University of Delaware
Department of Electrical and Computer Engineering
Computer Architecture and Parallel Systems Laboratory

A Study of Parallel Betweenness Centrality Algorithm on a Manycore Architecture

Guangming Tan†, Guang R. Gao

CAPSL Technical Memo 77
June 27, 2007

†Email: guangmin@capsl.udel.edu

Abstract

Large scale graph analysis algorithms–such as those in SCCA2 benchmarks studied in this paper–play an increasingly important role in high performance computing applications. Different from most of traditional scientific computing applications, graph algorithms often show dynamic and irregular computing behavior. It is difficult to attain good performance on large scale conventional parallel architectures because these programs exhibit (i). little locality and data reuse, (ii). dynamically non-contigous memory access pattern that is less amendable to static analysis and (iii). fine grain parallelism requring lock synchronization. With the rapid advance of multi-core/many-core chip technology , some new architecture features are emerging: the traditional data cache is being replaced with fast memories (sometime called scratch-pad memories) local to the cores in an explicity (user visible) memory hierarchy, and a large number of processing cores (sometime upto hundreds) are becoming available on a single chip. This presents both challenges and opportunities for mapping graph algorithms to be studied in this paper.

In this paper, a scalable parallel algorithm for computing betweenness centrality in scale free sparse graph is proposed and its performance and scalability is investigated. In particular, our algorithm addresses the parallelization challenges in the following ways:

1. We restructure the parallel algorithm to address the locality challenges by overlapping the latency of prefetching off-chip data into on-chip memory (via an explicit memory heirarchy of the underline many-core architcture) with computation in a pipelined fashion;
2. We “gather” the dynamically non-contigous off-chip memory accesses and convert them into contigous on-chip memory accesses - i.e. “create” on-chip spatial locality just in time;
3. The fine-grain synchronization overhead due to locking is reduced by taking advantage of a specific fine-grain lock mechanism on a many-core architecture and a novel lock free algorithm through exploiting addtional parallelism;
4. Our solution above take full advantage of the ample hardware thread unit resource to assist the parallel computation to manage data movement through memory hierarchy as well as fine-grain data synchronization.

We have implemented our algorithm on the 160 core IBM Cyclops-64 chip architecture. Our experiemental results confirmed the effectiveness of our methods in addressing the performance and scalability challenges of the studied graph problems.

Contents

List of Figures

List of Tables

1 Introduction

Larg scale network analysis is one of the most important reasearches 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 interatction networks [17, 23, 28], study of sexual networks and AIDS [26]and identifying key actors in terrorist networks [12, 25]. In these applications, betweenness centrality [19] is a popular quantitative index for the analysis of large scale complex networks. This metric can be considered as nomalized centrality. It measures the control a vertex has over commnication in the network, and can be used to indentify key vertices in the network. High cetrality 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.

Although graph analysis algorithms have been extensively used in many applications, there are still several grand challenges, which are different from traditional scientific computing in high performance computing field.

- *Little locality and data reuse.* Real world networks are often very large, which size ranges from several hundreds of thounsands to billions of vertices and edegs. A space-efficient data structure of such graphs is itself a big challenge. For huge graphs, parallel out-of-core algorithms [31] are alternative methods to improve the performance on parallel computer with limited physical memory. The data structure is partitionable to fit phsical memory in out-of-core algorithms. This is true for most scientific computing with regular computing behaviors; however, real world networks are highly unstructured. The degree or neighbors of a vertex in such graph is highly variable. The unstructured degree distribution leads to variable strides for memory access so that it is difficult to achieve locality on cahe-based architecutre. From the view point of memory behavior, it is actually a random access. Even in some graph travese algorithms, some vertices are only accessed at once. Therefore, little data reuse can be exploited.

- *Dynamically non-contiguous access pattern.* In most graph algorithms, there is essentially no computation to hide memory costs. Furthermore, the memory access pattern can not be determined statically. For example, in graph travese algorithms, the vertices are visited level by level, the memory access pattern in the next level is dynamically determined in the current level. That is, the memory access pattern is data dependent so that prefetching on currrent conventional architecture unlikely to help. Because the neighbors of two vertices are randomly generated, a significant number of non-contigous memory access are involved.

- *Fine grain parallelism.* Due to the data dependence, i.e. in level-wise graph traverse, we can not directly exploit parallelism among levels because of the data dependence. There exists explicit parallelism when exploring the neighbors of a vertex. On large scale parallel computers, this kind of parallelism is very fine grained. We may exploit the parallelism within accessing the neighbors of all vertex in the same level, but a high efficient lock synchronization mechanism is needed to handle conflicts.

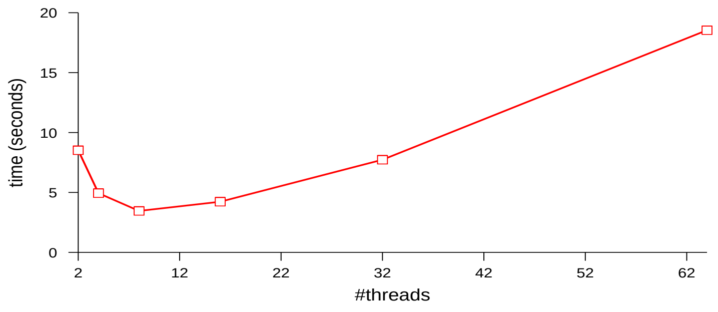


Figure 1: The running time of a fine-grain parallel betweenness centrality algorithm on a manycore system. The algorithm refers to Bader's work and $SCALE = 10$.

Obviously, these characteristics of graph algorithms discourage us to achieve high scalability on conventional parallel computers because most of cache-based parallel computers are inspired by high degree of locality, regular memory access pattern and coarse grain parallelism. With the rapid advance of multi-core/many-core chip technology [1, 2, 18, 21], some new architecture features are emerging: the traditional data cache is being replaced with fast memories (sometime called scratch-pad memories) local to the cores in an explicity (user visible) memory hierarchy, and a large number of processing cores (sometime upto hundreds) are becoming available on a single chip. No cache means that an explict memory hierachy is exposed to programmers and algorithm's designers. On the new architecture with explicit memory hierarchy, we may have to develop a new algorithmic technique to address the issues of poor locality and irregular memory access pattern. Many threads units are typically available on manycore architectures. In order to utilize the threads efficiently, it is necessary to exploit more parallelism in algorithms. Therefore, unlike the coarse paralell algorithm design on conventional parallel computers, this requires indentification of an addtional level of concurrency like fine grain parallelism through restructuring parallel algorithms. The new features of manycore architectures raise a new challenge to high performance algorithms. We may not simply adopt the algorithmic optimization techniques on the emerging manycore architectures. In order to give an intuition of this new shift, figure 1 plots the performance results of a simple fine grain parallel betweenness centrality algorithm, which is implemented using OpenMP [3, 14] on a manycore system. When the number of cores is more than 8, the performance degrades, even begins to *speeddown* when the number of cores is more than 16. The poor scalability forces us to take a fresh look at the parallel algorithms on manycore architectures. However, the trend of scaling performance by scaling parallelism also provides a great opportunity to develop new high performance algorithms, especially the graph algorithms. In this paper, we propose a scalable parallel algorithm for computing betweenness centrality of vertices in a scale free graph, and its performance and scalability is investigated. We address the challenges and the design choices involved in mapping the betweenness centrality algorithm to manycore architecture, where the key point is to take full advantage of the ample hardware thread unit resource to assist the parallel computation to manage data movement through memory hierarchy as well as fine-grain data synchronization. Our specific contributions are as follows:

- A detailed analysis of parallelism in the betweenness centraility algorithm. We propose a multigrain parallel algorithm, which puts emphasis on the domain decomposition and mapping to many threads in the two phases: forward breadth fist search (BFS) and backward accumulation (back-trace).

- We restructure the parallel algorithm to address the locality challenges by overlapping the latency of prefetching off-chip data into on-chip memory (via an explicit memory heirarchy of the underline many-core architcture) with computation in a pipelined fashion.
- We "gather" the dynamically non-contigous off-chip memory accesses and convert them into contigous on-chip memory accesses - i.e. "create" on-chip spatial locality just in time.
- The fine-grain synchronization overhead due to locking is reduced by taking advantage of a specific fine-grain lock mechanism on a many-core architecture and a novel lock free algorithm through exploiting addtional parallelism.
- In order to provide insight on the performance impact of architectural and algorithmic design choice, a comprehensive experimental evaluation of the proposed paralell algorithm is included in this paper. The more interesting points are our experimental indications to runtime and architecture design.

The rest of this paper is organized as follows. In section 2 we summarize the previous work on parallelizing similar graph algorithm on parallel architecture, especially manycore architecutre. Then, we give a simple description of the original algorithm for computing betweenness centrality in section 3, then discusses the granularity of parallelizing betweenness centrality algorithm. Based on the analysis of parallelsim, in section 4, we proposed a multigrain parallel algorithm which addresses the locality problem on explicit memory and reduces the overhead of lock synchronization. Section 5 describes the experiments we have performed to measure the performace of the proposed parallel algorithm on a manycore architecture–IBM Cyclops64. Finally, the conclusions are presented in section 6.

2 Related Work

Due to the importance of graph algorithms, there are many researches on parallelization and optimization. A great deal of parallel algorithms based on PRAM model have been proposed, especially for graph theory [13, 20, 24]. However, to parallelize and optimize graph algorithm on current real parallel computers is also an nontrivial problem. Bader et.al. [7] discusses fast parallel algorithms for evaluating several centrality indices frequently used in complex network analysis. They proposed the first parallel implementations of betweenness centrality algorithm on high-end shared memory symmetric multiprocessor and multithreaded architectures. Their work simply exploited the explicit two level parallelism and did not address the problems of memory behavior and synchronization. The main kernel of betweenness cetrality algorithm is breadth first search (BFS), which actually is a typical representation of graph analysis algorithms. Therefore, we present a brief summary of related graph search algorithms. Park et.al. [27] developed algorithmic optimizations to improve the cache performance of four fundamental graph algorithms. For dense graph, they focused on the data layout of the matrix of a graph and a cache-oblivious technique is used to optimize the performance on memory system based on cache hierarchy. In order to improve the poor locality of sparse graph search, a space efficient data structure–*adjacent array*–is used. In our work, we use *adjacent array* to represent the scale free graph. In [6], Bader et.al. designed a multithreaded algorithms for BFS on multithreaded architecture such as Cray MTA-2 [4]. The algorithm leverage the efficient architectural features such as atomic operations and low overhead of lock synchronization to achieve speedups. However, MTA-2 uses a flat memory structure so that the algorithm can not be simply adaptive to manycore architecture with explicit memory hierarchy. Yoo et.al. [32] presented a scalable distributed BFS scheme on IBM BlueGene/L. On the large scale distributed memory parallel computer, the efficiency of message passing is stressed for improving the scalability of parallel algorithms. In their work, they noted the optimization of local memory usage when the 2D matrix is partitioned, but the technique is designed for the desnse graph's data structure–adjacent matrix. The most recent work on parallel BFS on mulit-core architecutre is described in [30]. The authors presented the challenges and algorithm of parallelizing BFS on a state-of-the-art multi-core processor, the IBM Cell broadband engine [21]. The algorithm uses DMA attached to each SPU to hide memory latency in the explicit memory hierachy. An additional preprocessing is required in their algorithm, where all vertices are partitioned into several disjoint sets. That is, a vertex is exclusively mapped to a SPU. This idea is similar to the 2D adjacent matrix partition in [32], like the algorithm in distributed memory architecture, it involves a number of collective communication operations.

3 The Granularity of Parallelism

We follow the convention of introduction to betweenness centrality presented by Bader's work [5]. Consider a graph $G = (V, E)$, where V and E is the set of vertices and edges, respectively. In the graph model, vertices repsent actors in social network, or proteis in protein interaction network, edges represents the relationships/inertaction between actors/proteins in the social/protein network. The nubmer of vertices and edegs 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). 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}, and the number passing through vertex v by $\sigma_{st}(v)$. Then, betweenness centrality of a vertex v is defined as follows:

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

The main contribution of Brandes [9] 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

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

where $d_G(s,v)$ is the distance from s to v. For a unweighted graph, $w(u,v) = 1$. 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

$$\delta_s(v) = \sum_{t \in V} \delta_{st}(v) \tag{3}$$

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:

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

Like a dynamic programming algorithm, this fast algorithm proceeds as two phases: *forward BFS* and *backward accumulation* (*backtrace*). The algorithm is stated as follows. First, n BFS/Dijkstra's searches from each $s \in V$ are done in the 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 $s \in V$, using the information from the shortest paths tree, where the number of shortest paths through a vertex w is stored in $\sigma(w)$, and predecessor sets along the paths, compute the dependencies $\delta_s(v)$ for all other $v \in V$. In the end, the sum of all dependency values of a vertex v is computed according to recursive relation.

The procedure of computing betweenness centrality starts with each source vertex, search the graph using BFS, then backtrace the tree to accumulate betweenness centrality values. If the search starting with a vertex is consider as a task, the algorithm needs $n = |V|$ sub-tasks to get the betweenness centrality of each vertex. The concurrency can be exploited at three level of granularity for the sequential algorithm: (i) coarse-grain among the sub-tasks (parallelism in line 3). (ii) medium-grain at queue/stack level (parallelism in line 10 and 21). (ii) fine-grain at the neighbors level (parallelism in line 13 and 23). The three levels of parallel granularity are depicted in Figure 2.

- *Coarse-grain:* For the BFS phase, the coarse-grain implementation is embarrassingly parallel. The sum of betweenness centrality is partitioned into $N \gg P$, where P is the number of threads. Each BFS from a vertex independently proceeds in parallel. Each thread has a partial sum of betweenness centrality value. In the end, there is a reduction operation among all threads. Although the coarse-grain parallel algorithm can achieve dynamic load balancing handled by a library or runtime, it needs a whole copy of graph data structure on every processor/thread. For a large scale graph, it easily exceeds the available physical memory, and, even worse, the local memory on each core in manycore architecture is very small, therefore the problem of poor locality becomes more sharp.

- *Medium-grain:* The medium-grain parallelism is exploited when the algorithm visits the neighbors of a vertex in the queue/stack in the same BFS level. In each BFS level, some vertices are selected and organized in the queue (which is a stack when backtracing BFS tree). Each thread dequeues/pops a vertex and tries to visit its neighbors. If a neighbor is visited for the first time, the neighbor vertex is enqueued. The efficiency of medium-grain parallelism depends on the length of the queue in a BFS level and the conflicts of neighbors. The number of enqueued vertices in each BFS level is dynamic which results in load balance problem. Because the neighbors of dequeued vertices are marked if they are visited for the first time, the access to the same neighbor needs to be ordered by a synchronization mechanism which defects the exploited parallelism.

- *Fine-grain:* The fine-grain parallelism in the innermost level loops is exploited by allowing multiple threads to coorperate during visiting the neighbors of a dequeued/poped vertex. The degrees of a vertex determine the available parallelsim. This work studies a scale free sparse graph where the degree of a vertex is low, therefore it is difficult to achieve load balance for large scale threads. For the fine-grain parallelism, a lock is not needed unless there exist parallel edgesbetween two vertices. In this case, we have to keep the order in which the same vertex is visted, using some sychronization mechanism, such as a lock.

Both the medium- and fine-grain parallel algorithm can not achieve load balance because of the sparisity of scale free graph. Because the coarse-grain parallel agorithm can not exploit more intristic parallelsim of BFS and the locality is limited by the available size of physical memory, the scalability is poor. It is expected that we can achieve better performance using hybrid grain parallelsim. However, a simple multigrain parallel algorithm can not achieve locality on explicit memory hierachy. Besides, we have to seriously take lock synchronization into consideration because we need to exploit fine-grain parallelism.

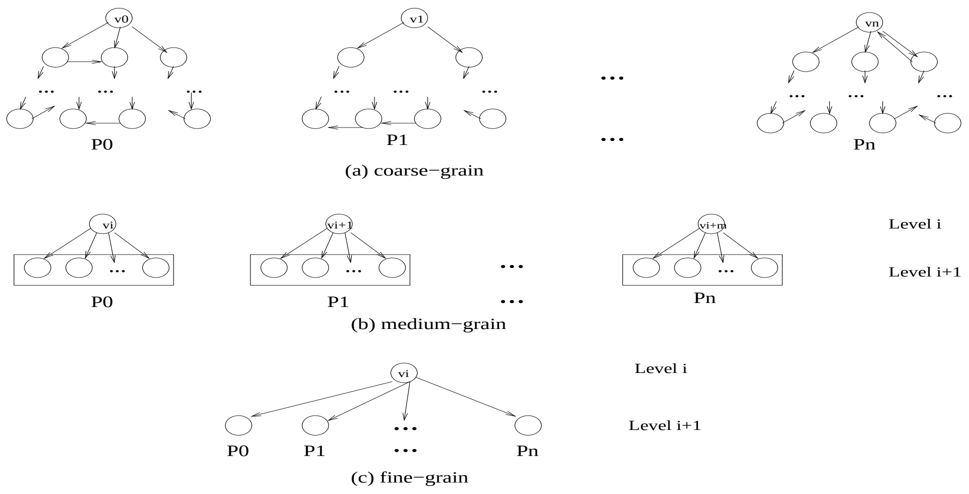


Figure 2: (a) A coarse-grain decomposition according to different source vertex. (b) medium-grain parallelism exploited for a queue/stack at each level of BFS. (c) fine-grain parallelism visits all neighbors of each vertex

4 Scalable Multigrain Parallel Algorithm

In this section, we propose the detailed parallel algorithm which exploits the mulitgrain parallelism for computing betweenness centrality. First, we present the parallel algorithm's framwork including the mapping between threads and fine-grain tasks. Because we need to address the issue of explicit memory hierachy, then, the preliminary parallel algorithm is restructed to be adaptive to memory hierachy. In the end, the novel lock free algorithm and the technique of using specific lock synchronization mechanism is described.

4.1 Task-Thread Mapping

In order to exploit multigrain parallelism, all threads are partitioned into several groups. The thread groups themselves exploit coarse-grain parallelism and the threads within each of these groups are used to exploit medium- and fine-grain parallelism. For the coarse-grain parallelism, each BFS from a source vertex is a task. The threads in the same group select the same task, start BFS from the source vertex and backtrace to compute a partial betweenness centrality value along the BFS tree in parallel. We denote S_i as the source vertex set on thread i.

The BFS algorithm traverses the graph level by level. Data dependencies exist between two consecutive levels, that is, the vertices in level $i+1$ are the unvisited neighbors of vertices in level i. In order to follow the data dependencies, we use level a synchronization algorithm as other parallel BFS algorithms. However, our proposed algorithm visits the neighbors of all vertices in each level in parallel, not just the neighbors of one vertices. When the algorithm finishes selecting the unvisited neighbors of vertices

in level i, it partitions all vertices in level $i+1$ among the threads in a group and repeats the same procedure. Let us denote the set of vertices in level i as $V_i = \{v_{i1}, v_{i2}, ..., v_{in}\}$. The neighbors set of each vertices v_{ij} is denoted as $W_j = \{w_{j1}, w_{j2}, ..., w_{jm_j}\}$ for $1 \leq j \leq n$. Intuitively the multigrain parallel algorithm compacts all neighbor sets into one larger set $NW_i = \bigcup_{1 \leq j \leq n} W_j$, which is a union set of the neighbor sets of vertices in level i, then distibutes the new neighbor set NW_i among the threads in a group. Because the graph has parallel edges and two different vertices in the same level may share the the same neigbors, serveral threads may visit the same neigbors w simultaneously. The access to calculate the distantce ($d[w]$) from the source vertex and paths information ($\sigma[w], P[w]$) through neighbors w must be protected with some synchronization mechanism such as locks or atomic operatoins.
The parallelization of computing partial betweenness centrality values is similar to that of BFS. Assume that there are n vertices in level i $W_i = \{w_{i1}, w_{i2}, ..., w_{in}\}$ and the predecessors set of each vertex is $V_j = \{v_{j1}, v_{j2}, ..., v_{jm_j}\}$ for $1 \leq j \leq n$. In the same way, all predecessors sets are compacted into one larger set $PV_i = \bigcup_{1 \leq j \leq n} P_j$, then partitioned among the threads in a group. Again, some mutex mechanism is requred to protect the conflicts when calculating the same partial value ($\delta(v)$) on some different threads. In the end, a parallel reduciton operation happens on all thread groups so that the final betweenness centrality value is accumulated from the partial values of each threads. In Algorithm 1, we give the pseudocode description of this high level parallel algorithm. For a reasonable simple optimizaiton, the basic data structure *queue* representing BFS tree is shared by the *stack* in this algorithm.
The preliminary version of parallel algorithm does not address the challenge of explicit memory hierarchy on manycore architecture and lock synchronization overhead for multi-thread systems. Although lock synchronization has been a serious problem for the scalability of parallel algortihms on conventional architectures, the optimization of explicit memory hierarchy is a new challenge on manycore architecture. In the next sub-sections, we propose an efficient strategy to optimize the parallel algorithm for explicit memory hierachy, then reduce the lock synchronizaiton overhead based on the optimized algorithm.

Algorithm 1 A high level description for parallel betweenness centrality algorithm on a thread of group i

Input: G(V, E)

Output: Array BC[1...n], where BC[v] gives the centrality metric for vertex v

```
1  for all $v \in V$ pardo
2    $BC_i[v] = 0$
3  for all $s \in S_i$ do
4    $P[w] \leftarrow$ empty list, $w \in V$
5    $\sigma[t] \leftarrow 0, t \in V; \sigma[s] \leftarrow 1$;
6    $d[t] \leftarrow -1, t \in V; d[s] \leftarrow 0$
7    Q←empty queue
8    level = 0
9    enqueue $s \leftarrow Q_{level}$
10     while $Q_{level}$ not empty do
11       $NW_{level} \leftarrow$ neighbors($\{v_{level,1}, v_{level,2}, ..., v_{level,n}\}$, $Q_{level}$)
12       for $w \in NW_{level}$ pardo
13         lock;
14         if $d[w] < 0$ then
15           enqueue $w \rightarrow Q_{level+1}$
16           $d[w] \leftarrow d[v] + 1$
17         if $d[w] = d[v] + 1$ then
18           $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$
19           append $v \rightarrow P[w]$
20         unlock;
21       level = level+1;
22       sync;
23     $\delta[v] \leftarrow 0, v \in V$;
24     level = leve-1;
25     while $level \geq 0$ do
26       $PV_{level} \leftarrow predecessors(\{w_{level,1}, w_{level,2}, ..., w_{level,n}\}, Q_{level})$
27       for $v \in PV_{level}$ pardo
28         lock;
29         $\delta[v] \leftarrow \delta[v] + \frac{\sigma[v]}{\sigma[w]}(1 + \delta[w])$
30         unlock;
31       if $w \neq s$ then
32         lock;
33         $BC_i[w] \leftarrow BC_i[w] + \delta[w]$
34         unlock;
35       level = level - 1;
36       sync;
37     ParReduction(BC, $BC_i$)
```

4.2 Optimization of Explicit Memory Hierarchy

On manycore architectures with explicit memory hierarchy, the on-chip memory is much smaller than off-chip memory, but the on-chip bandwidth is much larger than off-chip bandwidth. The relatively high latency of off-chip memory access creates a bottleneck in achiving performance for memory intensive applications, especially for that with irregular memory access. A generic techinque is to use DMA or multi-thread to tolerate memory access latency. If there are sufficent computation between memory access, the latency is not a problem. However, as shown in prevalthough sections, we do not have enough computation when traversing the neighbors of a vertex in the graph. Therefore, when all threads access memory concurrently, the off-chip memory latency and bandwidth becomes a bottleneck.

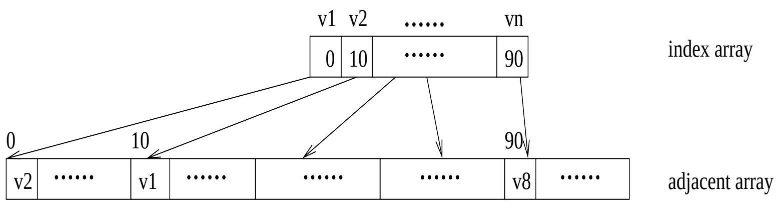


Figure 3: The adjacent array data structure

An efficient data structure representing a scale-free sparse graph is *adjacent array*. Figure 3 is an example of the adjacent array. The adjacent array structure is composed of two arrays: *index array* and *neighbors array*. The ith element of *index array* locates the offset of the neighbors of vertex i in *neighbors array*, where all neighbors of vertex i are continously stored. Therefore, prefetch or blocking techniques can be exploited on cache memory architecture when the program is visiting the neighbors of a vertex. However, the neighbors of two different vertices may be not consecutively store in *neighbors array*. Similarly, the memory access to other arrays (d,σ,δ, P) for keeping path information also are non-continous. For simplicity, we only present the issue of *neighbors array* in the following algorithmic descriptions. The main memory access operations in BFS and backtrace phases are to extract the neighbors/predecessors of a vertex in *queue*. Because the algorithm extracts the neighbors of all different vertices in a level, it is possible that there are many non-continuous memory access.

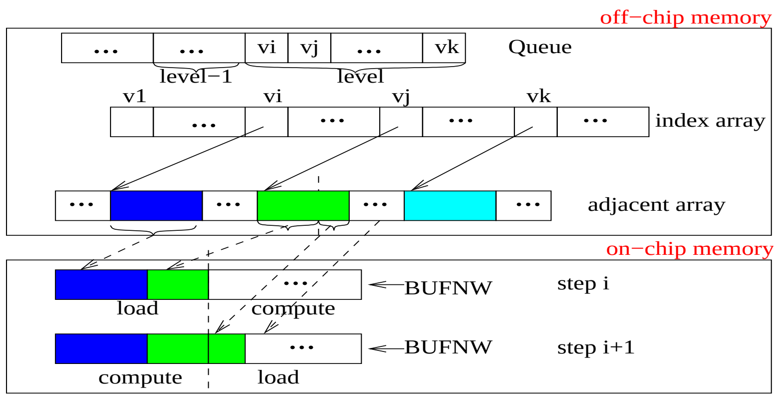


Figure 4: An example of memory images in double buffering framework. It depicts the double buffers of adjacent array between two time steps. In each step, the *load* and *computation* occurs concurrently. Because of the limitation of buffer size in on-chip memory, the green block in *adjacent array* have to be split and transferred to on-chip memory in serveral time steps. Only the double buffers $BUFNW$ are depicted. Another double buffers $BUFQ$ which are alternatively read/writen by *computation* and *store* threads is not presented in this figure.

Futhermore, the low latency and high bandwidth of on-chip memory in memory hierachy leads to following observations:

- To transform sparse off-chip memory access to continous on-chip memory access
- To decompose task into sub-tasks so that different level memory accesses are overlapped.
- To exploit additional parallelism within on-chip memory.

The compaction of the neighbors of different vertices in the *queue* provides the opportunity to exploit additional parallelism. Due to the limited on-chip memory, we have to load a small size of data from off-chip memory each time the neighbors of a vertex are visited, thus the task for transforming sparse off-chip memory access to continous on-chip memory access is naturally decompsed several sub-tasks. In order to achieve overlapped memory access, double bufferring is an alternative technique (See Figure 4). In the framework of double bufferring, the threads in a group are further divided into *computation threads* and *helper threads*. The *helper threads* are exclusively used to load/store data between off-chip and on-chip memory. The *computation threads* only access on-chip memory and proceed only when the required data is available in on-chip memory, which means that the main work of the *computation threads* involves only on-chip memory accesses. For simplicity, we only give the high level algorithmic descprition of the parallel BFS in one level(See Algorithm 2). The pseudocodes describe the algorithm iteration in one BFS level. Some neighbors, which have been visited when exploring other vertices before, are loaded into on-chip memory again because of the requirement of computing path information. In case that the buffers are full when partial neighbors of a vertex are loaded, we need to keep an offset in the adjacent array of the vertex. As an optimization for usage of *helper threads*, the *helper threads* are divided into *load threads* and *store threads*.

Algorithm 2 A double-buffering framework for parallel BFS in betweennees centrality algorithm. The backtrace to computing partial values can be implemented in the same way.

```
while $Q_{level}$ not empty do
    if load thread
        selects $v_{level,i} \in Q_{level}$ which has remained neighbors needed
        to be loaded into on-chip memory
        $BUFNW_{level} \leftarrow$ compact(adjacent array, $v_{level,i}$, bufsize)
    else if store thread
        flush($BUFQ_{level+1}, Q_{level+1}$)
    else if computation threads
        enqueue($BUFQ_{level+1}$, $BUFNW_{level}$)
        /*including computing path information*/
```

4.3 Optimization of Lock Synchronization

Bacically, the on-chip images of compacted *adjacent array* is evenly partitioned and mampped to the threads in a group. A good load balancing is achieved using round-robin strategy. However, as mentioned in previous sections, lock synchronizations are required for protecting the conflicts when serveral threads are visiting the same neighbor vertex. In this section, we propose algorithms to reduce the overhead of lock synchronization.

4.3.1 Lock Free for Backtrace

In the *adjacent array* data structure, each edge is composed of *start* and *end* vertex. From the view of computing behavor, the parallel BFS algorithm only needs to exclusively access the *end* vertex. However, the inverse procedure of accumulating betweenness centrality values requres an exclusive access to both *start* and *end* vertex of an edge (See line 29 and 33in Figure 1). Here we propose a lock free algorithm, which benefits from double-buffering for calculating betweenness centrality in the backtrace phase.

The *helper* threads are mainly used to transfer data between two level memory. In the backtrace phase, our observation is that the ratio of computation to memory access is high, which means multiple *helper* threads are not necessary, and the calculations are the accumulations of several partial values, which can be computed independently. Because the workload of *helper* threads is low, we can exploit some additional parallelism, then assign some *light weight* computation sub-tasks to *helper* threads. According to equation 3, we can divide the computation of $\delta(v)$ into several $\frac{\sigma_v}{\sigma_w}(1+\delta(w))$ sub-tasks and $\sum_{w:v\in P(w)}$ sub-tasks which only need addtion operations. Therefore, the *computation* threads only compute the partial values of some $\delta[v]$ in parallel. These partial values are stored in one on-chip buffer and the reduction of sum is delayed to be done by one *helper* thread, i.e. *store* thread, in the next iteration. Figure 5 illustrates an example of the lock-free calculation of the δ array. In current iteration, the computation of partial values and the sum of partial values computed in previous iteration proceed in parallel, thus, in addition to hide the off-chip memory access, some computation tasks are also overlapped. Because only one *load* and one *store* thread are used, there is no data race for accumulating the final values on off-chip memory.

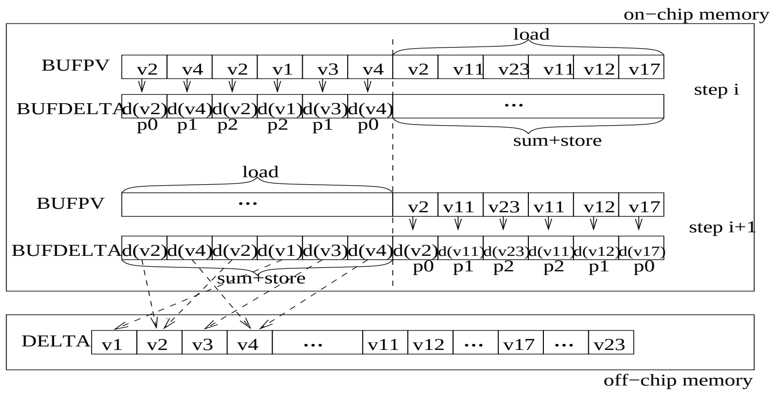

on−chip memory
load
BUFPV
v2 v4 v2 v1 v3 v4 v2 v11 v23 v11 v12 v17
step i
BUFDELTA
d(v2) d(v4) d(v2) d(v1) d(v3) d(v4)
...
p0 p1 p2 p2 p1 p0
sum+store
load
BUFPV
...
v2 v11 v23 v11 v12 v17
step i+1
BUFDELTA
d(v2) d(v4) d(v2) d(v1) d(v3) d(v4) d(v2) d(v11) d(v23) d(v11) d(v12) d(v17)
p0 p1 p2 p2 p1 p0
sum+store
DELTA
v1 v2 v3 v4 ... v11 v12 ... v17 ... v23
off−chip memory

Figure 5: An example of computing δ array (DELTA in this figure) without lock synchronization. In this algorithm, only one *load* and one *store* thread are used.

Algorithm 3 The parallel algorithm for computing betweenness centrality values in backtrace phase. The double buffering keeps the order of read/writer of $BUF\delta(v)$ among *computation* and *helper* threads

```
while $Q_{level}$ not empty do
  if load thread
    if some $w_{level,j}$ all of which predecessors have been loaded
      $BC_i[w] \leftarrow BC_i[w] + \delta[w]$
    selects $w_{level,i} \in Q_{level}$ which has remained predecessors
    needed to be loaded into on-chip memory
    $BUFPV_{level} \leftarrow$ compact(predecessors, $w_{level,i}$, bufsize)
  else if store thread
    for $v \in BUFPV_{level}$ do
      $\delta(v) \leftarrow$sum($BUF\delta(v), \delta(v)$)
  else if computation thread
    $BUF\delta$[0...bufsize]$\leftarrow$ 0
    for $v \in BUFPV_{level}$ pardo
      $BUF\delta(v) \leftarrow$accumulate($BUF\delta(v)$, $BUF\delta(w)$)
```

In order to eliminate the lock synchronization of visiting the *end* vertex w to accumulate betweenness centrality values $BC(w)$, we note that the accumulation operation is only performed after all predecessors v of the vertex w are visited to compute the value δ, therefore, we assign the accumulation of betweenness centrality values $BC(w)$ to the *load thread*. For a given vertex w in the shortest path, the *load thread* updates its betweenness centrality values $BC(w)$ after the traverse of its all predecessors is completed. Thus, the parallel algorithm of backtrace to compute betweenness centrality values in a level is presented in Algorithm 3.

4.3.2 Fine-grain Lock for BFS

When the threads are traversing the neighbors of some vertices in the same level, conflicts occurs only if they are visiting the same vertex. That means a lock synchronization should be associated with a single vertex, not the whole adjacent array. It is intuitive that a fine grain lock can handle this problem. In conventional architecture, we have to use an additional lock array to implement the fine grain lock mechanism. The size of lock array is the same with the number of vertices. Thus, for a large scale graph, the lock array locates in the off-chip memory and the performance degrades because of relatively high latency of off-chip memory on a manycore architecture. In order to save the additional lock array, we use the fine grain machanism–Synchronization State Buffer (SSB) [33] proposed for IBM Cyclops64 manycore architecture. Unlike the full/empty bits on MTA-2, SSB is a small buffer attached to the memory controller of each memory bank. It records and manages states of actively synchronized data units to support word level synchronization. SSB associates locking functions with memory locations dynamically. When a memory location needs to be accessed exclusively, the lock operation is issued with the address of the location, instead of a location in a lock array. Algorithm 4 details the fine grain lock synchronization algorithm using SSB on IBM Cyclops64.

5 Experimental Results

This section describes the experimental results and a comprehensive evaluation of the proposed parallel algorithm. We implemented the algorithm using a multi-thread libray on IBM Cyclops64 (C64), which

Algorithm 4 Fine grain lock synchronization algorithm using SSB. *swlock_l* and *sunlock* are the functions issuing a lock/unlock operation at a memory location, respectively.

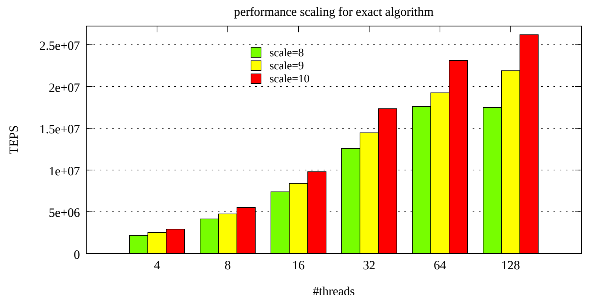
```
for $w \in NW_{level}$ pardo
    rt = swlock_l(&(d[w]), &dd);
    if rt == 0
        if $d[w] < 0$ then
            enqueue $w \rightarrow Q_{level+1}$
            $d[w] \leftarrow d[v] + 1$
            $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$
            append $v \rightarrow P[w]$
        else if $d[w] = d[v] + 1$ then
            $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$
            append $v \rightarrow P[w]$
        sunlock(&(d[w]));
    else if $d[w] = d[v] + 1$ then
        while ((rt = swlock_l(&(d[w]), &dd)) != 0);
        $\sigma[w] \leftarrow \sigma[w] + \sigma[v]$
        append $v \rightarrow P[w]$
        sunlock(&(d[w]));
```

is a 160-core chip architecture. C64 represents a class of manycore architectures featuring no cache and many threads.

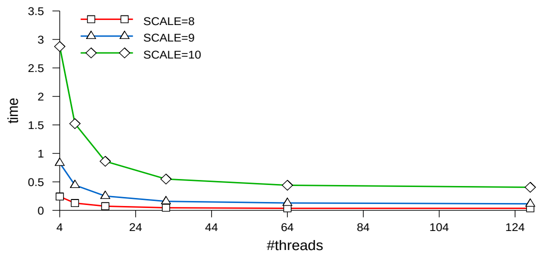
5.1 Experimental Platform

C64 has evolved from a preliminary design of BlueGene/Cyclops architecture [10]. The C64 chip contains 160 thread units (TU) (running at 500MHz) and 160 embedded SRAM memory banks (32KB each) in a single silicon die, and with a peak performance of 80GFLOPS. There are 80 floating point units, each of which is shared by two TUs. A 32KB instruction cache, is shared among 10 TUs. C64 has efficient support for thread level execution, such as ISA-level sleep/wakeup instructions. C64 features an explicitly addressable three-level memory hierarchy without data cache. A portion of each SRAM bank can be configured as the *scratchpad memory (SP)*, which can be accessed by a corresponding TU with very low and deterministic latency. The remaining sections of all on-chip SRAM banks, together, form the *global memory (GM)* that is uniformly addressable by all TUs. There are 4 memory controllers connected to *off-chip DRAM* banks (up to 2GB). All memory words are 8 bytes wide and the memory is byte-addressable. The memory accesses to contiguous address space are interleaved. For example, the access to GM is interleaved to SRAM banks by a 64-byte boundary, which ensures the full utilization of the bandwidth and the SSBs attached to all memory banks. Memory accesses to GM and DRAM go through an on-chip crossbar network, which sustains a 384 GB/s on-chip bandwidth. The crossbar also guarantees a sequential consistency memory model for the C64 chip architecture. Fence-like instructions are not needed to ensure the order between memory operations [18]. C64 provides no hardware support for context switching, and currently uses a non-preemptive thread execution model.

IBM Cyclops64 supercomputer is an on-going project and there is no real machine to date. The simulation tool, named Functionally Accurate Simulation Toolset (FAST) [15], is designed for the purpose of architecture design verification and software development. Based on the FAST simulator, a thread virtual machine (TNT) [16] and OpenMP runtime [14] are implemented to support a multi-thread programming environment. The parallel algorithms are implemented using TNT library.



(a)



(b)

Figure 6: *Strong scaling* results of parallel betweenness centrality algorithm. The number of vertices $n = 2^{scale}$, $E(n) = 7n$.

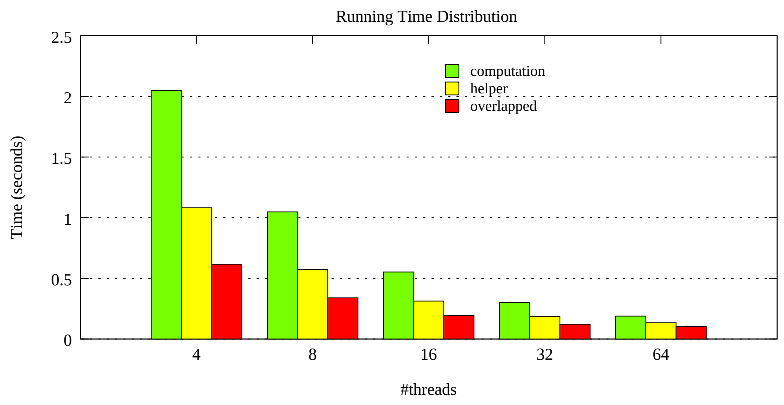


Figure 7: Time distribution and achieving off-chip memory latency tolerance.

5.2 Performance Evaluation

On traditional supercomptuers, most parallel applications have put emphasis on *weak scaling*, where speed is achieved when the number of processors is increased while the problem size per processor remains constant, effectively increasing the overall problem size. The *weak scaling* measures the exploitable parallelism to solve a larger problem. We can achieve better *weak scaling* by increasing the computational power of a single processor. However, on the emerging many-core architectures, although the number of cores grows rapidly, the speed of individual processing element is reduced. Therefore, we should measure the achieved speed when the number of processors increased while the overall problem size is kept constant, which effectively decreases the problem size per processor. That means *strong scaling* is greatly emphasized for the fine-grain parallel algorithm on many-core architectures. Due to the limitation of simulator, we only can give the experimental results for small problem sizes, where the *weak scaling* is not addressed.

In our performance evaluation, we use the metrics *TEPS (traversed edges per second)*, which is defined in the *SSCA2* specification [5]. The implementation of our algorithm follows the *exact* algorithm in the *SSCA2* specification. Assume that the number of vertices is n and the number of edges is $E(n)$[1]. Let

[1]In some cases, $E(n)$ only include the edges which satisfies some constraint. For an example, *SSCA2* selects the edges which weight is evenly divsible by 8

$T(n)$ be the execution time, *TEPS* is defined as $TEPS(n) = \frac{n*E(n)}{T(n)}$.
Figure 6 reports the *strong scaling* results. The *TEPS* performance in Figure 6(a) has a favorable sacability with the increasing number of threads. Figure 6(b) plots the corresponding running time. Compared with the performance of simple fine-grain parallel algorithms (See in Figure 1), The overall running time is reduced greatly and scales well with the number of threads. Although the problem size is small, the parallel algorithm achieves linear speedups for all test cases when the number of processors is less than 32. For the test case with problem size $scale = 8$, the performance stops increasing when the number of threads reaches 128. However, the parallel algorithm can ahieve better performance when the problem size increases, i.e. $scale = 9, 10$. Note that the parallel algorithm exploits fine grain parallelism when the neighbors of a vertex are visited and obviously, the degree of a vertex determines the parallelism we can exploit. Our multi-grain algorithm adapts medium-grain parallelism to reduce the number of idle threads, but the maximum degree of the case $scale = 8$ is 64. Therefore the available parallelism for this small problem size leads to the limited performance on 128 threads. For $scale = 9, 10$, the maximum degrees are 94 and 348, the parallel algorithm improves the performance further.

5.2.1 Performance of Memory Hierachy Optimization

In the programming model of many-core architectures with explicit memory hierachy, where on-chip memory is small, we use the on-chip double buffers to hide the off-chip memory latency and the *helper* threads are used to transfer data between two level memory. Figure 7 shows that the effect of memory latency tolerance using double-buffering. In order to figure out the overlapped time, we profiled the execution time of *computation* and *helper* threads. Although the *computation* threads only access on-chip memory, the overall execution time of *computation* threads is more than that of *helper* threads because lock synchronization is required among the *computation* threads in the BFS phase. The parallel algorithm has to load the *adjacent array* into on-chip buffers block by block. According to the feature of BFS algorithm, the computation behavior of each vertex may be different. For an example, if a vertex is not one of the predecessors of the loaded neighbor, we do not need to insert this vertex to the predecessor set of the neighbor, where serveral addtional memory accesses occur. Therefore, the execution time of each block may be different that means workload may not be balanced among threads. Thus, we can not achieve a perfect overlap between *computation* and *helper* threads. With the increasing number of threads, the workload on each thread decreases so that the difference of the workloads is little, then the portion of overlapped time becomes larger. This indicates that a fine-grain parallel algorithm is better for achieving memory latecy tolerance.
Like memssage buffers in message passing on distributed memory computers, a buffer size is an important factor of the scalability of parallel algorithms [22], where the buffer size based on the number of processors/threads prohibit a parallel algorithm scaling on large-scale parallel systems. In our proposed algirthm, the buffer size is independent of the number of threads. Therefore, the scalability of our parallel algorithm is confined by hardware parameters like on-chip memory size and bandwidth, instead of the buffers required by the algorithms. Acctually, an important observation is that a larger buffer has little effect on improving the performance. Note that degree of each vertex in scale-free graph is low so that we can not hide more off-chip memory access by increasing the buffer size. Figure 8 shows that increasing buffer size does not achieve better performance.

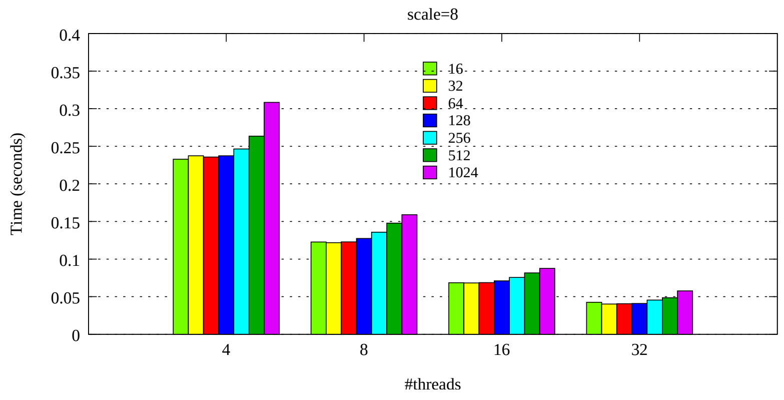


Figure 8: The comparison of running time using different sizes of buffers.

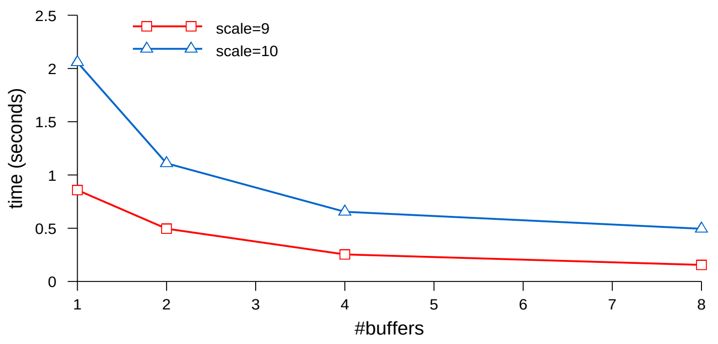


Figure 9: Comparsion of off-chip memory latency tolerance achieved by tuning the number of on-chip buffers.

Our multi-grain parallel algorithm also exploit coarse-grain parallelism, where the threads are partitioned into serveral groups, each of which has independent double buffers. The coarse-grain parallelism can avoid the lock synchronization caused by fine-grain parallelism in the BFS phase, but current manycore architectures with small on-chip memory size do not allow us to exploit more coase-grain parallelism. In other words, the coarse-grain parallel algorithm means multiple buffers, which has been proven to get better performance for applications on some multi-core architectures such as IBM CELL [11]. On IBM Cyclops64 manycore architecutre, as shown in Figure 9, the multiple-buffering techinque achieves better performace, too. However, we have to make a tradeoff between multiple buffers and memory size. With a more coarse-grained level of parallelism, more buffers are needed (one for each coarse-grain task). Therefore, on current manycore architectures with small memory size, a scalable parallel algorithm prefers to be fine-grained, where one buffer is shared amongst many more threads.
Through the detailed performance analysis, we observe that the performance of memory latency tolerace is determined by the size and number of on-chip buffers. However, due to the irregular behavior (i.e. the degree distribution), it is difficult to build a static performance model to determine an optimal performance by tunning the two buffer parameters. But it is optimistically expected that performance

can be achieved by dynamically scheduling the threads at runtime, however, this topic is out of scope of this paper. For these experiments we have selected the optimal results through tunning the two buffer parameters by hand.

5.2.2 Performance of Synchronization Optimization

One of the most important approaches to improve scalability of parallel algorithms is to reduce sychronization overhead. In the parallel betweenness centrality algorithm, synchronization overhead comes from (a). the barrier synchronization between two consecutive levels. (b). the lock synchronization when visiting the neighbor/predecessor vertices.

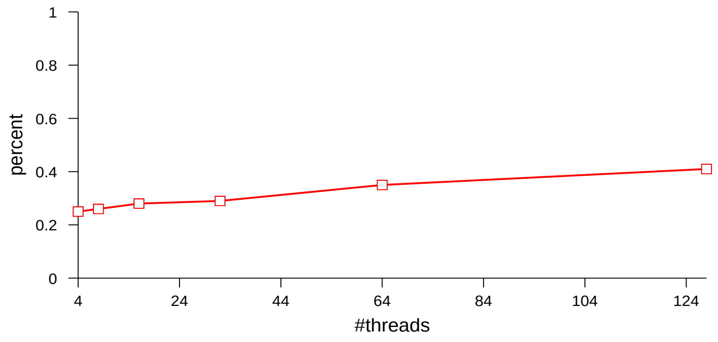


Figure 10: Overhead of barrier synchronization for scale = 10. The measured barriers include the barriers in both BFS and backtrace phase.

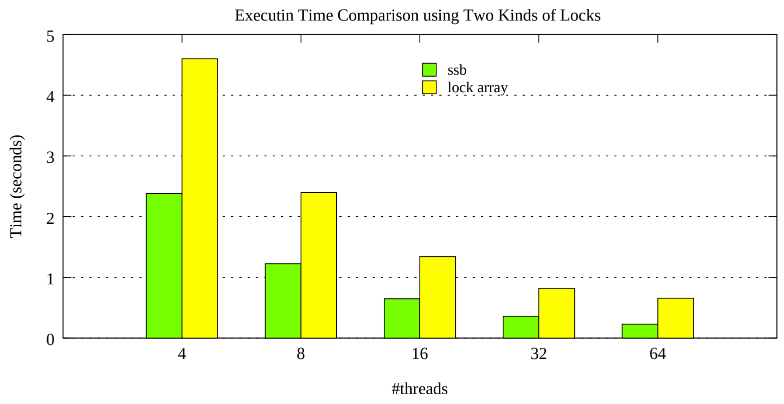


Figure 11: The execution time of the BFS phase using two kinds of lock synchronization mechanisms

A synchronization for a group of threads is used to keep consistency of the queue/stack after a level of neighbors/predecessors has been visited. The overhead of this level synchronization also relates to the degree distribution of the scale free graph, because the distribution of the number of edges per vertex is not uniform. Obviously, there is a maximal level synchronization if the degree of each vertex is only one. This work just implements the level synchronizaion using a barrier, not optimizing this kind of synchronization through algorithm and software. We measured the overhead of level synchronization in Figure 10. This figure plots the percent of barrier synchronization in the overall execution time. With an increasing number of threads, the overhead of barrier synchronization becomes a main performance bottleneck of the parallel algorithm. Note that our proposed parallel exploits multi-grain parallelism and the threads are partioned into serveal groups. Therefore, a barrier synchronization occurs among the threads within a group. In the experiments for figure 10, we keep the number of threads within a group constant, while the number of groups increases with the total number of threads. To a certain extent, this strategy can reduce the side effects of barrier algorithm itself, where the overhead of a barrier increases with number of threads involved with the barrier. Because each task assigned to each thread group is different, the number of barriers may be different, too. Like the case in multi-buffering, it also provides an opportunity to dynamically schedule at runtime. Meanwhile, it makes sense that a manycore architecture can support multiple hardware barrier for exploiting multi-grain parallelism.

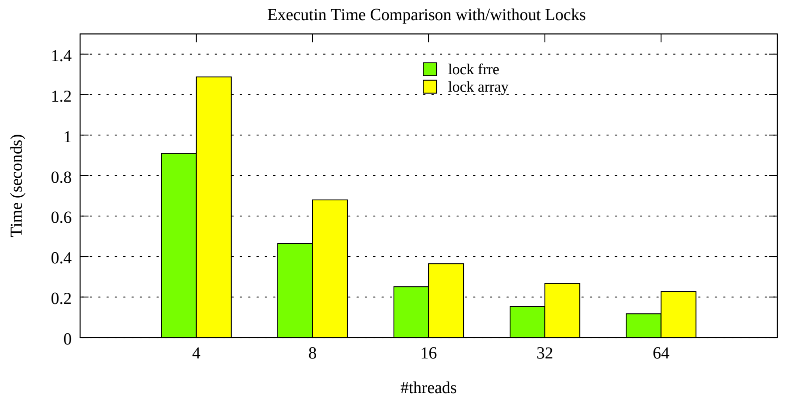


Figure 12: The execution time of the backtrace phase with/without lock synchronization

In the BFS phase, the fine-grain lock synchronization algorithm assigns an independent lock to each vertex. Thus, it avoids the additional data race in coarse-grain lock synchronizaiton, where all threads have to acquire a global lock even if they are visiting different vertices. In the fine-grain lock synchronization algorithm, a confilict of acquiring a lock occurs only if two threads are visiting the same vertex. An additional lock array, whose size is the same with the number of vertices, is used to implement the fine grain lock. However, due to require/release a lock, it results in additional memory accesses to the lock array. The access to lock array is also irregular because each lock is associated with each vertex. The fine-grain lock mechanism based on SSB, which associates locking functions with memory locations dynamically. Note that our optimizations for memory hierarchy, only parts of adjacent array are loaded into on-chip memory. Therefore, when a thread requires a lock, it accesses the associated memory location in the on-chip memory because the vertices are loaded into on-chip memory by *helper* threads, instead of accessing the memory locations in the off-chip memory. Figure 11 compares the performance of two kinds of fine-grain lock synchronization approaches. The fine-grain lock mechanism based on SSB reduce the execution time by two times. Comparing with the BFS phase, the backtrace phase consumes little running time, but our proposed lock free algorithm improves the performance further. Figure 12 shows the performance improvement of our proposed lock free algorithm.

5.2.3 Comparison with Other Architectures

In this paragraph, we compare the results obtained on C64 with that on other architectures. When we were doing this work, the platforms we can reach are Intel 4-way dual-cores Xeon SMPs and 40-processors Cray MTA-2. The program on SMPs is fine-grain parallelized using OpenMP by Bader et.al. The codes on Cray MTA-2 came from the personal commnication with John Feo in Cray Inc. Because the limitation of C64 simulator, we only can get the full results of experiments, where the scale is less than 10. Table 1 reports the *TEPS* metrics on the three platforms. It is not necessary to compare the performance on SMPs with more than 8 threads because the number of cores available is 8. Although the L2 cache size is 2MB on Xeon and it can contain the whole graph data structure for this small problem size, the peformace degrades with the increasing number threads because the lock synchronization is not optimized. The low performance on MTA-2 is caused by low utilization of thread streams for the small problem size. When the scale is more than 16, The metric *TEPS* increases greatly and scale well with the incrasing threads. In fact, We got the *TEPS* for $scale = 16$ and 4 threads on C64, it is two times less than that on MTA-2. This comparison indicates that the algorithms on MTA-2 has better *weak scaling*, but our algorithm has better *strong scalability*. This evaluation is to compare one C64 chip with mulitple MTA-2 processors, each of which has 128 streams. When the problem size is large enough (i.e. $scale \geq 16$), the streams is effectively utilized on a MTA-2 processor. However, the experiments on C64 only used a few thread units specified by user parameters. It is reasonable that the performance is lower on C64, while it indicates that our proposed parallel algorithms on C64 is comparable with that on the MAT-2 system.

Table 1: The comparison of TEPS on three platforms.

#threads	C64	SMPs	MTA-2
4	2917082	5369740	752256
8	5513257	2141457	619357
16	9799661	915617	488894
32	17349325	362390	482681

6 Conclusions

We have demonstrated an algorithmic reconstruction to achieve good performance on a manycore architecture. In order to utilize the large scale cores in a processor, we exploited the multigrain parallelsim in betweenness cetrality algorithm. This is a shift from traditional parallel computing, where parallelism is typically expressed in a single granlarity such as coarse grain parallelsim. Analogous to the term of dependence, in the levels of parallel betweenness cetrality algorithm, the coarse grain parallelsim is control parallelsim and the fine grain parallelism is data parallelism. This observation is important for a runtime to gain good performance through automatically exploiting multigrain parallelism. In fact,

Filip et.al. [8] presented a runtime system for dynamically scheduling multigrain parallelism on Cell processor and achieved a reasonable performance for RAxML algorithms [29].
Another important fact is that manycore architecture has an explict memory hierachy. A generic technique to improve the performance of memory access at algorithmic level is to orchastrate and schedule data through multi-threading. This requires a very careful algorithmic design that explicitly organizes the hierachy of data and movement between these levels. Obviously, it leads to more complex software developments.
The critical factor of fine grain parallelism is the performance of synhcronization. Although we can develop some lock free algorithms for some specific problems, it is more desirable that a manycore architecture can provide an effcient synchronization mechanism.
For the irregular computation like graph algorithms, we have to perform some nontrivial techniques to map a serial program to parallel computers. That is more important to shift to manycore era, where the performance is improved through scaling more parallelism. Although the algorithm and program development is suffering from the evolution to manycore architecture, it provide a great opportunity to achieve better performance for a class of irregular computation, which performance can not be improved in conventional architecture.

References

[1] http://grape-dr.adm.s.u-tokyo.ac.jp/system-en.html.

[2] http://www.cray.com/products/xmt/.

[3] http://www.openmp.org.

[4] The mta-2 multithreaded architecture. www.cray.com.

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

[6] David A. Bader and Kamesh 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.

[7] David A. Bader and Kamesh Madduri. Parallel algorithms for evaluating centrality indices in real-world networks. In *The 35th International Conference on Parallel Processing (ICPP 2006)*, 2006.

[8] Filip Blagojevic, Dimitrios S. Nikolopoulos, Alexandros Stamatakis, and Christos D. Antonopoulos. Dynamic multigrain parallelization on the cell broadband engine. In *the 2007 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2007.

[9] Ulrik Brandes. A faster algorithm for betweenness centrality. *Journal of Mathematical Socialogy*, 25(2):163–177, 2001.

[10] C. Cascaval, J.G. Castanos, L. Ceze, M. Denneau, and et. al. Evaluation of a multithreaded architecture for cellular computing. In *Proc. of 8th Intl. Symp. on High Performance Computer Architecture*, Boston, MA, 2002.

[11] Tong Chen, Zehra Sura, Kathryn O'Brien, and Kevin O'Brien. Optimizing the use of static buffers for dma on a cell chip. In *The 19th International Workshop on Languages and Compilers for Parallel Computing*, page 2006.

[12] Thayne Coffman, Seth Greenblatt, and Sherry Marcus. Graph-based technologies for intelligence analysis. *Communications of the ACM*, 47(3):45–47, 2004.

[13] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A parallelization of dijkstras shortest path algorithm. *Lecture Notes in Computer Science*, 1450:722–731, 1998.

[14] Juan del Cuvillo, Weirong Zhu, and Guang R. Gao. Landing openmp on cyclops-64: An efficient mapping of openmp to a many-core system-on-a-chip. In *The 3rd ACM International Conference on Computing Frontiers*, Ischia, Italy, 2005.

[15] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. Fast: A functionally accurate simulation toolset for the cyclops-64 cellular architecture. In *Workshop on Modeling, Benchmarking and Simulation (MoBS), held in conjunction with the Annual International Symposium on Computer Architecture (ISCA'05)*, 2005.

[16] Juan del Cuvillo, Weirong Zhu, Ziang Hu, and Guang R. Gao. Tiny threads: a thread virtual machine for the cyclops-64 cellular architecture. In *Fifth Workshop on Massively Parallel Processing (WMPP), held in conjunction with the 19th rnational Parallel and Distributed Processing System*, 2005.

[17] Antonio del Sol, Hirotomo Fujihashi, and Paul O'Meara. Topology of small-world networks of protein–protein complex structures. *Bioinformatics*, 21(8):1311–1315, 2005.

[18] Monty Denneau and Henry S. Warren, Jr. 64-bit Cyclops: Principles of operation. April 2005.

[19] Linton C. Freeman. A set of measures of centrality based on betweenness. *Sociomtry*, 40(1):35–41, 1977.

[20] Ananth Y. Grama and Vipin Kumar. A survey of parallel search algorithms for discrete optimization problems, 1993.

[21] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. Synergistic processing in cell's multicore architecture. *IEEE Micro*, pages 10–24, March 2006.

[22] Brian T. N. Gunney, Andrew M. Wissink, and David A. Hysom. Parallel clustering algorithms for structured amr. *Journal of Parallel and Distributed Computing*, 66(11):1419–1430, 2006.

[23] Hawoong Jeong, Sean P. Mason, Albert-Laszlo Barabasi, and Zoltan N. Oltvai. Lethality and centrality in protein networks. *Nature*, 411:41, 2001.

[24] P. N. Klein and S. Subramanian. A randomized parallel algorithm for single-source shortest paths. *Journal of Algorithms*, 25(2):205–220, 1997.

[25] Valdis E. Krebs. Mapping networks of terrorist cells. *Connections*, 24(3):43–52, 2002.

[26] Fredrik Liljeros, Christofer R. Edling, Luis A. Nunes Amaral, H. Eugene Stanley, and Yvonne Aberg. The web of human sexual contacts. *Nature*, 411:907, 2001.

[27] Joon-Sang Park, Michael Penner, and iktor K. Prasanna. Optimizing graph algorithms for improved cache performance. *IEEE Trans. Parallel Distrib. Syst.*, 15(9):769–782, 2004.

[28] John W. Pinney, Glenn A. McConkey, and David R. Westhead. Decomposition of biological networks using betweenness centrality decomposition of biological networks using betweenness centrality. In *the 9th conference on research in computational molecular biology*, 2005.

[29] A. Stamatakis, T. Ludwing, and H. Meier. Raxml-iii: a fast program for maximum likelihood-based inference of large phylogenetic trees. *Bioinformatics*, 21(4):456–463, 2005.

[30] Oreste Villa1, Daniele Paolo Scarpazza1, Fabrizio Petrini1, and Juan Fernandez Peinador. Challenges in mapping graph exploration algorithms on advanced multi-core processors. In *21th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[31] J. S. Vitter. External memory algorithms and data structures: Dealing with massive data. *ACM Comptuing Surveys*, 33(2):209–271, 2001.

[32] Andy Yoo, Edmond Chow, Keith Henderson, William McLendon, Bruce Hendrickson, and A Aatalyurek. A scalable distributed parallel breadth-first search algorithm on bluegene/l. In *ACM/IEEE Supercomputing'05*, 2005.

[33] Weirong Zhu, Vugranam C. Sreedhar, Ziang Hu, and Guang R. Gao. Synchronization state buffer: Supporting efficient fine-grain synchronization on many-core architectures. In *The 34th International Symposium on Computer Architecture*, 2007.