A Parallel Algorithm for Computing Betweenness Centrality

Guangming Tan, Dengbiao Tu, Ninghui Sun
Key Laboratory of Computer System and Architecture
Institute of Computing Technology,Chinese Academy of Science
Beijing, China
{*tgm,tudengbiao,snh*}*@ncic.ac.cn*

***Abstract*—In this paper we present a multi-grained parallel algorithm for computing betweenness centrality, which is extensively used in large-scale network analysis. Our method is based on a novel algorithmic handling of access conflicts for a CREW PRAM algorithm. We propose a proper data-processor mapping, a novel edge-numbering strategy and a new triple array data structure recording the shortest path for eliminating conflicts to access the shared memory. The algorithm requires $O(n+m)$ space and $O(\frac{nm}{p})$ (or $O(\frac{nm+n^2logn}{p})$) time for unweighted (or weighted) graphs, and it is a work-optimal CREW PRAM algorithm. On current multi-core platforms, our algorithm outperforms the previous algorithm by 2-3 times.**

***Keywords*-betweenness centrality; multi-core algorithm; multi-grained parallelism; CREW;**

I. Introduction

Large scale network analysis is one of the most important researches in a variety of applications such as social networks, transportation networks and biological networks. In most applications, graph abstractions and algorithms are naturally used to capture key features and extract interesting information. For a given real world application, network analysis and modeling, which constructs a graph for a real world dataset, is the primary step and has been paid considerable attention. Recently, a scale free graph, which degree distribution follows a power of law, has been used extensively to model the networks from some important applications including building protein interaction networks [1], [2], study of sexual networks and AIDS [3] and identifying key actors in terrorist networks [4], [5]. In these applications, betweenness centrality [6] is a popular quantitative index for the analysis of large scale complex networks. This metric is considered as a normalized centrality, which measures the control a vertex has over communication in the network, and can be used to identify key vertices in the network. High centrality indices indicate that a vertex reaches other vertices on relatively short paths, or that a vertex lies on a considerable fractions of shortest paths connecting pairs of other vertices. For example, the availability of genome-scale databases of pair-wise protein interactions data in yeast [7] has made it possible to analyze the structure of the entire protein interaction network (PIN) in light of concepts from graph theory and the study of complex networks. One way to identify functional related proteins is to cluster interaction graphs based on their topological properties. Proteins that are involved in the same cellular process or reside in the same protein complex are expected to have strong interactions with their partners. Therefore, the identification of densely connected subgraphs in PIN may reveal functional related protein modules. However, the ad hoc structural criteria used to define a module in physical networks remain somewhat arbitrary. Based on analysis of the betweenness measure, Joy et.al. [8] reported a new topological feature in the yeast PIN that is not found in randomly generated scale-free networks: the abundance of proteins characterized by high betweenness, yet low connectivity. The existence of such proteins points to the presence of modularity in the network, and suggests that these proteins may represent important connectors that link these putative modules.

A straightforward algorithm for computing betweenness centrality (BC) requires $\Theta(n^3)$ time and $\Theta(n^2)$ space, where n is the number of vertices in the graphs (or networks). Through eliminating the explicit redundant accumulation, Brandes [9] proposed a faster algorithm for computing betweenness centrality. Brandes' algorithm requires $O(n+m)$ space and runs in $O(nm)$ and $O(nm+n^2logn)$ time on unweighted and weighted graphs, where m is the number of edges. Bader et.al. [10] [11] discussed parallel algorithms for evaluating several centrality indices frequently used in complex network analysis. They proposed the first parallel implementations of BC algorithm on high-end shared memory symmetric multiprocessor and multithreaded architectures. Due to access conflicts, it is *possible* to achieve $O(\frac{nm}{p})$ and $O(\frac{nm+n^2logn}{p})$ time for unweighted and weighted graphs on PRAM models using a Fibonacci heap or pairing heap priority queue [10]. In fact, if only is the degree of each vertex 1, their parallel algorithm can achieve such time complexity (See section III). In a implementation on a shared memory machine, lock synchronization primitives are used to handle access conflicts. The practical performance is not so good as expected with respect to the overhead of handling conflicts through extra time or space requirements. In this paper we propose a new parallel BC algorithm which is adaptive to CREW PRAM. The main contribution of this paper includes:

- Our proposed algorithm eliminates access conflicts to the shared memory cells. The algorithmic techniques are highlighted by a proper data-processor mapping, a novel edge-numbering strategy and a new triple array data structure recording the shortest path.

- Under CREW PRAM we prove that our algorithm requires $O(n+m)$ space and $O(\frac{nm}{p})$ (or $O(\frac{nm+n^2logn}{p})$) time for unweighted (or weighted) graphs. It is a work-optimal CREW PRAM algorithm.
- The proposed parallel algorithm is implemented on two currently commercial multi-core platforms. We report the detailed experimental results which show our algorithm achieves 2-3 times improvement compared with the previous algorithm.

The rest of this paper is organized as follows: Section II describes the known best sequential algorithm for computing betweenness centrality. In section III we propose the algorithmic technique to handle access conflicts and present quantitative analysis of the parallel algorithm. Section 4 reports the experimental results of the implementation on multi-core architecture. Section 5 concludes this paper.

II. Computing Betweenness Centrality

Consider a graph $G = (V, E)$, where V and E is the set of vertices and edges, respectively. For an example of PIN, vertices represent actors in proteins in PIN, edges represents the interaction between proteins in PIN. The number of vertices and edges are denoted by n and m, respectively. Each edge $e \in E$ may be associated with an positive integer weight $w(e)$ ($w(e) = 1$ for unweighted graphs. *For simplicity of presentation, the following context refers to graph as unweighted one*). Define a path from $s \in V$ to $t \in V$ as an sequence of edges $< v_i, v_{i+1} >, 0 \leq i \leq l$, where l is the length of a path, $v_0 = s$ and $v_l = t$. The length of a path is the sum of the weights of its edges. We use $d(s,t)$ to denote the distance between vertices s and t. Let us denote the total number of shortest paths between vertices s and t by σ_{st}, the number passing through vertex v by $\sigma_{st}(v)$, and the fraction of shortest paths between s and t that pass through v by $\delta_{st}(v)$ (*pairwise dependency*):

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}} \tag{1}$$

Betweenness centrality (BC) of a vertex v is defined as follows:

$$bc(v) = \sum_{s \neq v \neq t \in V} \delta_{st}(v) \tag{2}$$

A straightforward way proceeds: (i) compute the length and number of shortest paths between all pairs (s,t). (ii) for each vertex v, calculate every pair-dependency $\delta_{st}(v)$ and sum them up. Obviously, this naive algorithm consumes $\Theta(n^3)$ time and $\Theta(n^2)$ space. According to Brandes' algorithm [9] we define a set of *predecessors* of a vertex v on shortest paths from s as:

$$P_s(v) = \{u \in V : \{u, v\} \in E, d_G(s,v) = d_G(s,u) + w(u,v)\} \tag{3}$$

where $d_G(s,v)$ is the distance from s to v. The computation of $P_s(v)$ can be easily integrated into breadth first search (BFS) algorithm. In order to eliminate the need of explicit redundant summation of all pair-wise dependencies, Brandes' algorithm defines the *dependency* of a source vertex $s \in V$ on a single vertex $v \in V$ as

$$\delta_s(v) = \sum_{t \in V} \delta_{st}(v) \tag{4}$$

$\delta_s(v)$ is one partial sum of $bc(v)$, thus, the betweenness centrality of a vertex v can be expressed as $bc(v) = \sum_{s \neq v \in V} \delta_s(v)$. A crucial observation of Brandes' algorithm is that the partial sum obeys a recursive relation:

$$\delta_s(v) = \sum_{w: v \in P_s(w)} \frac{\sigma_{sv}}{\sigma_{sw}} (1 + \delta_s(w)) \tag{5}$$

Therefore, the fast BC algorithm is stated as follows (See Algorithm 1). First, n BFS searches from each $s \in V$ are done in a *forward phase*. The predecessors sets $P_s(v)$ are maintained and the numbers of shortest path through the internal vertex w are recorded during these computations. Next, for every $s \in V$, both the number of shortest paths through a vertex w stored in $\sigma(w)$ and predecessor sets along the paths are used to compute the dependencies $\delta_s(v)$ for all other $v \in V$ in a *backtrace phase*. In the end, the sum of all dependency values of a vertex v is computed according to the recursive relation. This fast algorithm needs $O(n + m)$ space and $O(nm)$ time.

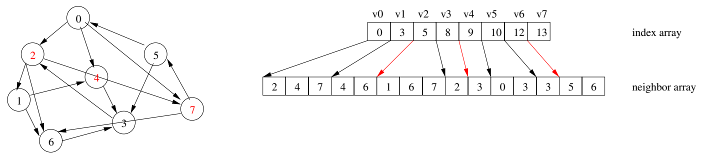


Figure 1. Adjacency array of a graph.

Figure 2 demonstrates an instance starting with vertex v_0 in Figure 1. Both *step 1* and *step 2* generate a BFS tree (*predecessor*) and record its information (σ and d). Then according to equation II, the values of δ is accumulated along the BFS tree from *step 3* to *step 5*. At this time, the five steps compute the partial betweenness centrality values of all vertices. After the algorithm performs similar steps from all other vertices we get the final accumulated results. Here a space-efficient data structure for sparse graph G is an adjacency array. Figure 1 shows an example of an adjacency array, which is composed of index array and a neighbor array. In fact, the predecessor set P recording the trace of BFS tree is also stored in another adjacency array. Other parameters d, δ, σ, and the measure bc are represented as

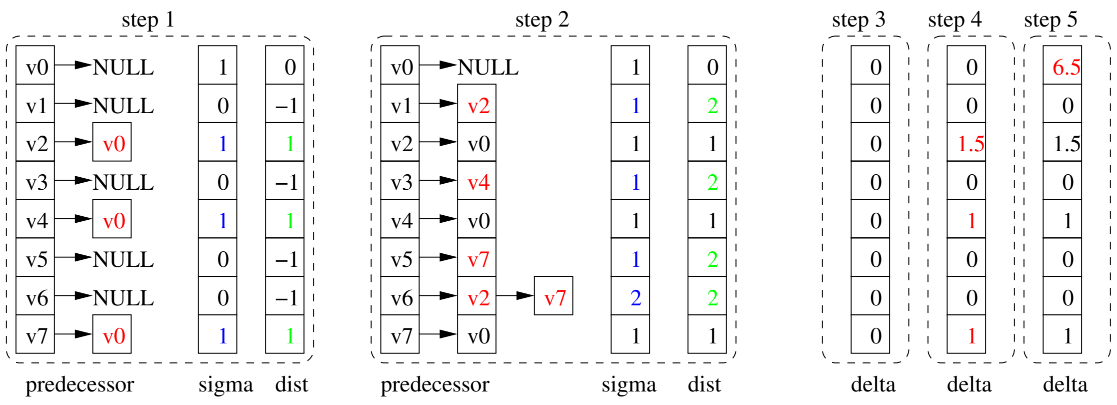


Figure 2. A demonstration of BC algorithm.

Algorithm 1 The sequential BC algorithm.

```
for each s in V do {
 Q <— empty queue;
  for each w in V do {
    P[w] <— empty list;
    sig[w] = 0; sig[s] = 1;
    d[w] = −1; d[s] = 0;
  }
  enqueue s —> Q;
  while Q not empty do {
    v <— head of Q;
    for each neighbor w of v do {
    //w found for the first time?
      if d[w] < 0 then {
       enqueue w —> Q;
       d[w] = d[v] + 1;
       }
       // shortest path to w via v?
       if d[w] = d[v] + 1 then {
        sig[w] = sig[w] + sig[v];
        append v —> P[w];
       }
    }
  }
  S <— Q; //emulate queue as a stack
  while S not empty do {
   pop w <— S;
   for each v in P[w] do {
    delta[v] += (sig[w]/sig[w])*(1+delta[w]);
   }
   if w != s then {
    bc[w] = bc[w] + delta[w];
   }
  }
 }
```

linear arrays. However, the references to the three linear arrays are very dependent on that to the two adjacency arrays.

III. Parallel BC Algorithms

In this section, we present our study on parallelizing BC algorithm. First, BC algorithm itself requires nontrivial tradeoff between different levels of parallelism, the concurrency may be exploited at three levels of granularity:

- *Coarse-grained:* The computation starting from each source vertex can be considered as a task (i.e. the all five steps in Figure 2), the algorithm needs $n = |V|$ tasks to compute partial values, which can proceed in parallel, If p processors are used, p copies of data structures are required ($O(p(n+m)$ space). In a real world, this space usage for a large scale graph easily exceeds the available physical memory on conventional parallel computers.
- *Medium-grained:* The forward phase constructs a BFS tree level by level. It explores all neighbors of each vertex in current level and the neighboring vertices which are visited in the first time are selected as the ones in the next level. Again, one exploration of a vertex can be considered as one task, thus, all tasks in one level could proceed totally in parallel if there was no shared neighbor between any two vertices. Otherwise, memory access conflicts occur and a synchronization mechanism is required to exploit this granularity of parallelism.
- *Fine-grained:* The task of exploring the neighbors of a vertex itself can also be parallelized. The amount of available parallelism depends on the degree of a vertex. Although only few vertices which have high degrees in scale free sparse graph, the fine-grained parallelism is conducive to load balance.

Due to the high requirement on memory space and embarrassingly parallelism, we do not discuss the coarse-grained parallelism. In the previous work [10] [11], the parallel algorithms only exploit either medium- or fine-grained parallelism. Note that the algorithm traverse the graph along the BFS tree level-by-level and each edge is visited by only one time, thus it traverses m edges at most. If there exist edges sharing the same vertex, the traverse along these edges is serialized, which hinders parallelism from being exploited. We have the following lemma:

Lemma 3.1: Without taking into consideration access conflicts, one BFS traversal runs $O(\frac{m}{p})$ time when either of the following two cases is true:
(i) hybrid medium- and fine-grained parallelism is exploited.
(ii) the degree of each vertex is one when either medium-grained or fine-grained parallelism is exploited.

Proof: Assume, the length of BFS tree is L at the level i of BFS, there are k_i vertices of leaves and the degree of the vertices are $dr_{i1}, ..., dr_{ik_i}$. Thus, there are at most $\sum_{j=1}^{k_i} dr_{ij}$ edges to be traversed.
(i) In the case of hybrid medium- and fine-grained parallelism, all unvisited edges are assigned to p processor using any partition. That is, it consumes $T_i = O(\frac{\sum_{j=1}^{k_i} dr_{ij}}{p})$ time to traverse the edges at level i. Because each edge is visited by only one time, the algorithm completes the traversal in time of T:

$$T = \sum_{l=1}^{L} T_i \leq O(\frac{m}{p})$$

(ii) In the case of medium- or fine-grained parallelism, there exist sequential work. The medium-grained parallelism has the sequential work of $T_i = O(max_{j=1}^{k_i} dr_{ij})$. Similarly, the fine-grained parallelism performs the sequential work of $T_i = O(\sum_{j=1}^{k_i} \frac{dr_{ij}}{p})$. Then the total time $T = O(f(dr) \times \frac{m}{p})$, where $f(dr)$ is a constant determined by the degree distribution. In fact, when the degree of each vertex is one, the parallel algorithm naturally exploits both granularities of parallelism. ■

Thus, an important problem to be addressed is how to avoid conflicts for exploiting both granularities of parallelism.

A. Forward Phase

Let's refer to the process of visiting the neighboring vertices of a vertex in queue as an *extension* operation (line 10-19 in Algorithm 1). During the forward phase performing BFS, except for the potential conflict at the shared neighboring vertices, both enqueuing new vertices and appending predecessors require a synchronization mechanism to maintain their consistency on a shared memory. Assume that there are p processors, we partition the set of vertices V into disjoint sub-sets V_i, $0 \leq i < p$. Each sub-set is only assigned to one processor, and processor i is only allowed to perform extension operations of the vertices it owns (say V_i). Thus, when a processor explores the vertices at a level, it identifies those who belong to its own set, then performs extension. For example in Figure 1 assume that the vertices are partitioned into: $V_0 = \{v_0, v_1\}$, $V_1 = \{v_2, v_3\}$,$V_2 = \{v_4, v_5\}$,$V_3 = \{v_6, v_7\}$, and there are 4 processors. At the *step 2* in Figure 2, the extension operation of v_1 is assigned to processor p_0, v_3 to p_1, v_5 to p_2, v_6 to p_3, respectively. Although v_6 is the shared neighboring vertex of both v_2 and v_7, the two extension operations (modifying $d[6], sig[6]$ and $P[6]$) of v_6 are assigned to the same processor so that access conflicts between different processors are eliminated.

If the neighboring vertices are visited for the first time, they are enqueued. We partition the shared queue Q into p sub-queues Q_i, $0 \leq i < p$. Although an extension may identify multiple unvisited vertices to be added to the queues, each processor i only selects their own unvisited neighboring vertices $w \in V_i$, then enqueues them onto Q_i. A fact is that Q_i is determined by V_i, that is:

Fact 3.2: Give a disjoint partition of vertices sets: $\{V_1, ..., V_p\}$, we have $V_i \equiv Q_i$, $0 \leq i < p$ at the end of forward phase.

Proof: Let's denote the neighboring vertices of a vertex $v \in V_i$ to be W. At BFS level k all processors perform extension operations on W. Processor i only selects the vertex w who belongs to its own set V_i to extend, that is $w \in W \cap V_i$. If w is visited at the first time, it is inserted to Q_i. Therefore, $Q_i \subseteq V_i$. On the other hand, the graph is assumed to be connected, then every vertex must be visited for at least one time. Thus, $V_i \subseteq Q_i$. So we have $Q_i \equiv V_i$. ■

Thus, the following two claims is straightforward:

Claim 3.3: Give a disjoint partition of vertices sets: $\{V_1, ..., V_p\}$, the length of each Q_i, $0 \leq i < p$ is statically determined at the beginning of forward phase, and $\sum_{i=1}^{p} |Q_i| = |V|$

Claim 3.4: Give a disjoint partition of vertices sets: $\{V_1, ..., V_p\}$, $Q_i \cap Q_j = \phi$, $0 \leq i \neq j < p$

Combining the above observations, we conclude that:

Lemma 3.5: Give a disjoint partition of vertices sets: $\{V_1, ..., V_p\}$, our strategy for splitting the shared queue does not result in any extra memory space.

Again for the previous example, at *step 1* vertices v_2, v_4, v_7 need to be added to queue. Our algorithm stores them into $Q_1 = \{v_2\}, Q_2 = \{v_4\}, Q_3 = \{v_7\}$. Then at step 2 (the end of forward phase) we have $Q_1 = \{v_0, v_1\}, Q_1 = \{v_2, v_3\}, Q_2 = \{v_4, v_5\}, Q_3 = \{v_7, v_6\}$.

In backtrace phase the algorithm calculates BC values level by level from leaves to root along the BFS tree. Because the data structure of predecessor sets does not record *level* information of each vertex, the sequential algorithm easily emulates the queue as a stack to pop level by level. In [10] [11] the parallel algorithm uses an extra space to record the number of vertices in each level. Because the shared queue is split into p parts, a direct use of their idea leads to p times of more space. Therefore, we develop a new data structure to record the shortest path.

In the sequential algorithm, the predecessor set represents the shortest path. Each vertex keep the trace of its parents. Because multiple vertices may share a parent, synchronization is required to insert a parent to the set. Essentially, the predecessor set keeps track of path used in backtrace phase, that is, the edges of BFS tree are stored. In order to facilitate backtrace accumulation, it is necessary to record which level

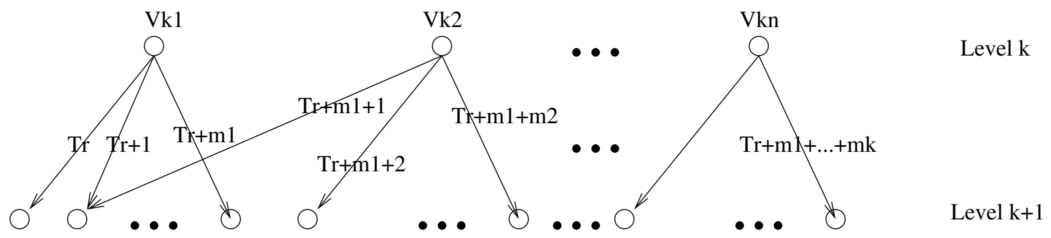


Figure 4. Assign an unique number to each edge

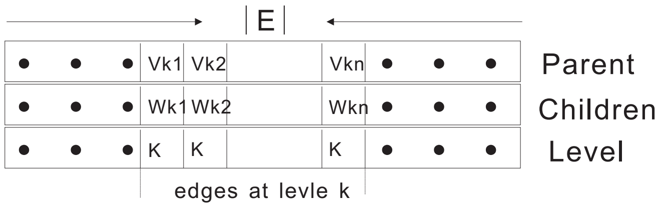


Figure 3. A triple array shared by all processor

an edge belongs to. The new data structure is a triple array, each element of which is $< parent, children, level >$ (See the left picture in Figure 3). Given an edge $< v, w >$ at level k, we set an entry of the triple array as $< v, w, k >$. For example in Figure 2, the *step 1* is traversing the edges in level 1, thus the corresponding entries in triple array contain $< v_0, v_2, 1 >, < v_0, v_4, 1 >, < v_0, v_7, 1 >$. Like the adjacency array, the triple array is shared by all processes and requires the handling of access conflicts.

Observe that if each edge is assigned with a unique number, this number can be used as identifier for inserting into the triple array. In order to find an entry which a processor should set, we number the traversed edges. Assume that at level k the parents are $\{v_{k1}, v_{k2}, ..., v_{kn}\}$, the neighboring vertices of each parent are $\{w_{11}, w_{12}, .., w_{1m_1}\}$, $\{w_{21}, w_{22}, .., w_{2m_2}\}$, ..., $\{w_{n1}, w_{n2}, .., w_{nm_n}\}$, respectively, and the number of traversed edges before level k is Tr. Each processor numbers the edges as $Tr + 1, Tr + 2, ..., Tr + m_1, Tr + m_1 + 1, Tr + m_1 + 2, ..., Tr + m_1 + m_2, ..., Tr + \sum_{i=1}^{n} m_i$.

Note that each processor calculates a independent counter Tr (line 4 in Algorithm 2). If an edge (v, w) is visited for the first time (line 6 in Algorithm 2) and lies in the shortest path (line 10 in Algorithm 2), this edge is inserted into position Tr of the triple array. As shown in line 5-16 of Algorithm 2, all operations are performed by processor i, where $w \in V_i$. Even if w is shared by two edges, two different numbers (i.e. $Tr + 1$ and $Tr + m_1 + 1$) in Figure 4 are associated with them, respectively. Thus, there is no access conflict for the triple array. We use the example in Figure 1 to illustrate how the algorithm works. In the graph there are 15 edges, which is numbered from 1 to 15. Assume that there are $p = 4$ processors. Then, the parallel algorithm proceeds as follows

Algorithm 2 Forward(): The parallel algorithm for forward phase

```
1  while each Q[i] is not changed {
2   for each v in Q[i] do in parallel  {
3    for each neighbors w of v do in parallel {
4      Tr++;
5      if w in V[tid] {//tid is processor's id
6          if (d[w] < 0) {
7                enqueue w ---> Q[tid];
8                d[w] = d[v] + 1;
9          }
10         if (d[w] == d[v] + 1) {
11               sig[w] = sig[w] +  sig[v];
12               PTriple[Tr].parent = v;
13               PTriple[Tr].child = w;
14               PTriple[Tr].level = L;
15          }
16      }
17   }
18  }
19  L++; //next level
20  }
```

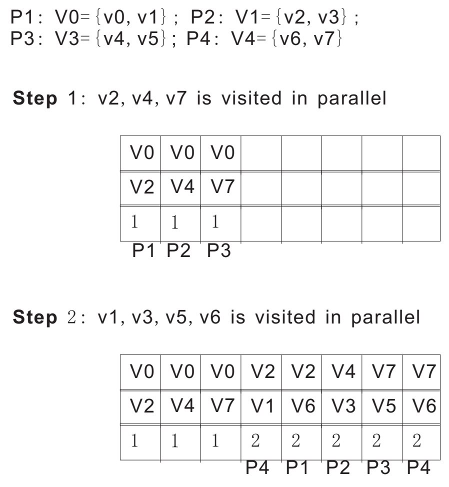
(See Figure 5):

- *step 1:* processor p_2 stores triple $< v_0, v_2, 1 >$ to entry 1, p_3 stores $< v_0, v_4, 1 >$ to entry 2, p_4 stores $< v_0, v_7, 1 >$ to entry 3.
- *step 2:* processor p_1 stores triple $< v_2, v_1, 2 >$ to entry 4, p_4 stores $< v_2, v_6, 2 >$ to entry 5, p_2 stores $< v_4, v_3, 2 >$ to entry 6, p_3 stores $< v_7, v_5, 2 >$ to entry 7, p_4 stores triple $< v_7, v_6, 8 >$ to entry 4.

Obviously, the size of tuple array is bound by the number of edges. Because BFS tree is a sub-graph, some entries of the tuple array is wasted. However, the new algorithm consecutively numbers the edges level by level, all edges are stored in a contiguously region, which could take advantage of locality in a cache-based architecture.

B. Backtrace Phase

The algorithm of backtrace phase is straightforward. After the triple array is constructed, backtrace along the BFS tree accumulate the partial BC values. The accumulation is finished through scanning the triple array level by level in a backward way. Again we use the same idea to avoid access



P1: V0={v0, v1} ; P2: V1={v2, v3} ;
P3: V3={v4, v5}; P4: V4={v6, v7}

Step 1: v2, v4, v7 is visited in parallel

V0	V0	V0					
V2	V4	V7					
1	1	1					
P1	P2	P3					

Step 2: v1, v3, v5, v6 is visited in parallel

V0	V0	V0	V2	V2	V4	V7	V7
V2	V4	V7	V1	V6	V3	V5	V6
1	1	1	2	2	2	2	2
			P4	P1	P2	P3	P4

Figure 5. An example showing how the triple array works

conflict, that is, processor i accumulates the value of parent $v \in V_i$. Algorithm 3 describes pesudocodes of the proposed parallel algorithm.

Algorithm 3 Backtrace(): The parallel algorithm for backtrace phase

```
while L >  0 { // level-by-level
 for each edge (v, w) of L do in parallel{
   if v in V[tid] {
    delta[v] += (sig[v]/sig[w])(1+delta[w]);
   }
 }
 L--;
}
```

Algorithm 4 BC(): The algorithm for computing betweenness centrality.

```
   for each s in V do {
     //starting from vertex s
     Foward();
     Backtrace();
     for each v in V do {
          bc[v] = bc[v] + delta[v];
     }
   }
```

C. Overall Analysis

The final BC Algorithm 4 is composed of Algorithm 2 and 3. We give a quantitative analysis in terms of time and space complexity under PRAM models.

Theorem 3.6: Given a graph $G = (V, E), |V| = n, |E| = m$, the proposed parallel algorithm needs $O(n + m)$ space complexity.

Proof: As the same with the sequential algorithm, the graph is represented as a adjacency array, which needs $O(n + m)$ space. The arrays of distance (d), dependency (σ, δ) and BC values (bc) need $O(4n)$ space in total. According to Lemma 3.5, the parallel algorithm also needs $O(n)$ space to be used as a queue. There is a difference between the sequential and parallel algorithms for space of predecessor set. The sequential algorithm (and the previous parallel algorithm in [10] [11]) adopts the adjacency array, which requiring $O(n+m)$ space. Our parallel algorithm uses a triple array, which needs O(3m) space. Therefore, the space complexity of the parallel algorithm is yet O(n+m). ■

Theorem 3.7: Given a graph $G = (V, E), |V| = n, |E| = m$, the proposed parallel algorithm needs $O(\frac{nm}{p})$ (or $O(\frac{nm+n^2 logn}{p})$ for weighted graphs) time complexity.

Proof: Algorithm 2 successfully eliminates the access conflicts for all shared memory cells. According to Lemma 3.1, Algorithm 2 runs in time of $T_f = O(\frac{m}{p})$. Algorithm 3 traverses the triple array in a backward way. Since the length of the triple array is less than m and no conflicts occurs, the running time of Algorithm 3 is $T_b < O(\frac{m}{p})$. Algorithm 2 and 3 are executed in a serial way for n times, therefore, the total time complexity of the proposed BC algorithm is $O(\frac{nm}{p})$ ■

Note that the sequential algorithm runs in $O(nm)$ time. Based on the proof of the above two theorems, we conclude that:

Theorem 3.8: Given a graph $G = (V, E), |V| = n, |E| = m$, the proposed parallel algorithm is work-optimal.

IV. Performance Evaluation

In this section we describe the experimental methodology and results. We report the comparison of execution time on multi-core platforms. Due to the importance of computing betweenness centrality, as one of HPCS benchmarks [12], SSCA2 (Scalable Synthetic Compact Application #2) project [11] makes a specification of implementing BC algorithm for evaluating high performance architecture research. The graph generator in SSCA2 is based on the Recursive MATrix (R-MAT) scale-free graph generation algorithm [13]. SSCA2 package includes an OpenMP implementation of parallel BC algorithms.

A. Experimental Platforms

We examine the parallel BC algorithm on two 2-ways 4-cores SMPs which are composed by the leading commercial multi-core processors – Intel Clovertown and AMD Barcelona, respectively. Intel Colvertown is a quad-core processors integrating two dual-core Xeon chips onto a single multi-chip module. Each Clovertown core includes a 32KB, 8-way L1 cache, and each chip has a shared 4MB, 16-way L2 cache. The full system has 74.67GFlops/s peak performance at memory bandwidth of 21.3GB/s. AMD Barcelona is a quad-core processor which features

Table I
THE EXECUTION TIME (SECONDS) OF THE PARALLEL PROGRAMS ON MULTI-CORE PLATFORMS

S		18			20			21			22		
#cores		2	4	8	2	4	8	2	4	8	2	4	8
Bacelona	SSCA2	482	269	304	2371	1280	1184	5180	2696	2370	11422	5930	4723
	New	169	134	107	954	708	590	2132	1557	1239	4637	3398	2602
Clovertown	SSCA2	363	233	238	1808	1049	922	3753	2177	1852	8141	4663	3688
	New	123	100	72	710	579	428	1569	1320	1011	3383	2863	2258

a highly integrated design with all four cores on a single die with shared resources, which is in contrast to package-level integration to get two separate dual-core dies in Intel Clovertown. Each core has its own private 128KB L1 cache and 512KB L2 cache. All four cores share a common L3 cache that is at least 2MB in size. The full system provides an aggregate memory bandwidth of 21.4GB/s and 54.4GFlops/s peak performance. On multi-core platforms we implement a new BC kernel in SSCA2 package using the proposed parallel algorithm with OpenMP. The OpenMP program is compiled by PGI compiler (Version 6.2).

B. Experimental Results

The performance is measured by execution time and scalability of two parallel algorithms. Table I summarizes the execution time on Intel Clovertown and AMD Barcelona. The *SSCA2* represents the parallel algorithm using lock for access conflicts in HPCS SSCA2, and the *New* represents our proposed parallel algorithm without lock synchronization. The *S* denotes the number of vertices is 2^S. Compared with the original SSCA2, the new parallel algorithm without lock synchronization reduces the execution time by 2-3 times.

However, there is a big gap between the achieved performance and the peak performance on the multi-core processors. Through a deep insight of algorithmic analysis, we observe that observe that BC algorithm exhibits two remarkable irregular features, which hinder the program from better performance:

1) *dynamically non-contiguous memory access pattern.*In Fig. 6 we track the accessed entries in *neighbor array* during BFS. The y-axis in Fig. 6 represents the first entry of neighboring vertices of the vertex in work queue. Because the neighboring vertices of one vertex is stored in contiguous region, the non-contiguous accesses appear between two different regions of neighboring vertices. For example in Fig. 1, at time t it accesses entry 5, at time $t+1$ accesses entry 9, then accesses entry 13 at time $t+2$. Again in Fig. 1, the stride between entry 5 and 9 is 1, between entry 9 and 13 is 3. Obviously, the random memory access pattern and highly variant strides make prefetching and speculation on current architecture impractical.
2) *low arithmetic intensity.* Looking at the pesudocode in Algorithm 1, an extension of one vertex needs only two arithmetic (float point addition) operations,

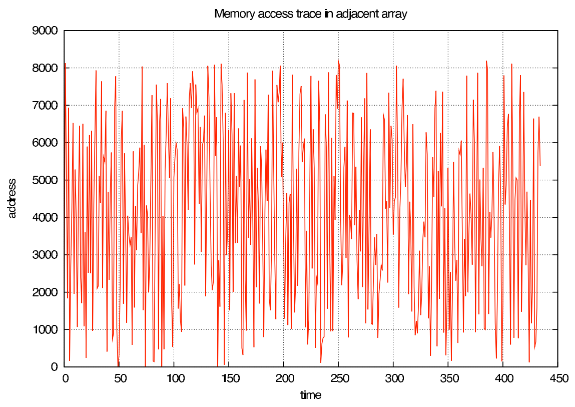


Figure 6. Trace of memory access in adjacent array

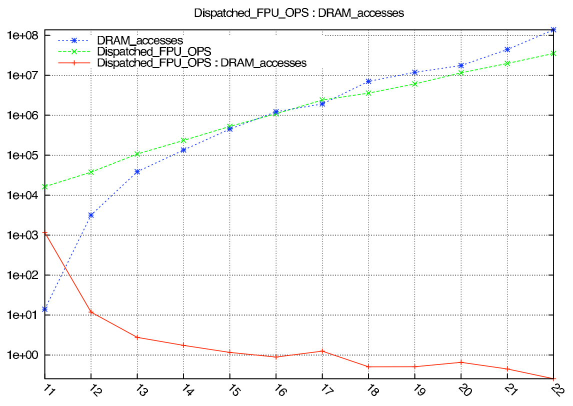


Figure 7. The ratio of arithmetic operations to memory access

about ten memory operations. If the operands are 32-bits, the arithmetic intensity is $2 : (10 \times 4) = 1 : 20$. Thus, we estimate that the required peak bandwidths on Clovertown and Barcelona are 1493GB/s and 1090GB/s, respectively. Obviously, the low bandwidth on Clovertown starves the floating execution. As shown in Fig. 7, for small work set the number of float operations is far more compared to the number of memory access times, about 1000 times or more. Although current multi-core designs do not resort to increase the speed of single core any more, the number of cores in a chip is increasing for a higher arithmetic performance. For traditional scientific computing applications with high arithmetic intensity and high parallelism, they naturally benefit from multi-core architectures. In order to further improve the performance of memory bound programs like BC algorithm, the future work will pay more attention to reduce the memory access latency leveraging by the massive parallel thread units.

V. Conclusion

In this paper, we present an efficient algorithm to compute betweenness centrality for large-scale network analysis on the CREW PRAM model. The algorithm requires $O(\frac{nm}{p})$ (or $O(\frac{nm+n^2logn}{p})$ for weighted graphs) time and $O(n+m)$ space. Thus, the algorithm is work-optimal. The implementations on current multi-core architectures show the algorithm achieve better practical performance than the previous SMP algorithm.

Acknowledgment

We would like to thank all reviewers for improving this paper. This research is supported by National Natural Science Foundation of China (No.60803030, No.60633040) and Chinese Academy of Sciences (No.KGCX1-YW-13).

References

[1] A. del Sol, H. Fujihashi, and P. O'Meara, "Topology of small-world networks of protein–protein complex structures," *Bioinformatics*, vol. 21, no. 8, pp. 1311–1315, 2005.

[2] H. Jeong, S. P. Mason, A.-L. Barabasi, and Z. N. Oltvai, "Lethality and centrality in protein networks," *Nature*, vol. 411, p. 41, 2001.

[3] F. Liljeros, C. R. Edling, L. A. N. Amaral, H. E. Stanley, and Y. Aberg, "The web of human sexual contacts," *Nature*, vol. 411, p. 907, 2001.

[4] V. E. Krebs, "Mapping networks of terrorist cells," *Connections*, vol. 24, no. 3, pp. 43–52, 2002.

[5] T. Coffman, S. Greenblatt, and S. Marcus, "Graph-based technologies for intelligence analysis," *Communications of the ACM*, vol. 47, no. 3, pp. 45–47, 2004.

[6] L. C. Freeman, "A set of measures of centrality based on betweenness," *Sociomtry*, vol. 40, no. 1, pp. 35–41, 1977.

[7] I. Xenarios and D. Eisenberg, "Protein interaction databases," *Curr Opin Biotechnol*, vol. 12, no. 4, pp. 334–339, 2001.

[8] M. Joy, A. Brock, D. Ingber, and S. Huang, "High-betweenness proteins in the yeast protein interaction network," *J. Biomed Biotechnol*, vol. 2, pp. 96–103, 2005.

[9] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of Mathematical Socialogy*, vol. 25, no. 2, pp. 163–177, 2001.

[10] D. A. Bader and K. Madduri, "Parallel algorithms for evaluating centrality indices in real-world networks," in *The 35th International Conference on Parallel Processing (ICPP 2006)*, 2006.

[11] D. A. Bader, "Hpcs scalable synthetic compact applications 2 graph analysis," www.highproductivity.org/SSCABmks.htm, 2006.

[12] "www.highproductivity.org."

[13] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *In SDM*, 2004.