarchy

We devised a method of multithreaded computation based on the memory hierarchy. The method consists of three algorithms: n-BFS, n-Dijkstra, and n/β-MSLC, for APSP in the case of a directed graph. n-BFS and n-Dijkstra compute multipathBFS and multipathSSSP n times each, and n/β-MSLC computes distanceMSSP n/β times. First, computational resource requests were quantified for the labeling algorithm for the partitioned SSSP, and a suitable algorithm was selected on the basis of the results. We [23] have already implemented Dijkstra's algorithm with a binary heap (2-HEAP) that considers memory hierarchy. It was four times faster than the 9th DIMACS benchmark for the same number of memory requests. We incorporated 2-HEAP in n-BFS, n-Dijkstra, and n/β-MSLC. Furthermore, we improved memory referencing through a memory layout that considers the general CPU architecture and succeeded in improving performance. These implementations operate with 64-bit integers.

3.1. Memory hierarchy of computer

Progress in information technology has made it difficult to separate theoretical and experimental improvements to algorithms. First, we will discuss the memory hierarchy [16] shown in Figure 2. In the higher layers of this hierarchy, the access speed is high and the memory capacity is small, whereas in the lower layers, the access speed is low and the memory capacity is large. In particular, a CPU possesses extremely high-speed memory areas such as registers, cache memory, and TLB (Translation Look-aside Buffer). Although programs can be executed through the registers in order to obtain extremely high access speeds, the capacity of registers is extremely small. Conversely, at more than several gigabytes, the main memory (RAM) has an extremely large, but its access speed is extremely low compared with registers. Thus, it is extremely important to appropriately position data by considering the computation volume and amount of data to be transferred and by effectively using cache memory between registers and the main memory.

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

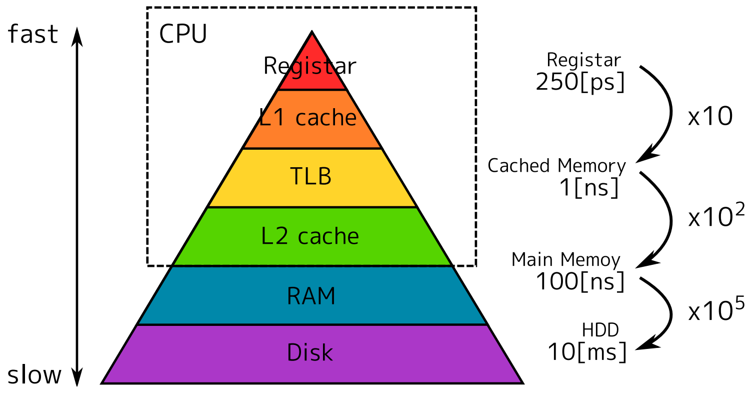


Figure 2: Computer memory hierarchy

3.2. Bottleneck analysis using resource request collisions

Because of the complexity of the memory hierarchy, the distance between the processor cores is not fixed. For a 2-way Intel Xeon X5460 (Figure 3), depending on the increasing order of the distance between the two cores, the memory structure can be divided up into "different processors," "(same processor) different L2 caches," "(same processor) same L2 cache" (Figure 4). The memory paths of the Intel Xeon X5460 are shared in the case of "different processors," and the relative bandwidth between the processor and RAM is half that of a "sequential" processor. In the case of "different L2 caches," the relative bandwidth within the processor is half that of "different processors." Furthermore, because it shares the L2 cache, the "same L2 cache" uses only half the area and bandwidth of "different L2 caches." In other words, as the distance between the processors decreases and the percentage of shared computational resources increases, performance may decrease as a result of resource request collisions. This property enables us to specify the bottlenecks in the memory hierarchy by computing the rate of performance reduction from collisions of computational resource requests for the "sequential" processor (Table 3). As this bottleneck analysis runs the same sequential program simultaneously, the data stored in the cache by one processor cannot be used by another processor, and the number of resource requests is simply double that of single processor. The `numactl` library is used to select the executing processor and core.

Table 3: Bottleneck analysis of the memory hierarchy by considering the rate of performance reduction due to simultaneous execution

bottleneck	different processors	different L2 caches	same L2 cache
processor $\leftrightarrow$ RAM bandwidth	down	down	down
processor inside bandwidth	-	down	down
L2 cache sharing	-	-	down
Arithmetic performance	-	-	-

Table 4 lists the results of the computational resource conflict tests for the labeling algorithm in the 2-way Intel Xeon X5460. For comparison, we used the SSSP computation in relation to `USA-road-d.USA.gr` (n =24M, m =58M), in terms of increase in the percentage of shared computational resources, performance falls in the order "sequential," "different processors," "different L2 caches," and "same L2 cache" smallest reduction. DIKH and DIKF, with the heap-based priority queue, show less of a low performance decline and higher parallel efficiency in comparison with DIKB, DIKBD, and MLB, which use a bucket-

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

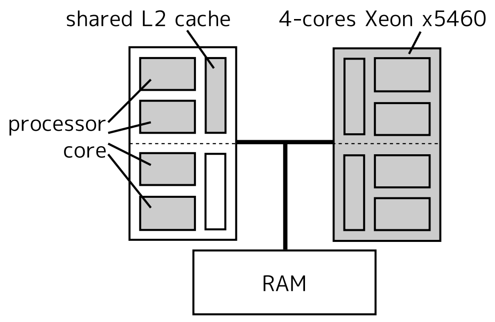


Figure 3: 2-way Intel Xeon X5460

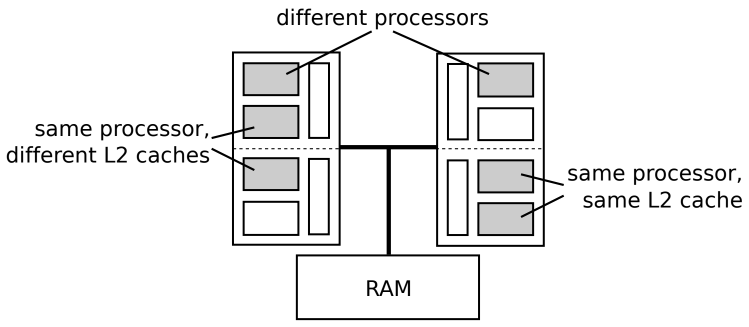


Figure 4: Two-core combination on 2-way Intel Xeon X5460

based priority queue. DS has characteristics similar to those of the bucket-based priority queue. The 2-HEAP [23], which uses a binary heap, was a shorter execution time, which is a measure of absolute performance, and fewer computational resource conflicts, which is a measure of relative performance. The memory hierarchy is useful not only for calculating the shortest path but also for other computations that require a large memory bandwidth.

Table 4: Computational results (CPU time in seconds and performance ratio %) for simultaneous tests using two processor cores on a 2-way Intel Xeon X5460

	sequential	different processors	different L2 caches	same L2 cache
2-HEAP[23]	5.34 s (± 0.00%)	5.44 s (- 1.93%)	5.62 s (- 5.05%)	6.63 s (-18.94%)
DIKH($k = 4$)	7.23 s (± 0.00%)	7.26 s (- 0.41%)	7.59 s (- 4.74%)	8.79 s (-17.75%)
DIKF	15.95 s (± 0.00%)	16.09 s (- 0.87%)	16.56 s (- 3.68%)	18.17 s (-12.22%)
DIKB	4.38 s (± 0.00%)	4.54 s (- 3.52%)	5.01 s (-12.58%)	6.38 s (-31.35%)
DIKBD	4.65 s (± 0.00%)	4.88 s (- 4.71%)	5.25 s (-11.43%)	6.64 s (-29.97%)
MLB	5.69 s (± 0.00%)	5.85 s (- 2.74%)	6.17 s (- 7.78%)	7.73 s (-26.39%)
DS	11.74 s (± 0.00%)	12.06 s (- 2.66%)	12.55 s (- 6.41%)	16.49 s (-28.76%)

Table 5 summarizes the threaded parallel processing performance of SSSP for the road networks `USA-road-d.NY.gr` (n =264K, m =734K) and `USA-road-d.USA.gr` (n =24M, m =58M). For the sequential implementation, 2-HEAP is at approximately the same level as MLB, i.e., the 9th DIMACS reference implementation. In contrast, the threaded processing gives high efficiency. The four-threaded processing of 2-HEAP enables an ideal parallelization by avoiding the "same L2 cache" level. In contrast, in eight-threaded processing, a performance efficiency deteriorates owing to computational resource conflicts at the "same L2 cache" level. These are consistent with the results listed in Table 4. As these implementations output only one shortest path, they must be amended in order to list multiple shortest paths necessary for the centrality computation and counting the number of shortest paths.

3.3. Data referencing improvement based on CPU architecture

Processor architectures are transitioning from UMA (Uniform Memory Access), such as Intel Xeon X5460 (Figure 3), to NUMA (Non-Uniform Memory Access). In case of the NUMA architecture-based 12-core AMD Opteron 6174 (Figure 5), the each core has a separate L2 cache, and six cores share an L3 cache. Each processor has a separate memory (local memory) directly beneath it and is connected to a dedicated memory bus. To reference memory located under other processors (remote memory), it is necessary to reference the data via the processor. In this manner, the memory hierarchy becomes more complex in

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

Table 5: Parallel performance (CPU time, speedup ratio, and memory requirement) of SSSP on a 2-way Intel Xeon X5460

	`USA-road-d.NY.gr`		`USA-road-d.USA.gr`	
	CPU time (speedup ratio)	MB	CPU time (speedup ratio)	MB
2-HEAP(1 threads)	35.69 ms (× 1.00)	10.70	5300.8 ms (× 1.00)	902
2-HEAP(2 threads)	18.02 ms (× 1.98)	14.80	2734.4 ms (× 1.94)	1267
2-HEAP(4 threads)	8.94 ms (× 3.99)	22.99	1451.3 ms (× 3.65)	1998
2-HEAP(8 threads)	4.85 ms (× 7.36)	39.38	1031.7 ms (× 5.14)	3460
MLB	41.53 ms	25.32	5873.0 ms	2169

the NUMA architecture, and as the distance varies between core memories, it is necessary to consider how best to allocate processing resources to improve computing performance. However, such an allocation will have to be left as a subject for future research because current OSs have no functions to allocate the cores efficiently.

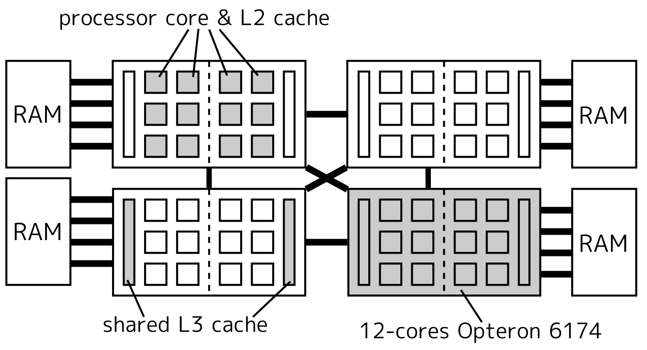


Figure 5: NUMA architecture 4-way AMD Opteron 6174

The graph data that is shared and referenced by multiple processor cores and graph data that is required for the particular work area of each thread are treated differently when performing threaded parallel processing to find the shortest path. For efficient threaded parallel processing, it is vital to select a memory layout that considers both sorts of graph data. For the former type, it is possible to reduce the distance to each processor core through distributed allocation of memory. In the latter, the best arrangement is to consecutively place the graph data in the core processor's local memory. We compared the threaded parallel processing performance of consecutive placement in local memory, which is the default objective, with that of memory locations (Figure 6) distributed using the `numactl` library. Table 6 summarizes the execution times for 48-threaded parallel processing for APSP of `USA-road-d.NY.gr` on a 4-way AMD Opteron 6174. n-Dijkstra yielded only a 4.64% performance improvement, while n-BFS and n/β-MLSC decreased performance by 26.82% and 16.78%, respectively. If data need to be handled in different ways, the implementation tends to fixate on one way, and this likely leads to a performance decrease.

Table 6: Default memory policy v.s. distribution of memory on a 4-way AMD Opteron 6174

	default memory policy	"`numactl --interleave=all`"	performance ratio
	CPU time (TEPS ratio)	CPU time (TEPS ratio)	
n-BFS	346.2 s (560.3 MTEPS)	473.2 s (410.0 MTEPS)	− 26.82 %
n-Dijkstra	517.2 s (375.0 MTEPS)	494.4 s (392.4 MTEPS)	+ 4.64 %
n/β-MSLC(β=32)	51.0 s (3805.6 MTEPS)	61.3 s (3167.0 MTEPS)	− 16.78 %

We clarified the relationship between the memory location and the referenced processor

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

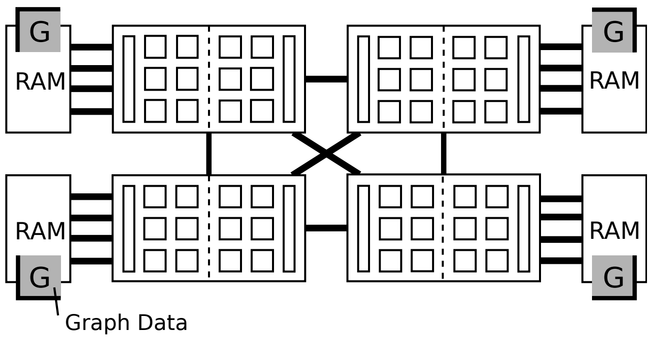


Figure 6: Data layout with "`numactl --interleave`" on a 4-way AMD Opteron 6174

core. In this experiment, the settings for the memory location and the processor core are referred to as "⟨*number of own cores*⟩ × ⟨*number of bindings*⟩*-affinity.*" Through this affinity, the graph data is copied for each binding, and the copy is configured to refer to the bound core. In other words, the number of bindings is the same as the number of graph data elements. Figure 7 shows a 48×1-affinity in which one (number of bindings = 1) graph data is shared by 48 cores. Figure 8 shows a 6×8-affinity in which eight (number of bindings = 8) graph data are shared by six cores. Here, 6×8-affinity means that eight (number of bindings = 8) graph data are shared by six cores. Note that the experiment used the `set_sched_affinity` function for allocations the cores.

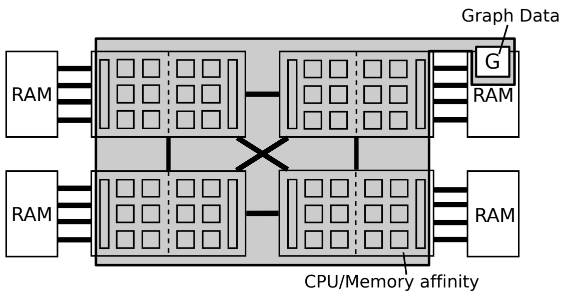


Figure 7: 48×1-affinity on a 4-way AMD Opteron 6174

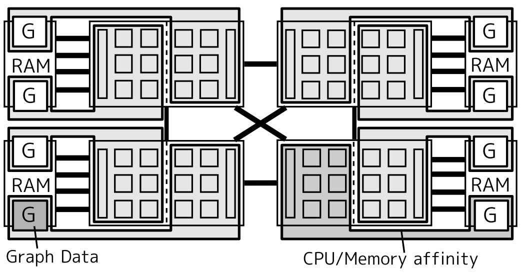


Figure 8: 6×8-affinity on a 4-way AMD Opteron 6174

Tables 7, 8, and 9 summarize the APSP execution time (speedup ratio; performance improvement from single thread), TEPS ratio, and the amount of memory used for each affinity and for each implementation of n-BFS, n-Dijkstra, and n/β-MSLC($\beta = 32$) executing `USA-road-d.NY.gr` (n =264K, m =734K). 6×8-affinity, which binds the shared local memory to a core, performed best for all implementations. 6×8-affinity outperformed 12×4-affinity because the L3 cache of the AMD Opteron 6174 was shared between each of the six cores and the cores were connected to a memory bus. On the other hand, 24×1-affinity and 48×1-affinity frequently accessed remote memory and this caused frequent computational resource request conflicts, and performance decreased owing to affinity settings. Figure 9 summarizes the best and worst cases for the TEPS ratios in Tables 7, 8, and 9. The figure shows that an appropriate affinity configuration can result in extremely high performance, and it is necessary to pay attention to discrepancies in performance owing to an erroneous configuration. With 48-threaded parallel processing, the performance difference between the best and the worst is 40.93% for n-BFS, 20.50% for n-Dijkstra, and 13.10% for n/β-MSLC. Figure 10 summarizes the parallel speedup ratios for the best case. Moreover, n-BFS, n-

Dijkstra, and n/β-MSLC($\beta = 32$) showed near-linear scaling of 35.7 times, 46.2 times, and 43.8 times compared with a single-thread implementation.

Table 7: Parallel performance (CPU time in seconds, TEPS ratio, and memory space in Mbytes) of n-BFS for APSP (`USA-road-d.NY.gr`) on a 4-way AMD Opteron 6714

		affinity	CPU time (speedup ratio)	MTEPS	MB
sequential		default	9449.8 s (× 1.0)	20.5	53.1
12 threads	worst	default	821.4 s (×11.5)	236.2	292.2
		12 × 1	810.4 s (×11.7)	239.4	292.2
	best	{1, 2} × 8	648.1 s (×14.6)	299.3	398.8
24 threads		default	426.1 s (×22.2)	455.3	553.0
	worst	24 × 1	500.2 s (×18.9)	387.8	553.0
	best	3 × 8	353.1 s (×26.8)	549.4	659.6
48 threads	worst	default	346.2 s (×27.3)	560.3	1074.6
		48 × 1	343.1 s (×27.5)	565.4	1074.6
	best	6 × 8	204.5 s (×46.2)	948.6	1181.2

Table 8: Parallel performance (CPU time in seconds, TEPS ratio, and memory space in Mbytes) of n-Dijkstra for APSP (`USA-road-d.NY.gr`) on a 4-way AMD Opteron 6714

		affinity	CPU time (speedup ratio)	MTEPS	MB
sequential		default	17915.8 s (× 1.0)	10.8	53.1
12 threads	worst	default	1804.7 s (× 9.9)	107.5	292.2
		12 × 1	1613.8 s (×11.1)	120.2	292.2
	best	{1, 2} × 8	1444.5 s (×12.4)	134.3	398.8
24 threads		default	875.1 s (×20.5)	221.7	553.0
	worst	24 × 1	910.6 s (×19.7)	213.0	553.0
	best	3 × 8	776.7 s (×23.1)	249.8	659.6
48 threads		default	517.2 s (×34.6)	375.0	553.0
	worst	48 × 1	549.6 s (×32.6)	353.0	1074.6
	best	6 × 8	411.2 s (×43.6)	471.7	1181.2

Table 9: Parallel performance (CPU time in seconds, TEPS ratio, and memory space in Mbytes) of n/β-MSLC($\beta = 32$) for APSP (`USA-road-d.NY.gr`) on a 4-way AMD Opteron 6714

		affinity	CPU time (speedup ratio)	MTEPS	MB
sequential		default	1584.4 s (× 1.0)	122.4	109.8
12 threads		default	159.2 s (×10.0)	1218.5	1036.6
	worst	12 × 1	170.7 s (× 9.2)	1136.6	1036.6
	best	{1, 2} × 8	135.6 s (×11.7)	1430.9	1143.2
24 threads		default	80.2 s (×19.8)	2418.6	2047.6
	worst	24 × 1	94.1 s (×16.8)	2060.8	2047.6
	best	3 × 8	73.6 s (×21.5)	2634.0	2154.2
48 threads	worst	default	51.1 s (×31.0)	3799.9	4069.7
		48 × 1	51.0 s (×31.1)	3805.6	4069.7
	best	6 × 8	44.4 s (×35.7)	4372.5	4176.3

Let us now summarize the appropriate number of source nodes β for n/β-MSLC. The results in Table 10 demonstrate that compared with Yanagisawa's implementation in which

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

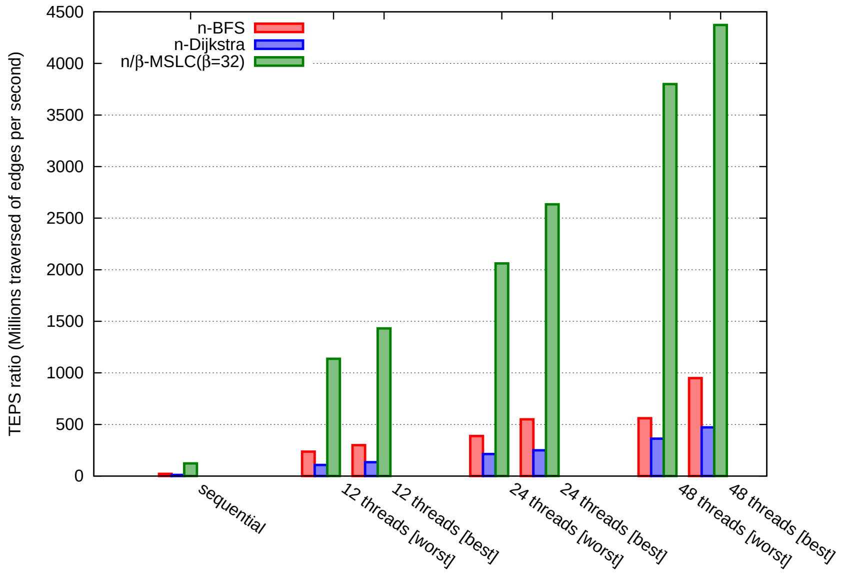


Figure 9: TEPS ratio for APSP (`USA-road-d.NY.gr`) on a 4-way AMD Opteron 6174

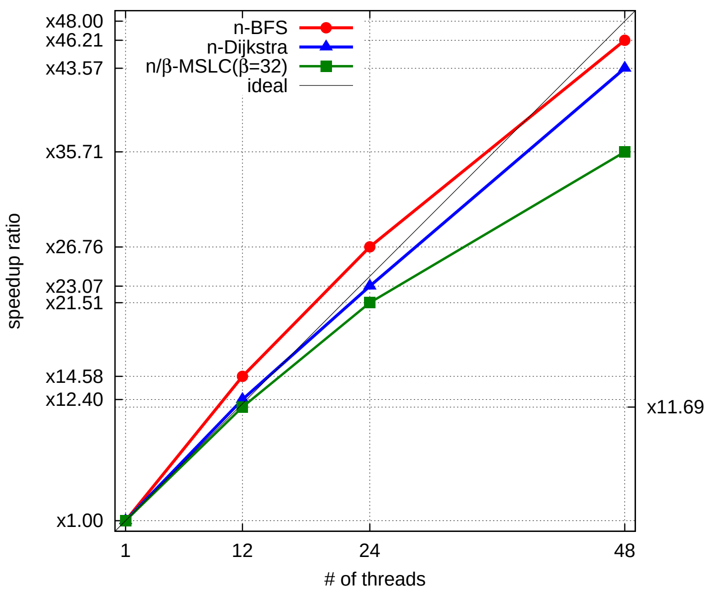


Figure 10: Speedup ratio for APSP (`USA-road-d.NY.gr`) on a 4-way AMD Opteron 6174

the appropriate of $\beta = |V_S|$ is 128, the appropriate value of β in our implementation is 32. Thus, it should be able to handle large-scale graphs. One reason for this increase is that Yanagisawa [22]'s implementation targets comparatively small graphs because of the use of a 32-bit floating number, whereas our implementation uses the 64-bit integer data type for large data. Moreover, Table 10 shows that Yanagisawa's results, for the priority queue strategy in the n/β-MSLC, PQ_min, which uses the minimum value of the temporary distances label $d(s, v)$ for each node $v \in V$ from each source node $s \in V_S$, outperforms PQ_max, which uses the maximum value, or the average value PQ_avg.

Now let us look at the variances of three implementations. Figure 11 and Table 11 effectively illustrate the computation variance. For 48-threaded parallel computation with

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

Table 10: Parallel performance (CPU time in seconds) of priority queue strategy for the n/β-MSLC multithreaded computation of APSP (`USA-road-d.NY.gr`) and the appropriate value of the source parameter β for a 4-way AMD Opteron 6174

strategy	priority queue key $k(v)$ $(v \in V)$	$\beta = 16$	$\beta = 32$	$\beta = 48$
PQ_min	$k(v) = \min\{d(s,v) \mid s \in V_S\}$	53.4 s	**44.4** s	44.6 s
PQ_max	$k(v) = \max\{d(s,v) \mid d(s,v) \neq \infty, s \in V_S\}$	59.2 s	56.4 s	60.9 s
PQ_avg	$k(v) = \sum_{s \in V_S, d(s,v) \neq \infty} d(s,v)$	801.9 s	1241.1 s	1477.1 s

the best affinity, APSP is 100 times faster with affinity setting than without it. A poor affinity causes resource conflicts and increases the variance.

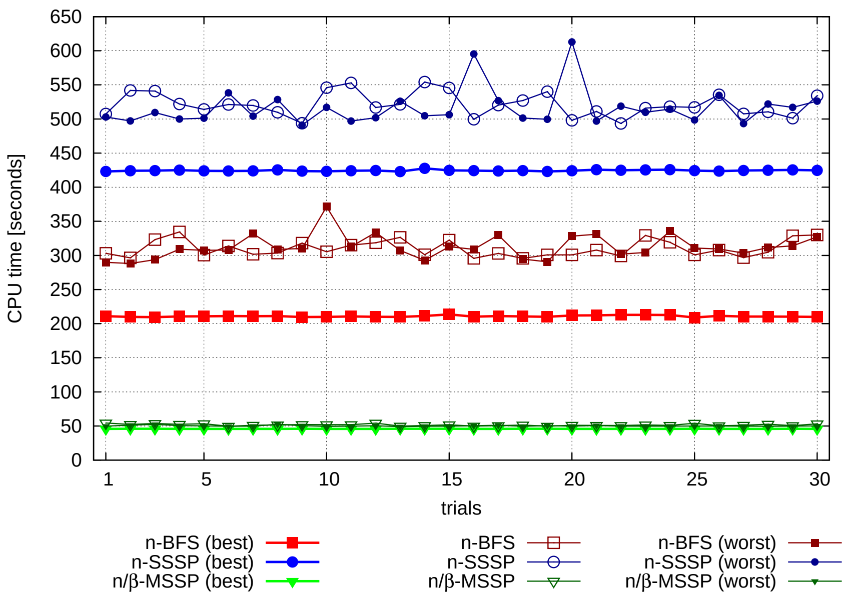


Figure 11: CPU time for APSP (`USA-road-d.NY.gr`) on a 4-way AMD Opteron 6174

Table 11: Variance of CPU time for APSP (`USA-road-d.NY.gr`) (48-threaded computation) on a 4-way AMD Opteron 6174

	best affinity	default (w/o affinity)	worst affinity
n-BFS	1.181	138.828	298.304
n-Dijkstra	0.939	287.916	711.513
n/β-MSLC	0.007	1.841	0.572

4. Numerical Experiments

We evaluated the performance of n/β-MSLC, n-Dijkstra, and n-BFS implementations by conducting numerical tests on APSP and centrality by using a 64-bit integer data type and the C language. The computer was a 4-way AMD Opteron 6174 with a 2.2 GHz frequency (12 cores $\times$ 4) and 256 GB of RAM. The OS was Fedora 15 (64 bit), and the C compiler was GCC 4.6.0. In Tables 14 and 15, the data marked with * are the estimated from the experimental results on APSP, and the data marked with * in Table 17 estimated from the centrality results. The graph instances used in this test were the road network that was the 9th DIMACS [25] target, web graph, communication network, citation network, a social network which was published by the SNAP project [26], and a synthetic R-MAT graph [8] which was generated by SSCA#2 [7] (Table 12). The results of the test succeeded in discovering the exact diameter of `web-BerkStan` (669 $\rightarrow$ 714), `web-Google` (22 $\rightarrow$ 51), `wiki-Talk` (9 $\rightarrow$ 11), and `cit-Patents` (22 $\rightarrow$ 24) (figures in brackets denote the published estimated values and exact diameters).

Table 12: Graph Instances

instance	n (SCALE(n))	m (SCALE(m))	edge factor (m/n)	diameter
9th DIMACS instance (weighted)[25]				
`USA-road-d.NY.gr`	264 346 (18.01)	733 846 (19.49)	2.78	720
`USA-road-d.LKS.gr`	2 758 119 (21.40)	6 885 658 (22.72)	2.50	4127
`USA-road-d.USA.gr`	23 947 347 (24.51)	58 333 344 (25.80)	2.44	8352*
SNAP instance (unweighted)[26]				
`web-BerkStan`	685 230 (19.39)	7 600 595 (22.86)	11.09	714
`web-Google`	875 713 (19.74)	5 105 039 (22.28)	5.83	51
`Wiki-Talk`	2 394 385 (21.19)	5 021 410 (22.26)	2.10	11
`cit-Patents`	3 774 768 (21.85)	16 518 948 (23.98)	4.38	24
`soc-LiveJournal1`	4 847 571 (22.21)	68 993 773 (26.04)	14.23	18*
SSCA#2 instance (weighted)[8]				
`RMAT-SCALE24`	16 777 216 (24.00)	134 217 728 (27.00)	8.00	21*

* estimated value

4.1. All-pairs shortest paths

Table 13 summarizes the implementations used for comparison with n-BFS, n-Dijkstra, and n/β-MSLC for APSP. In addition to the shortest path distance labels, the table includes the multiple shortest paths and number of shortest paths required for the centrality computation. n/β-MSLC only computes the distance labels. DS and LS-BFS are not compatible with the threaded parallel processing of the general computing environments.

Table 13: Implementations for APSP (*all-pairs shortest paths*) evaluation

implementation	algorithm	shortest paths	computation
NETAL (this paper)			
n-BFS	breadth-first search (BFS)	multipathBFS $\times n$	parallel
n-Dijkstra	Dijkstra's algorithm with binary heap	multipathSSSP $\times n$	parallel
n/β-MSLC	multi-source label correcting algorithm	distanceMSSP $\times n/\beta$	parallel
other implementations			
LS-BFS [4]	level-synchronized parallel BFS	singlepathBFS $\times n$	sequential
MLB [13]	Dijkstra's algorithm with multi-level buckets	singlepathSSSP $\times n$	sequential
DS [17]	Δ-stepping algorithm	distanceSSSP $\times n$	sequential

9th DIMACS instance

Figure 12 and Table 14 summarize the results for the road network with a large diameter. Our implementation substantially outperforms existing implementations. For the `USA-road-d.NY.gr` instance, n-Dijkstra was 32.7 (= 224/6.85) times faster than MLB and n/β-MSLC($\beta = 32$) was 302.7 (= 224/0.74) times faster. Similar results were obtained for `USA-road-d.LKS.gr`. Furthermore, for the largest `USA-road-d.USA.gr`, it was possible to get an shortest path distance labels in 7.75 days with n/β-MSLC. The result showed our implementation performance was 432.4 (= 3351/7.75) times that of the Δ-stepping algorithm and 228.9 (= 1774/7.75) times that of the 9th DIMACS benchmark solver.

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

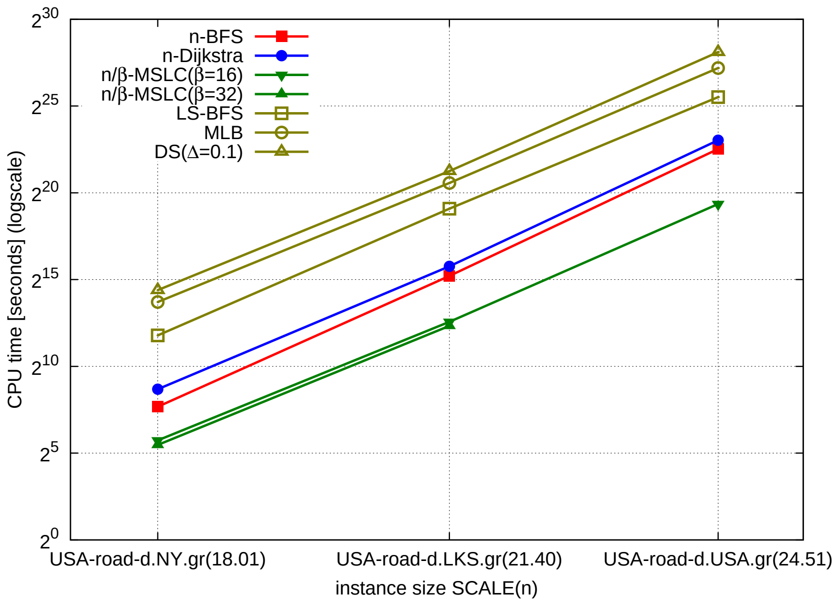


Figure 12: Computational results (CPU time in seconds (logscale)) of the APSP for the 9th DIMACS instances on a 4-way AMD Opteron 6174

Table 14: Computational results (CPU time and TEPS ratio) of the APSP for the 9th DIMACS instances on a 4-way AMD Opteron 6174

implementation	`USA-road-d.NY.gr`	`USA-road-d.LKS.gr`	`USA-road-d.USA.gr`
n-BFS	3.41 min. (0.95 G)	10.54 hours (0.50 G)	70 days*
n-Dijkstra	6.85 min. (0.47 G)	15.45 hours (0.34 G)	99 days*
n/β-MSLC ($\beta = 16$)	0.89 min. (3.64 G)	1.69 hours (3.12 G)	**7.75 days (2.09 G)**
n/β-MSLC ($\beta = 32$)	0.74 min. (4.37 G)	1.43 hours (3.70 G)	memory over
LS-BFS	59 min.*	155 hours*	555 days*
MLB	224 min.*	432 hours*	1774 days*
DS ($\Delta = 0.1$)	359 min.*	693 hours*	3351 days*

* estimated value

SNAP instance

Figure 13 and Tables 15 summarize the results for a SNAP instance with a small diameter. Our implementations were much faster and stabler than the existing methods. The following results were obtained for `cit-Patents`, 17.72 GTEPS with n-BFS, 8.78 GTEPS with n-Dijkstra, and 13.60 GTEPS with n/β-MSLC.

4.2. Centrality

Table 17 and Figure 14 summarize the performance of n-BFS and n-Dijkstra in which the four types of centrality – *closeness*, *graph*, *stress*, and *betweenness* – are simultaneously computed. Table 16 summarizes the compared implementations. Also shown is, the GraphCT [11] performance, which computes *betweenness* without considering the arc weight. The execution time for GraphCT was estimated from the results of random sampling of 4096 sources. The upper rows of the table list the centrality measurement times and TEPS ratios, while the lower rows list the ratios of the shortest path phase (sp) and update phase (up) relative to the execution time. Our implementation of n-BFS and n-Dijkstra significantly outperformed GraphCT. In the case of `web-BerkStan`, GraphCT took 18 hours, one hour n-BFS and n-Dijkstra. The performance improvement was 27.3 (= 17.99/0.66) times that of GraphCT for n-BFS and 17.1 (= 17.99/1.05) times for n-Dijkstra.

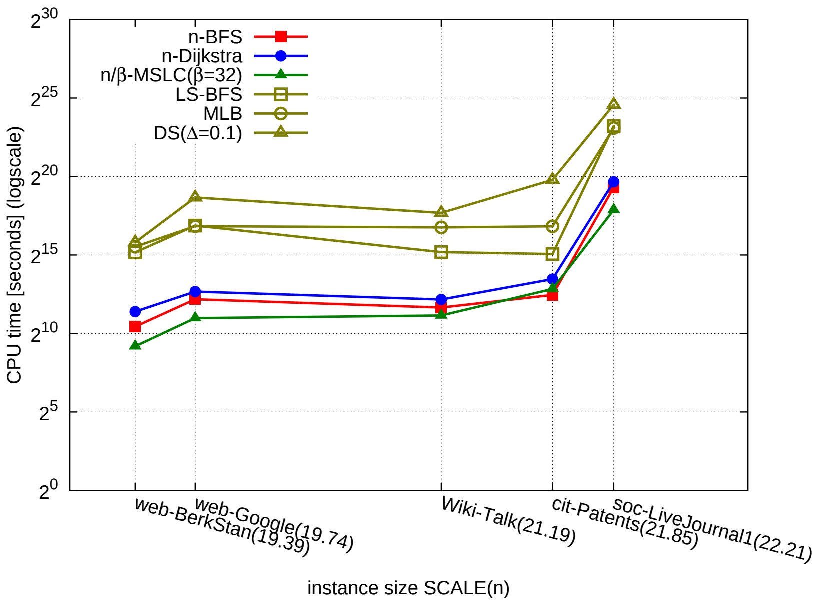


Figure 13: Computational results (CPU time in seconds (logscale)) of the APSP for SNAP instances on a 4-way AMD Opteron 6174

Table 15: Computational results (CPU time and TEPS ratio) of the APSP for SNAP instances on a 4-way AMD Opteron 6174

implementation	`web-BerkStan`	`web-Google`
n-BFS	23.10 min. (3.76 G)	77.35 min. (0.96 G)
n-Dijkstra	44.83 min. (1.94 G)	108.23 min. (0.69 G)
n/β-MSLC ($\beta = 32$)	9.69 min. (8.96 G)	33.63 min. (2.22 G)
LS-BFS	613 min.*	1998 min.*
MLB	789 min.*	1946 min.*
DS ($\Delta = 10.0$)	951 min.*	6904 min.*

implementation	`Wiki-Talk`	`cit-Patents`	`soc-LiveJournal1`
n-BFS	53.74 min. (3.73 G)	93.38 min. (17.72 G)	7.5 days*
n-Dijkstra	76.61 min. (2.62 G)	188.44 min. (8.78 G)	9.6 days*
n/β-MSLC ($\beta = 32$)	37.88 min. (5.29 G)	121.65 min. (13.60 G)	**2.78 days (1.39 G)**
LS-BFS	621 min.*	569 min.*	113.1 days*
MLB	1847 min.*	1940 min.*	103.4 days*
DS ($\Delta = 10.0$)	3500 min.*	15096 min.*	288.6 days*

* estimated value

Table 16: Implementations for centrality evaluation

implementation	graph search algorithm	centrality	computation
NETAL (this paper)			
n-BFS	BFS	C_C, C_G, C_S, C_B	parallel
n-Dijkstra	Dijkstra's algorithm with binary heap	C_C, C_G, C_S, C_B (weighted)	parallel
other implementations			
GraphCT [4]	Level-synchronized parallel BFS	C_B	sequential
SSCA#2 [18]	Level-synchronized parallel BFS	C_B	parallel

Table 17 summarizes the results of tests on SSCA#2 [7], which is a performance benchmark using *betweenness*. The graph instance was the R-MAT graph ($n = 2^{SCALE}, m = 8n, SCALE = 24$) generated by SSCA#2. To enable comparison under the same condi-

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

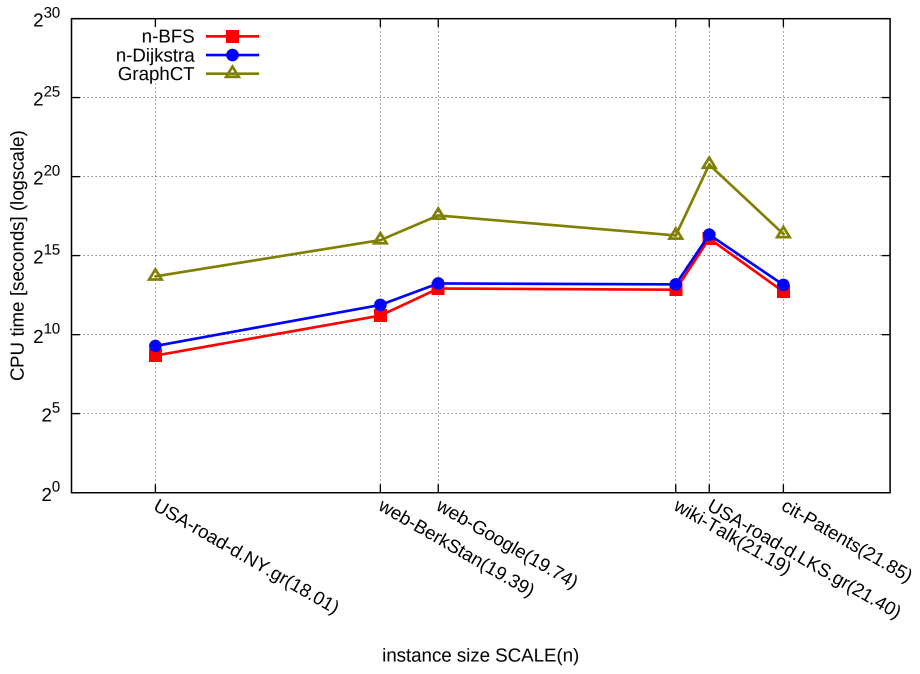


Figure 14: Computational results (CPU time in seconds(logscale)) of centrality for 9th DIMACS / SNAP instances on a 4-way AMD Opteron 6174

Table 17: Computational results (CPU time, TEPS ratio, and the ratio of the shortest path phase (sp) and update phase (up)) of centrality for 9th DIMACS / SNAP instances on a 4-way AMD Opteron 6174

instance	n-BFS (C_C, C_G, C_S, C_B)	n-Dijkstra (weighted C_C, C_G, C_S, C_B)	GraphCT (C_B)
`USA-road-d.NY.gr`	0.11 hours (0.47 GTEPS) sp: 49.50 %, up: 50.07 %	0.17 hours (0.31 GTEPS) sp: 67.66 %, up: 32.06 %	3.66 hours*
`web-BerkStan`	0.66 hours (2.21 GTEPS) sp: 60.33 %, up: 39.51 %	1.05 hours (1.38 GTEPS) sp: 71.83 %, up: 28.07 %	17.99 hours*
`web-Google`	2.15 hours (0.58 GTEPS) sp: 60.61 %, up: 39.34 %	2.70 hours (0.46 GTEPS) sp: 68.04 %, up: 31.92 %	52.97 hours*
`Wiki-Talk`	2.05 hours (1.63 GTEPS) sp: 42.15 %, up: 57.75 %	2.57 hours (1.30 GTEPS) sp: 51.93 %, up: 47.98 %	22.10 hours*
`USA-road-d.LKS.gr`	19.35 hours (0.27 GTEPS) sp: 54.52 %, up: 45.46 %	22.84 hours (0.23 GTEPS) sp: 69.48 %, up: 30.50 %	493.77 hours*
`cit-Patents`	1.87 hours (9.24 GTEPS) sp: 27.11 %, up: 72.68 %	2.52 hours (6.87 GTEPS) sp: 39.65 %, up: 60.20 %	23.61 hours*

* estimated value

tions, this test used 256 sources (a randomly sampled number), as specified by SSCA#2. GraphCT resulted in an error because the computation exceeded the range that could be handled by the used data type (32-bit integer). In a typical computing environment, n-BFS was 3.8 times faster than SSCA#2 and n-Dijkstra was 2.4 times faster (Note that SSCA#2 compatible with threaded parallel processing).

5. Conclusion and Future Work

Because of the increase in the number of cores and volume of memory requests in the current computing environment, it is now possible to handle much bigger problems than those of the

Table 18: AMD Opteron 6714 parallel performance (CPU time in seconds and TEPS ratio) of our multiple centrality computation (256-random-sampling) on R-MAT graph (SCALE24)

instance	n-BFS (C_C, C_G, C_S, C_B)	n-Dijkstra (weighted C_C, C_G, C_S, C_B)	GraphCT (C_B)	SSCA#2 (C_B)
R-MAT graph (SCALE24)	163.0 seconds 210.8 MTEPS	260.5 seconds 131.9 MTEPS	error	620.0 seconds 48.5 MTEPS

past. We believe that it is vital to make both theoretical and experimental improvements because there will be multi-hierarchical or even more complex computer memory hierarchies in the future. In particular, there is an urgent need for systematizing implementation technologies for algorithms. This study assessed a multithreaded computation method that efficiently uses computer resources for the shortest paths on a 4-way AMD Opteron 6174, a typical multicore environment. Our ultimate aim is to develop high-performance kernels to handle the shortest path problems, which are basic combinatorial optimization problems. Our multithreaded processing method configures the processor core and local memory allocation (affinity), to avoid computational resource request conflicts by considering the difference in distances between processor cores and the RAM within the NUMA architecture of the AMD Opteron 6174. n-BFS (unweighted) and n-Dijkstra (weighted), which computes the shortest paths with multiple predecessors, and n/β-MSLC (weighted), which computes the shortest path distance labels only, were all found to have exceptional performance. In the all-pairs shortest paths for `USA-road-d.NY.gr`, the shortest paths for which an arc weight was not considered were computed by n-BFS in 204.5 seconds, the all-pairs shortest paths were computed by n-Dijkstra in 411.2 seconds, and the shortest path distance labels were computed by n/β-MSLC in 44.4 seconds. In comparison with sequential computations, the parallel computations demonstrated a very high parallel speedup, 46.2 times,43.8 times, and 35.7 times, respectively for 48-thread processing. Furthermore, we succeeded in finding the all-pairs shortest path distance labels for `USA-road-d.USA.gr` by using n/β-MSLC in 7.75 days without pre-processing. For the centrality computation using n-BFS (unweighted) and n-Dijkstra (weighted), the performance was at several fold that of GraphCT and SSCA#2. In relation to `web-BerkStan`, n-BFS was 27.3 times faster than GraphCT and n-Dijkstra was 17.1 times faster. Furthermore, in relation to SSCA#2, n-BFS was 3.8 times faster and n-Dijkstra was 2.4 times faster. Our implementation simultaneously finds *closeness* C_C, *graph* C_G, *stress* C_S, and *betweenness* C_B and can select the centrality index of the graph. The applications of this study are not limited to network analysis tools using high-performance kernels; rather, our study's results can be used in a wide variety of high-performance computations with various algorithms.

Our aim is to go beyond simple graph analysis and to construct evacuation route search and transport control systems that could be used in unforeseen circumstances such as large-scale disasters. The results of this research will be useful for systems that require extremely high-performance processing of data that is regularly updated.

References

[1] U. Brandes: A faster algorithm for betweenness centrality. *Journal of Mathematical Sociology*, **25-2** (2001), 163–177.

[2] U. Brandes: On variants of shortest-path betweenness centrality and their generic computation. *Social Networks*, **30-2** (2008), 136–145.

[3] D.A. Bader, S. Kintali, K. Madduri, and M. Mihail: Approximating betewenness cen-

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

trality. *The 5th International Workshop on Algorithms and Models for the Web-Graph (WAW)* (2007), LNCS **4863**, 540–551.

[4] D.A. Bader and K. Madduri: Designing multithreaded algorithms for breadth-first search and *st*-connectivity on the Cray MTA-2. *IEEE International Conference on Parallel Processing (ICPP)* (2006), 523–530.

[5] D.A. Bader and K. Madduri: Parallel algorithms for evaluating centrality indices in real-world networks. *IEEE International Conference on Parallel Processing (ICPP)* (2006), 539–550.

[6] D.A. Bader and K. Madduri: SNAP, Small-world Network Analysis and Partitioning: an open-source parallel graph framework for the exploration of large-scale networks. *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)* (2008).

[7] D.A. Bader, K. Madduri, J. Gilbert, V. Shah, J. Kepner, T. Meuse, and A. Krishnamurthy: Designing scalable synthetic compact applications for benchmarking high productivity computing systems. *CTWatch Quarterly*, **2** (2006), 41–51.

[8] D. Chakrabarti, Y. Zhan, and C. Faloutsos: R-MAT: A recursive model for graph mining. *SIAM International Conference on Data Mining (SDM)* (2004).

[9] B.V. Cherkassky, A.V. Goldberg, and T. Radzik: Shortest paths algorithms: theory and experimental evaluation. *Mathematical Programming*, **73-2** (1996), 129–174.

[10] E.W. Dijkstra: A note on two problems in connexion with graphs. *Numerische Mathematik*, **1** (1959), 269–271.

[11] D. Ediger, K. Jiang, J. Riedy, D.A. Bader, and C. Corley: Massive social network analysis: mining twitter for social good. *IEEE International Conference on Parallel Processing (ICPP)* (2010), 583–593.

[12] L.C. Freeman: A set of measures of centrality based on betweenness. *Sociometry*, **40** (1977), 35–41.

[13] A.V. Goldberg: A simple shortest path algorithm with linear average time. *Algorithms - ESA 2001, 9th Annual European Symposium on Algorithms* (Springer Verlag, 2001), 230–241.

[14] P. Hage and F. Harary: Eccentricity and centrality in networks. *Social Networks*, **17** (1995), 57–63.

[15] P. Harish and P.J. Narayanan: Accelerating large graph algorithms on GPU using CUDA. *The 14th annual IEEE International Conference on High Performance Computing (HiPC 2007)*, LNCS **4873** (Springer Verlag, 2007), 197–208.

[16] J.L. Hennessy and D.A. Patterson: *Computer Architecture: A Quantitative Approach*, Fourth Edition, (Morgan Kaufmann, 2006).

[17] K. Madduri, D.A. Bader, J.W. Berry, and J.R. Crobak: Parallel shortest path algorithms for solving large-scale instances. *The 9th DIMACS Implementation Challenge – The Shortest Path Problem* (2006).

[18] K. Madduri, D. Ediger, K. Jiang, D.A. Bader, and D.C.-Miranda: A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets. *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)* (2009).

[19] T. Okuyama, F. Ino, and K. Hagihara: A task parallel algorithm for computing the costs of all-pairs shortest paths on CUDA-compatible GPU. *IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA)* (2008), 284–291.

[20] G. Sabidussi: The centrality index of a graph. *Psychometrika*, **31** (1966), 581–603.

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.

[21] A. Shimbel: Structural parameters of communication networks. *Bulletin of Mathematical Biophysics*, **15** (1953), 501–507.

[22] H. Yanagisawa, A multi-source label-correcting algorithm for the all-pairs shortest paths problem. *IEEE International Symposium on Parallel and Distributed Processing (IPDPS)* (2010), 1–10.

[23] Y. Yasui, K. Fujisawa, S. Sasajima, and K. Goto: Fast implementation of the Dijkstra's algorithm for the large-scale shortest path problems. *Transactions of the Operations Research Society of Japan*, **54** (2011), 58–83 (in Japanese).

[24] Graph500.org - The Graph 500 List. `http://www.graph500.org/`.

[25] The 9th DIMACS Implementation Challenge.
`http://www.dis.uniroma1.it/~challenge9/`.

[26] SNAP: Stanford Network Analysis Project. `http://snap.stanford.edu/`.

Yuichiro Yasui
Graduate School of Science and Engineering,
Chuo University, 1-13-27 Kasuga, Bunkyo-ku, Tokyo 112-8551, Japan.
E-mail: `yasui@indsys.chuo-u.ac.jp`

Copyright © by ORSJ. Unauthorized reproduction of this article is prohibited.