

# Characterizing Betweenness Centrality Algorithm on Multi-core Architectures

Dengbiao Tu<sup>\*,†</sup>, Guangming Tan<sup>\*</sup>

<sup>\*</sup> Key Laboratory of Computer System and Architecture, Institute of Computing Technology,  
Chinese Academy of Science

<sup>†</sup> Graduate University, Chinese Academy of Sciences

Email: {tudengbiao,tgm}@ncic.ac.cn

**Abstract**—This paper presents an in-depth analysis of characterization for an irregular application – computing betweenness centrality (BC) – on multi-core architectures. BC algorithm is widely used in large scale graph analysis applications, which play an increasingly important role in high performance computing community. Through a joint study of architecture and application, we find that dynamically non-contiguous memory access, unstructured parallelism and low arithmetic intensity in BC program pose an obstacle to an efficient execution on parallel architectures. The experimental results report a comparison between Intel Clovertown and Sun Niagara1 for running such irregular program. Finally, several implications on multi-core architecture and programming are proposed.

## I. INTRODUCTION

Emerging microprocessor chip technology unveils a new generation of multi-core chip architectures that may contain 100 to 1,000 processing cores. Computer architects, system software designers and application scientists are realizing that they must work closely together to investigate how to exploit the computational power of such new many-core architecture to improve performance and scalability of large-scale scientific applications. At a high level there are two kinds of applications—“regular applications” where data access and control flow follow regular and (statically) predictable patterns, and “irregular applications” where data access and control flow have statically (and often even dynamically) unpredictable patterns. Analysis and optimization of such irregular applications are notoriously difficult. With the advent of massive many-core architectures, such as Intel Tera-scale [?], IBM Cyclops64 [?], GPU [?] and ClearSpeed [?] that contain tens or even hundreds of on-chip cores, it is extremely important to tackle the difficult problem of optimizing and scaling irregular applications. On-chip memory hierarchy, limited on-chip memory per core, and other features in such architectures make the problem even more difficult. Researchers are realizing that for many-core architectures the problem of scaling and optimizing irregular applications have to be done at different phases, including algorithmic changes and improvements to take advantage of the many-core architecture features [?], [?]. Many irregular applications are often implemented using pointer data structures such as graphs and queues and recursive control flow to traverse and manipulate such pointer data structures. It is difficult and often impossible to capture the data access patterns at compilation time. For architectures that

support memory hierarchy, unpredictable data access patterns often lead to higher off-chip memory access latency, which in turn can degrade the performance and scalability of irregular applications.

Computing betweenness centrality (BC) [?] in graph analysis is a good example of such irregular problems. 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 construct a graph for a real world dataset, is the primary step and has been paid considerable attention. Recently, a scale free graph, where the degree distribution follows a power of law, has been used extensively to model the networks from some important applications including building protein interaction networks [?], [?], study of sexual networks and AIDS [?] and identifying key actors in terrorist networks [?], [?]. In these applications, betweenness centrality [?] is a popular quantitative index for the analysis of large scale complex networks. It 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 can reach other vertices on relatively short paths, or that a vertex lies on a considerable fractions of shortest paths connecting pairs of other vertices. Brandes’ algorithm [?] is one of fast algorithms for computing BC. In this paper, we refer to BC algorithm as one proposed by Brandes [?]. In general, BC algorithm calculates the centrality through two steps: BFS (breadth first search) traversal and backtrace accumulation. Due to scale-free [?] sparse graph traversal in these important applications, BC algorithm exhibits *dynamically non-contiguous memory access* and *unstructured parallelism*. Another distinct characteristic is *low arithmetic intensity* – the ratio between arithmetic operations and memory operations, which obviously forces programmers to expose an adequate amount of parallelism to the underlying many-core architecture within an application, instead of using higher speed processor.

The rest of the paper is organized as follows: In section 2, we describe betweenness centrality (BC) algorithm. In section 3, we introduce the experimental platforms and methodology.

Section 4 discusses the irregular characteristics of BC algorithm. Section 5 draws implications on many-core architecture and programming. Finally, section 6 concludes this paper.

## II. BRIEF INTRODUCTION OF BC ALGORITHM

In this section, we will briefly describe the definition of BC metric and the algorithm to calculate it (for the detailed algorithm, refer to [?]), then examine the irregular characteristics.

### A. Definition of BC metric

Consider a graph  $G = (V, E)$ , where  $V$  and  $E$  is the set of vertices and edges, respectively. 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  $w(e)$ . Define a path from  $s \in V$  to  $t \in V$  as an sequence of edges  $\langle v_i, v_{i+1} \rangle$ , where  $v_0 = s$  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  $t$ . We denote the total number of shortest paths between vertices  $s$  and  $t$  by  $\sigma_{st}$ , and the number passing through vertex  $v$  by  $\sigma_{st}(v)$ . Then, betweenness centrality of a vertex  $v$  is defined as follows:

$$BC(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{st}(v)}{\sigma_{st}} \quad (1)$$

$BC(v)$  is used to measure the number of times that node  $v$  is on the shortest paths that connect all the pairs of nodes in the network. Hence, betweenness accounts for the influences of the node in the network sites and allows one to relate local network structure to global network topology.

### B. BC Algorithm

The complexity to compute betweenness using the straightforward algorithm is  $O(n^3)$ . Recently, Brandes proposed an improved algorithm reduced the complexity to  $O(nm)$ . The main contribution of Brandes [?] algorithm is the elimination of explicit redundant summation of all pair-wise dependencies. Actually, this fast algorithm uses a dynamic programming technique to reduce the search space. A typical graph traversal algorithm like breadth-first search (BFS) and Dijkstra's algorithm can be used for unweighted and weighted graph, respectively. From a given source vertex, it discovers all shortest path to compute the pair-wise dependencies. Define the set of *predecessors* of a vertex  $v$  on shortest path from  $s$  as follows:

$$P_s(v) = \{u \in V : \{u, v\} \in E, d(s, v) = d(s, u) + w(u, v)\} \quad (2)$$

where  $d(s, v)$  is the distance from  $s$  to  $v$ . The time of a direct augment of BFS or Dijkstra's search is still dominated by counting all pair-wise dependencies. In order to eliminate the need of explicit summation of all pair-wise dependencies, it defines the *dependency* of a vertex  $s \in V$  on a single vertex  $v \in V$  as

$$\delta_s(v) = \sum_{t \in V} \sigma_{st}(v) \quad (3)$$

Obviously,  $\delta_s(v)$  is one partial sum of  $BC(v)$  and 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 obey a recursive relation:

$$\delta_s(v) = \sum_{w \in P_s(w)} \frac{\sigma_{sw}}{\sigma_{sw}} (1 + \delta_s(w)) \quad (4)$$

A space efficient data structure for sparse graph  $G$  is an indexed adjacency array data structure. Fig. 1 is an example of an index adjacency array, which is composed of index array and a successor array. In fact, the predecessor set  $P$  records the trace of BFS tree, it is stored in another adjacency array. The parameters  $d, \delta, \sigma$ , and the measure  $bc$  are implemented in linear array. However, the references to the three linear arrays are very dependent on that to the adjacency array of  $G, P$ . Unlike regular applications where the inherent locality and

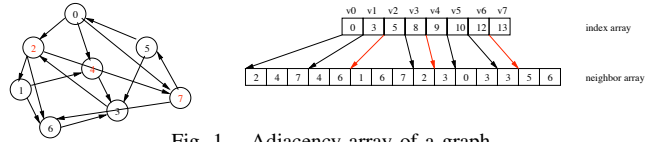


Fig. 1. Adjacency array of a graph.

parallelism are apparent and easy to exploit, it takes careful understanding of the locality and parallelism behavior of irregular applications before one can achieve high performance and scalability of such applications. In the next sections, we present a detailed analysis of its characteristics, which are representations of a large class of irregular applications.

## III. EXPERIMENTAL METHODOLOGY

In this section we first describe program and data sets used in the experimental evaluation, then introduce the multi-core platforms for executing the program.

### A. Implementation of BC algorithm

Due to the importance of computing betweenness centrality, there have been several works on parallelization on different parallel architectures [?], [?], [?]. As one of HPCS benchmarks [?], SSCA2 (Scalable Synthetic Compact Application #2) project [?] makes a specification of implementing BC algorithm for evaluating high performance architecture research. The SSCA2 package includes an OpenMP implementation of parallel BC algorithms.

For integrality of this paper, we present a brief description of SSCA2 benchmark (For detail, refer to [?], [?]). SSCA2 consists of a data generator and four kernels. It includes a scalable data generator that produces edge tuples containing the start vertex, end vertex, and weight for each directed edge. The first kernel constructs the graph in a format usable by all subsequent kernels. The second kernel extracts edges by weight from the graph representation and forms a list of the selected edges. The third kernel extracts a series of subgraphs formed by following paths of specified length from a start set of initial vertices. The set of initial vertices are determined by kernel 2. The fourth computational kernel computes a centrality metric that identifies vertices of key importance

along shortest paths of the graph. The graph generator is based on the Recursive MATrix (R-MAT) scale-free graph generation algorithm. This model recursively sub-divides the adjacency matrix of the graph into four equal-sized partitions and distributes edges within these partitions with unequal probabilities. Initially, the adjacency matrix is empty, and edges are added one at a time. Each edge chooses one of the four partitions with different probabilities. At each stage of the recursion, the parameters are varied slightly and renormalized (Multiple edges, self-loops, and isolated vertices, are ignored). The algorithm also generates the data tuples with high degrees of locality. Thus, as a final step, vertex numbers must be randomly permuted, and then edge tuples randomly shuffled.

Based on the parallel algorithm proposed in [?], [?], BC algorithm is parallelized using OpenMP in SSCA2 benchmark. Fig. 2 shows the pseudocode for visiting neighbors of vertices in one level.

```

1  #pragma omp for schedule(dynamic)
2  for each v in Q {
3      /* all vertices in current queue */
4      for each neighbors w of v {
5          /* possible shared by more than
6           one vertex in Q */
7          omp_set_lock(&lock[w]);
8          if (d[w] < 0) {
9              enqueue w -> Q;
10             d[w] = d[v] + 1;
11         }
12         if (d[w] == d[v] + 1) {
13             sig[w] += sig[v];
14             append v -> P[w];
15         }
16         omp_unset_lock(&lock[w]);
17     }
18 }

```

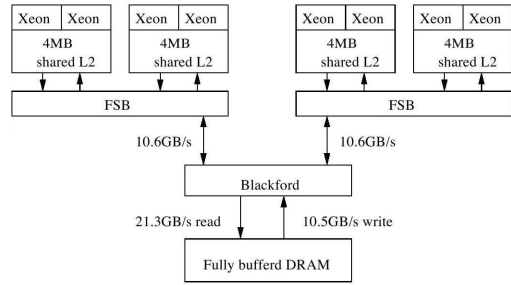
Fig. 2. An illustration of BFS codes at one level

### B. Multi-core Platforms

In the context of parallel BC program, this section summarizes a set of architectural features of two leading multi-core architectures: Intel Clovertown and Sun Niagara1.

Intel Clovertown has dual-socket and per socket integrates two dual-core Xeon chips onto a single multi-chip module (See Fig. 3). Each core can fetch and decode four instructions per cycle, and can execute 6 micro-ops per cycle. Each Clovertown core includes a 32KB, 8-way L1 cache, and each chip has a shared 4MB, 16-way L2 cache. Each socket has a single front side running at 1.33GHz connected to the Blackford chipset, which provide the interface to deliver an aggregate read memory bandwidth of 21.3GB/s. The full system has 16MB of L2 and 74.67GFlops/s peak performance. The OpenMP program is compiled with Intel C/C++ compiler.

Sun Niagara1 is a multi-core architecture integrating SMT and CMP technique. As shown in Fig. 4, there are 8 cores, each of which runs 4 thread streams. An Niagara processor runs up to 8 cores simultaneously for a total of 32 concurrent threads or strands. The OS provides a view of 32 cores to programmers. A strand that stalls for any reason is switched out and its slot on the pipeline is given to the next strand



Courtesy: S. Williams et.al.

Fig. 3. Intel Clovertown chip architecture

automatically. The stalled strand is inserted back in the queue when the stall is complete. All cores are connected by a high-speed, low-latency crossbar in silicon. Each core is a single issue, in-order execution unit. However, a single floating-point unit (FPU) is shared by all cores. Each core has a private L1 data/instruction cache. There is a twelve-way associative unified 3MB L2 on-chip cache. All cores share the entire L2 cache. The memory model supports uniform memory access (UMA) across all cores. The OpenMP program is compiled with Sun Studio.

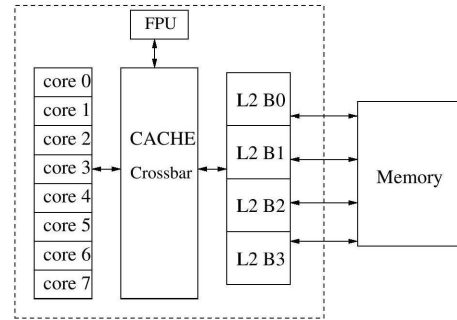


Fig. 4. Sun Niagara1 chip architecture

Table I summarizes the main architectural parameters of Intel Clovertown and Sun Niagara1.

### IV. ANALYSIS OF CHARACTERISTICS FOR BC ALGORITHM

In this section we describe the irregular characteristics of BC algorithm. By going deep into the algorithm and data structure we observe that BC algorithm exhibits three kinds of remarkable irregular behavior:

- 1) *dynamically non-contiguous memory access pattern,*
- 2) *unstructured parallelism,*
- 3) *low arithmetic intensity.*

In the following text, we refer to a term of *scale* as the problem size, where the number of vertices is  $2^{scale}$ .

#### A. Dynamically non-contiguous memory access pattern

For instance, during the BFS phase, a queue is used to maintain the current vertices that is being extended (visiting the neighboring vertices of a vertex is referred to as an extension operation). The effectiveness of the existing locality optimization techniques, such as prefetching and speculation rely on the continuity of the neighboring vertices and regular

TABLE I  
SUMMARY OF MULTI-CORE ARCHITECTURAL FEATURES

chip	Type	clock	L1 cache	$\frac{flops}{cycle}$	$\frac{cores}{sockets}$	L2 cache	Gflops	Memory bandwidth	flop:byte
Clovertown	super scalar out of order	2.3GHz	32KB	4	4/2	16MB 4MB/2cores	74.7	21.3GB/s	3.25
Niagara1	single issue in order	1GHz	8KB	1	8/1	3.2MB shared	1	20GB/s	0.05

distance of different region of neighboring vertices in adjacency arrays, is very bad. In a scale-free sparse graph, the degrees or neighbors of vertices are highly variable.

Fig. 5 plots the variance of degrees with different number of vertices. The variance is more than 50, which means the distance between two neighboring regions ranges from 1 to 50 units. Much worse, the variance is dependent of input sets. With increasing number of vertices, the variance also increase. Therefore, there is no such prefetching or speculation method which could be effective in all cases. Note that adjacency array arranges the neighbors of one vertex in a linear sub-region. The conventional data prefetching should help for locality. However, we observe that the average of degrees is very low (See Fig. 6). Due to the distance between two neighboring region is highly variable, most of the prefetched data could be of no use. Therefore, it wastes memory bandwidth and increase the number of cache missess.

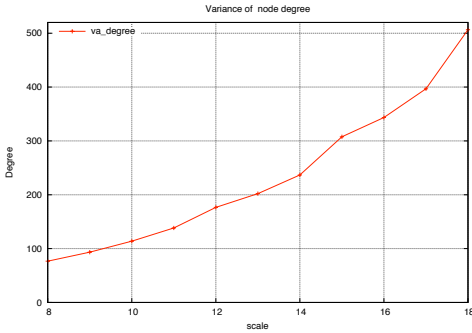


Fig. 5. The variance of degree

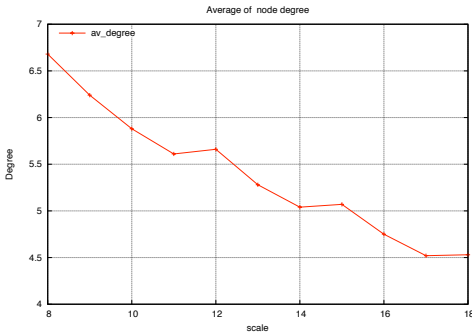


Fig. 6. The average of degree

Considering the simple example shown in Fig. 1, we assume that the nodes  $v_2$ ,  $v_4$  and  $v_7$  are currently in queue from which we pick nodes and process them. Notice that not only the neighboring nodes of  $v_2$ ,  $v_4$  and  $v_7$  are located in different

region in the adjacency array, but also the strides between the different regions are not constant. Also, the references to  $d, \delta, \sigma, BC$  are almost random because the neighbors or predecessors of a node depends on the input graph. For an instance of visiting neighbors of  $v_2$ ,  $v_4$  and  $v_7$ , the sequence of reference to the array  $d$  is  $d[1], d[6], d[7], d[3], d[5], d[6]$  (the same for array  $\sigma$ ). The dependence between  $d, \delta, \sigma$ , and  $bc$  and the adjacency arrays means that the references to  $d, \delta, \sigma$ , and  $bc$  are determined at runtime. Therefore, such non-contiguous or non-linear memory access pattern cannot benefit from current prefetching or speculation techniques.

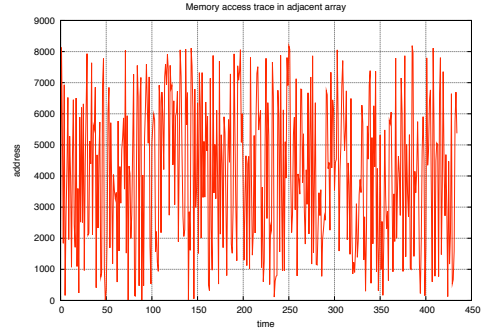


Fig. 7. Trace of memory access in adjacent array

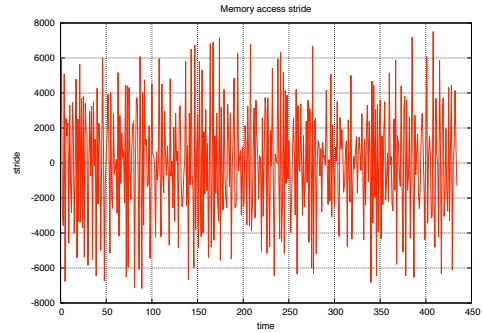


Fig. 8. Strides of memory access in adjacent array

Fig. 7 and Fig. 8 show the memory access pattern in adjacent array. In Fig. 7 we track the accessed entries in *neighbor array* during BFS. The y-axis in Fig. 7 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  $t + 2$ . Fig. 8 depicts the distance between two consecutive accesses of neighboring regions. 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 impractical.

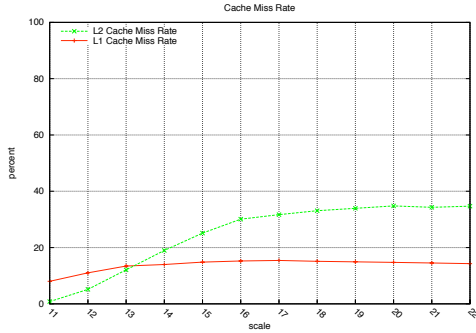


Fig. 9. The L1/L2 cache miss rate

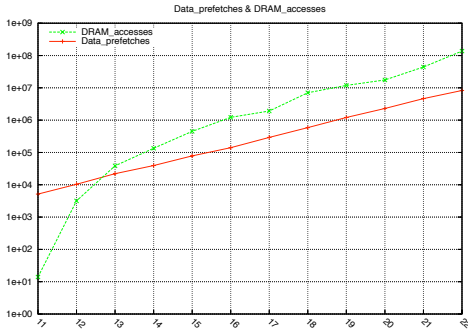


Fig. 10. The performance of memory access

Accordingly Fig. 9 depicts the L2 cache performance of BC program. When the number of vertices is  $2^{12}$ , the work set of BC is larger than the capacity of L2 cache. Thus, we observe the number of L2 cache misses increases significantly when the number of vertices is larger than  $2^{12}$ . Looking at Fig. 10, when L2 cache is larger enough to contain all data structure, data prefetching work well because it exploit locality for visiting the contiguous neighboring region of one vertex and most of non-contiguous memory access are satisfied within L2 cache. When the memory usage exceeds the capacity of L2 cache, prefetching have little effect on reducing the number of system memory accesses.

### B. Unstructured parallelism

The available parallelism within an extension operation is proportional to the degrees of vertices. However, the degrees in the scale free graph obeys a power-law distribution [?], which means most of vertices has low degrees. Therefore, on many-core architectures with massive cores the parallelism will be very fine-grained. In order to utilize the ample processing units, an alternative way is to exploit more parallelism in BC algorithm, i.e. multi-grain parallel algorithms.

Intuitively, BC algorithm exhibits three level of parallelism itself. Each BFS from one source vertex can be started in parallel. For example in Fig. 1, two BFS searches from vertex  $v_0$  and  $v_2$  can be dispatched to different parallel threads, respectively. The coarse-grained parallelism require multiple copies of the whole data structure. The medium-grained parallelism is exploited when two threads are visiting the neighboring vertices of different vertices in the same level.

For example,  $v_2, v_4, v_7$  lie in the same level of BFS tree. When visiting their neighbors, three parallel threads are activated to do the three task, respectively. However, if two vertices share the same neighboring vertices, a synchronization is forced to keep the shared neighbors being visited for just one times. In Fig. 1  $v_2, v_4, v_7$  are at the same level of BFS tree. There exits parallelism during the extension of them, however,  $v_2$  and  $v_7$  share the same neighboring vertex  $v_6$ , a synchronization between two thread units processing  $v_2$  and  $v_7$  is required so that  $v_6$  is touched by only one thread. The fine-grained parallelism is to visit the neighbors of a vertex in parallel. We may schedule three threads to visit the three neighboring vertices  $v_2, v_4, v_7$  of vertex  $v_0$  in parallel. However, there are two factors hindering the various parallelism. First, The embarrassingly coarse-grained parallelism needs a copy of all data structures in the memory space of each thread. For a large scale graph in real world, the memory space usage is so huge that it often exceeds the physical memory even in traditional parallel computers, not speaking of current many-core architectures with small on-chip memory size. On the other hand, intensively concurrent memory accesses place burden on bandwidth, then slack the scalability. Second, note that the scale-free sparse graph has few vertices with high degrees. Both the available medium- and fine-grained parallelism depend on the degrees of vertices. Therefore, current parallel implementations [?], [?], [?] which only exploit either one of medium- and fine-grain parallelism can not achieve good performance on massive multi-threading architectures.

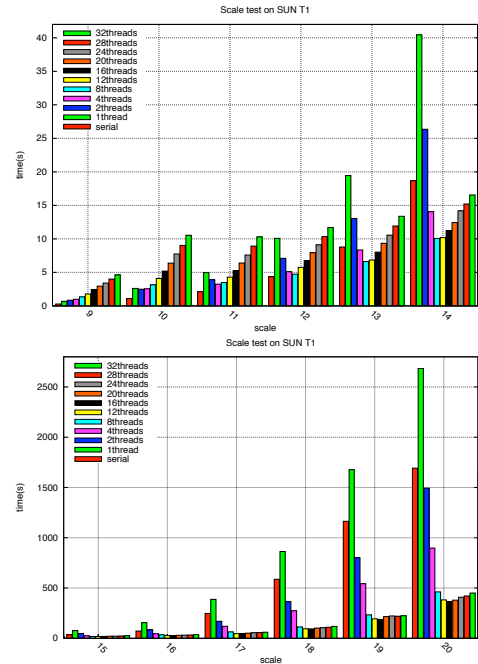


Fig. 11. The scalability on Sun Niagara1

1) *Scalability on Multi-core Platforms:* We examine the performance of BC algorithm on both Intel Clovertown and Sun Niagara1 for scalability and execution time. Fig. 11 and 12 report the scalability with different problem sizes. The parallel program achieves good relative speedups on both architectures

when the number of threads (cores) is less than 8. Although Sun Niagara1 supports hardware SMT so that a parallel program may have more threads (32) to tolerate memory latency, the story is not always true because the dynamically non-contiguous memory accesses. A more important observation is that both architectures show poor absolute speedups and even the parallel program is slower than the serial one. Comparing Fig. 11 with Fig. 12, Sun Niagara1 achieves better absolute speedups because it utilizes the shared cache and higher bandwidth through multi-threading techniques. However, considering the comparison of execution time in Fig. 17, thanks to larger on-chip shared cache, Intel Clovertown achieves faster speeds. The increasing number of threads increasing conflicts of cache and require higher memory bandwidth so that both platforms show comparable execution time.

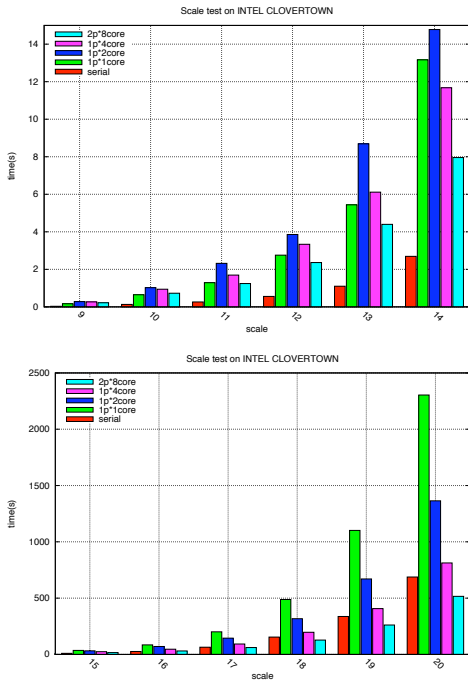


Fig. 12. The scalability on Intel Clovertown

2) *Where is the Performance Going?:* In the multi-grain parallel algorithm, the coarse-grain parallelism is easy to understand, we in detail present the combination of medium- and fine-grain parallelism. Let us denote the set of the vertices that is being extended in the current queue (the  $i$ th level of BFS tree) as  $V_i = \{v_{i1}, v_{i2}, \dots, v_{ik}\}$ . Let  $N_{ij} = \{u_{ij1}, u_{ij2}, \dots, u_{ijl}\}, 1 \leq j \leq k$  denote the neighboring set of vertices of a vertex  $v_{ij}$ . During execution the unvisited neighbor vertices  $u$  ( $d[u] = -1$ ) are added to the current queue and the vertices being extended in the shortest path ( $d[u] = d[v] + w[u][v]$ ) are added to the set of predecessor  $P[u]$ . The multi-grained parallel algorithm *logically* compacts all the neighbors in one level into one large set:  $UN_i = \bigcup_{1 \leq j \leq k} N_{ij}$ , then partitions it among parallel threads. In the case of ignoring shared neighboring vertices, the multi-grained parallel algorithm achieves at least  $p = \frac{\sum_{j=1}^k |N_{ij}|}{\max_{j=1}^k |N_{ij}|}$  times of parallelism than the fine-grained parallel algorithm at

each level. However, in this initial multi-grained parallel BC algorithm there are two problems to be addressed:

- The multi-grained parallel algorithm achieves  $p$  times of parallelism at the cost of concurrently accessing  $p$  times of memory addresses. The small on-chip memory on multi-core is too small to hold the entire combined neighboring set, therefore a large number of high latency off-chip memory accesses happen. Meanwhile, the concurrent off-chip memory accesses make the contention of the limited off-chip bandwidth worse. *Therefore, due to the limitation of memory consumption for coarse-grain parallel program, it is necessary to exploit more fine-grain parallelism.*
- In the fine-grain parallel algorithm, we have to deal with conflict when two vertices share the same neighboring vertices. A fine-grained mutex lock is a solution to the conflicts (See Fig. 2). We note that the size of memory storing lock is the number of vertices, which is usually too huge for the small local memory or cache to hold it. Much worse, the memory access pattern to the lock array depends on that to the vertices, therefore, it is also dynamically non-contiguous. Fig. 13 shows the cache performance of OMP lock synchronization. On the other hand, because the operations on one vertex are involved with only two arithmetic operations, the critical section protected by synchronization operations are so small that the synchronization overhead dilates the size of critical section. Fig. 14 plots the comparison of each lock synchronization overhead and the span of each critical section when the number of threads is 2. If we could exploit enough parallelism, we would observe that the synchronization overhead slightly increases, but the critical section becomes smaller.

In fact, another important observation is that the number of shared vertices in one level of BFS search is little and also highly variance. Fig. 15 plots the distribution of shared neighboring vertices in one level. Most of neighboring vertices are visited only by one vertex in the previous level. The "others" in Fig. 15 is the sum of vertices which are shared by more than 10 vertices. For  $2^{18}$  vertices, there is a few neighboring vertices shared by more than one hundred vertices. *This observation may support the feasibility of optimistic parallelism.*

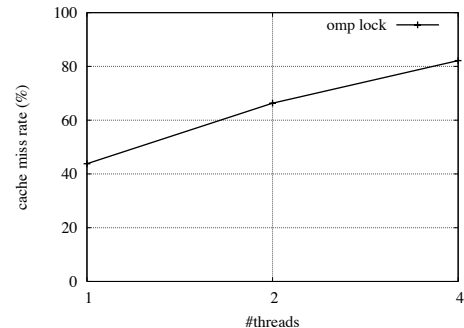


Fig. 13. Cache performance of lock synchronization

### C. Low Arithmetic Intensity

Looking at BFS code in Fig. 2, an extension of one vertex needs two arithmetic (float point addition) operations, about ten memory operations. If the operands are 32-bit, the arithmetic intensity is  $2 : (10 \times 4) = 1 : 20$ . Thus, we estimate the required peak bandwidths on Clovertown and Niagara1 are 1493GB/s and 20GB/s, respectively. Obviously, the low bandwidth on Clovertown starves the floating execution. However, the algorithm can not also achieve reasonable performance on Sun Niagara1 even though the delivered bandwidth is satisfied. For both medium- and fine-grain parallel algorithm, the really required memory bandwidth depends on the degrees of vertices. As noted in the previous sections, the average degree (See Fig. 6) is so low that bandwidth is not an issue.

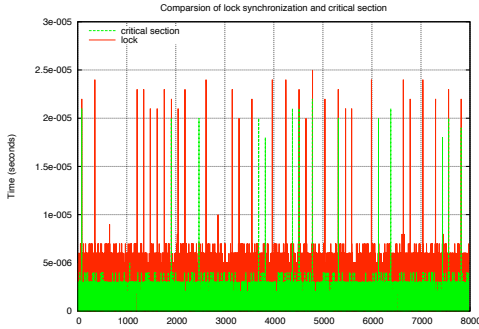


Fig. 14. Time comparison of lock and critical section

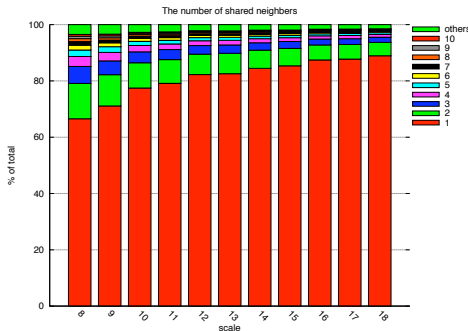


Fig. 15. Distribution of shared neighboring vertices

However, the multi-threading on Sun Niagara1 also does not hide the latency of memory access. As shown in Fig. 16, 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. With the increasing number of problem sizes, the number of the memory access time is at least 4 times than the number of the float operations. Although most of many-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 many-core architectures. *In order to improve the performance of memory bound programs like BC algorithm, the key to a successful parallel program will be an efficient strategy to reduce the memory access latency using the massive parallel thread units.*

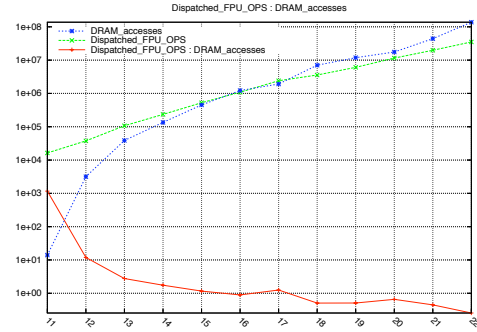


Fig. 16. The ratio of arithmetic operations to memory access

### D. Implications on architecture and programming

Although we present the results of one case of computing betweenness centrality, it represents a class of general applications with *irregular memory access pattern, low arithmetic intensity and unstructured parallelism*, which are different from traditional scientific computing. The experiments give some interesting hints on many-core architectural design space and programming:

- *Architectural support of data-centric execution model.* Since most of current architectures are designed for regular applications with inherent locality. Traditional techniques often do not work well for such irregular programs with dynamically non-contiguous memory access pattern. Most previous work on prefetching also focuses on moving data (mostly contiguous data) from main memory to local memory (either to register or cache) prior to execution. In software prefetching order of execution of a program is predetermined and prefetch instructions are inserted to ensure that the data is available when it is needed by the computation. In other words, conceptually computation threads "pull" the data locally using prefetch instructions. Inspired by dataflow model [?], we think a data-centric execution model probably is an alternative solution. In a data-centric execution model, the data, which is local to a core or thread, will "pull" the threads to execute on the core. The concept may be implemented on a software-managed memory hierarchy architectures like STI CELL and IBM Cyclops64 because programmer can control the data movement through memory hierarchy and thread scheduling. However, this way obviously increase the complexity of programming, a combination of architecture and software should be better.
- *Fine-grain synchronization.* The fine-grain synchronization is not only need by the irregular programs in this paper, but also other traditional scientific computing. As the number of core increasing, the limited on-chip memory size require finer grain parallelism. Thus, like the instance of BC program, the critical section often be too small to amortize the overhead of synchronization itself. G. Tan et.al. [?] has shown that a fine-grain data synchronization can improve the performance on IBM Cyclops64 many-core architecture. An interesting feature of BC algorithm

is that the number of shared neighboring vertices is little. Therefore, optimistic parallelism or transactional memory mechanism may be adaptive to the little conflicts. We will evaluate the performance of existing transactional memory for implementing BC algorithm in the future.

- *Support of multi-grain parallel programming.* As noted in section 4.2, BC algorithm exhibits multi-level of parallelism. However, the multi-grain parallelism is limited by different payoff. In fact, many other applications usually shows the similar behavior. An efficient programming environment should facilitate programmer to easily specify or express the multi-grain parallelism, then a runtime system will automatically solve the optimal combination of different grained parallelism for maximum performance.

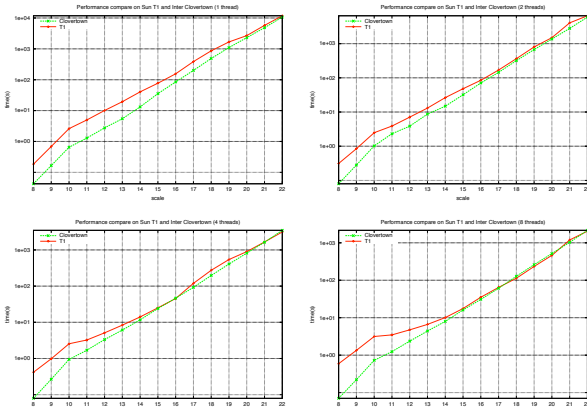


Fig. 17. Performance comparison of Clovertown and Niagara1

## V. CONCLUSION

In this paper we have a deep analysis of an irregular application – computing betweenness centrality (BC) – on multi-core architectures. Our analysis shows that the characterizations of the application are different from traditional scientific computing. This application has three main characterizations.

- *Irregular memory access pattern* This leads to some existing locality optimization techniques such as prefetching and speculation are not effective so that it is hard to exploit locality. A more efficient and smart prefetching and speculation technique is necessary.
- *Low arithmetic intensity* Because those applications have a low computation memory access rate, they are more sensitive to the memory bandwidth and latency. Therefore, high computational capability without sufficient bandwidth will have little help to those applications. Compute-centric model does not fit for those applications.
- *Unstructured parallelism* It is difficult to achieve scalable parallel program because the unstructured parallelism needs more co-design between hardware and software.

From the experimental results we can see the current multi-/many core platforms are not efficient for those applications. We have to go deep into the analysis of algorithm and data structure, thus identify some key features of architecture to improve performance.

## ACKNOWLEDGEMENTS

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] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar, “An 80-tile 1.28tflops network-on-chip in 65 nm cmos,” in *In Proceedings of IEEE International Solid-State Circuits Conference*, 2007, pp. 98–589.
- [2] M. Denneau and H. S. Warren, Jr., “64-bit Cyclops: Principles of operation,” Apr. 2005.
- [3] “www.gpgpu.org.”
- [4] “www.clearspeed.com.”
- [5] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and P. Chew, “Optimistic parallelism requires abstractions,” in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, 2007, pp. 211–222.
- [6] M. Smelyanskiy, V. W. Lee, D. Kim, A. Nguyen, and P. Dubey, “Scaling performance of interior-point method on large-scale chip multiprocessor system,” in *IEEE/ACM SC’07*, 2007.
- [7] L. C. Freeman, “A set of measures of centrality based on betweenness,” *Sociometry*, vol. 40, no. 1, pp. 35–41, 1977.
- [8] 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.
- [9] H. Jeong, S. P. Mason, A.-L. Barabasi, and Z. N. Oltvai, “Lethality and centrality in protein networks,” *Nature*, vol. 411, p. 41, 2001.
- [10] 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.
- [11] V. E. Krebs, “Mapping networks of terrorist cells,” *Connections*, vol. 24, no. 3, pp. 43–52, 2002.
- [12] 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.
- [13] U. Brandes, “A faster algorithm for betweenness centrality,” *Journal of Mathematical Sociology*, vol. 25, no. 2, pp. 163–177, 2001.
- [14] D. Alderson, J. C. Doyle, L. Li, and W. Willinger, “Towards a theory of scale-free graphs: Definition, properties, and implications,” *Internet Math*, vol. 2, no. 4, pp. 431–523, 2005.
- [15] D. A. Bader, “Hpcs scalable synthetic compact applications 2 graph analysis,” [www.highproductivity.org/SSCABmks.htm](http://www.highproductivity.org/SSCABmks.htm), 2006.
- [16] D. A. Bader and K. Madduri, “Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2,” in *The 35th International Conference on Parallel Processing (ICPP 2006)*, 2006.
- [17] —, “Parallel algorithms for evaluating centrality indices in real-world networks,” in *The 35th International Conference on Parallel Processing (ICPP 2006)*, 2006.
- [18] “www.highproductivity.org.”
- [19] “http://www.graphanalysis.org/.”
- [20] J. B. Dennis and D. P. Misunas, “A preliminary architecture for a basic data-flow processor,” in *ISCA ’75: Proceedings of the 2nd annual symposium on Computer architecture*. New York, NY, USA: ACM, 1975, pp. 126–132.
- [21] G. Tan and G. R. Gao, “A study of parallel betweenness centrality algorithm on a many-core architecture,” University of Delaware, Tech. Rep., 2007.