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

Analysis and Performance Results of Computing Betweenness Centrality on IBM Cyclops64

Guangming Tan†, Vugranam Sreedhar††, Guang R. Gao†

CAPSL Technical Memo 090
October 20, 2009

†Email: guangmin@capsl.udel.edu or tgm@ict.ac.cn,ggao@capsl.udel.edu
††Email: vugranam@us.ibm.com

Abstract

This paper presents a joint study of application and architecture to improve the performance and scalability of an irregular application – computing betweenness centrality – on a many-core architecture IBM Cyclops64. The characteristics of unstructured parallelism, dynamically non-contiguous memory access and low arithmetic intensity in betweenness centrality pose an obstacle to an efficient mapping of parallel algorithms on such many-core architectures. By identifying several key architectural features, we propose and evaluate efficient strategies for achieving scalability on a massive multi-threading many-core architecture. We demonstrate several optimization strategies including multi-grain parallelism, just-in-time locality with explicit memory hierarchy and non-preemptive thread execution, and fine-grain data synchronization. Comparing with a conventional parallel algorithm, we get 4X-50X improvement in performance and 16X improvement in scalability on a 128-cores IBM Cyclops64 simulator.

Contents

List of Figures

List of Tables

1 Introduction

Computer architects are exploring the massive many-core architecture space with the hope of improved execution of scientific applications. At a high level there are two kinds of applications– regular applications where data access and control flow follow regular and 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. Computing betweenness centrality (BC) [16] used in a network analysis is a good example of such irregular problems. BC is a popular quantitative index for the analysis of large scale complex network and measures the control a vertex has over communication in the network. It has been used extensively to build protein interaction network, identify key actors in terrorist network and study sexual/AIDS network. Brandes' algorithm [5] is a fast algorithm for computing BC. In this paper, we refer to BC algorithm as the one proposed by Brandes [5]. BC algorithm calculates the centrality through two steps: BFS (breadth first search) traversal and backtrace accumulation. Since most of the networks in a real world are represented as scale-free sparse graphs [1], BC algorithm exhibits *unstructured parallelism* and *dynamically non-contiguous memory access*. Moreover, especially for this kind of emerging applications in high performance community, another distinct characteristic is *low arithmetic intensity* – the ratio between arithmetic operations and memory operations.

With the advent of many-core architectures, such as IBM Cyclops64 [14] [36] contains hundreds of on-chip cores, it is extremely challenging to tackle the difficult problem of optimizing and scaling irregular applications. In fact, memory hierarchy with small on-chip memory per core in such an architecture makes the problem even more difficult. While there is no consensus on many-core architectures, it is important to identify not only how programmers will use the mechanisms provided in the emerging many-core architecture, but the relative usefulness of various mechanisms as evidenced by their impact on application performance. In this paper, we leverage some key features on a many-core architecture – IBM Cyclops64 to improve performance of computing betweenness centrality for scale-free sparse graph. In consideration of unstructured parallelism, dynamically non-contiguous memory access and low arithmetic intensity exposed by a large class of irregular applications, we identify four key properties of IBM Cyclops64 to address the challenge of executing irregular programs on such many-core architectures: *massive light-weight hardware thread units, non-preemptive thread execution model, explicit memory hierarchy* and *fine-grain data synchronization*. The main contribution of this paper includes:

- Being aware of massive light-weight hardware threads, we developed a fine-grained parallel algorithm by combining multi-level parallelism. The new algorithm is well-structured for load balance and locality optimization.

- We found that the properties of non-preemptive thread execution model and explicit memory hierarchy are useful to achieve *just-in-time locality*. We proposed a data-centric strategy to exploit just-in-time locality [19, 20, 32] for the fine-grained parallel algorithm.

- The restructured program takes good advantage of the architectural support of fine-grain data synchronization. Comparing with the parallel algorithm described in HPCS SSCA2 [2], the imple-

mentation on IBM Cyclops64 obtained a performance improvement of 4-50 times and scalability of 16 times.

To the best of our knowledge, this paper is the first indepth study of implementing a high performance BC program on a many-core architecture. The rest of the paper is organized as follows: In section 2, we describe betweenness centrality (BC) algorithm and its characteristics. In section 3, we introduce IBM Cyclops64 (C64) architecture. Section 4 discusses how to leverage the key properties of IBM C64 to re-structure BC algorithm. Section 5 evaluates the performance and section 6 discusses related techniques. Finally, section 7 concludes this paper.

2 BC Algorithm and Its Irregular Characteristics

In this section, we briefly describe the best sequential algorithm for calculating BC (for the detailed algorithm, refer to [5]). Then we examine its irregular characteristics, which identified by experimental results on commercial multi-core platforms.

2.1 Brandes Algorithm

Given a graph $G = (V, E)$ where V denotes the set of vertices and E the set of edges in G. Let σ_{st} denote the number of shortest paths from $s \in V$ to $t \in V$, where $\sigma_{ss} = 1$ by convention. Let $\sigma_{st}(v)$ denote the number of shortest paths from s to t that some $v \in V$ lies on. The BC measure of a vertex v is given by:

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

Given pairwise distance and shortest paths counts, the pair-dependency $\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}$ of pair $s, t \in V$ denotes the fraction of shortest paths between s and t that pass through a particular vertex v. The BFS traversal algorithm calculates all shortest paths. Let's denote $d_G(u, v)$ to be length of the shortest path between u and v, $w(u, v)$ is weight of edge (u, v). In the BFS traversal, the set of *predecessors* of a vertex v on a shortest path from source vertex s is generated:

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

To eliminate explicit summation of all pair-dependencies, Brandes' algorithm introduces a notion of the dependency of a vertex $s \in V$ on a single vertex $v \in V$, defined as:

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

The crucial observation is that these partial sums obey a recursive relation as presented in theorem 1. We omit a formal proof here, a reader refers to [5] for details.

Theorem 1 *The dependency of $s \in V$ on any $v \in V$ obeys*

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

In the backtrace accumulation, a partial bc value of a predecessor is accumulated according to its successors.

$$bc(v) = \sum_{s \neq v \in V} \delta_{s\bullet}(v) \tag{5}$$

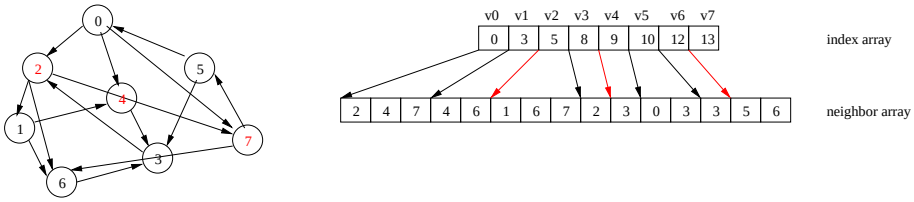


Figure 1: Adjacency array of a graph.

A space efficient data structure for sparse graph G is an indexed adjacency array data structure. Figure 1 shows an example, which is composed of an *index array* and a *successor array*. In fact, the predecessor set P recording the trace of BFS tree is stored in another adjacency array. The parameters d, δ, σ, 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 arrays of G, P.

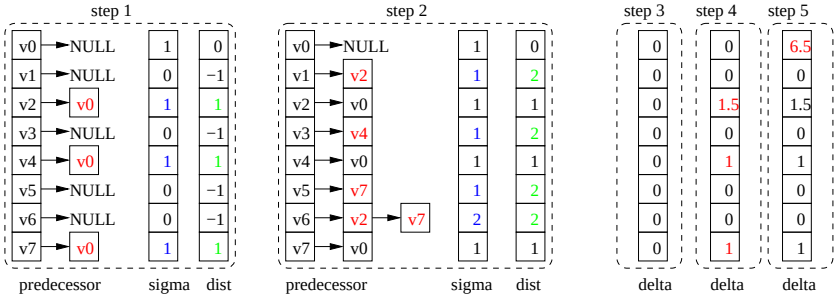


Figure 2: A demonstration of BC algorithm.

Figure 2 demonstrates an instance starting with vertex v_0 in Figure 1. Both *step 1* and *step 2* generate a BFS tree (*predecessor*) and record its information (σ and d). Then according to equation 4, the values of δ is accumulated along the BFS tree from *step 3* to *step 5*. At this time, the five steps compute the partial betweenness centrality values of all vertices. After the algorithm performs similar steps from all other vertices we get the final accumulated results.

2.2 Irregular Characteristics

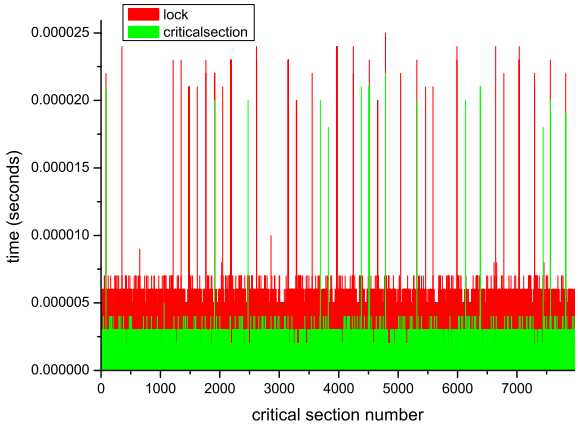


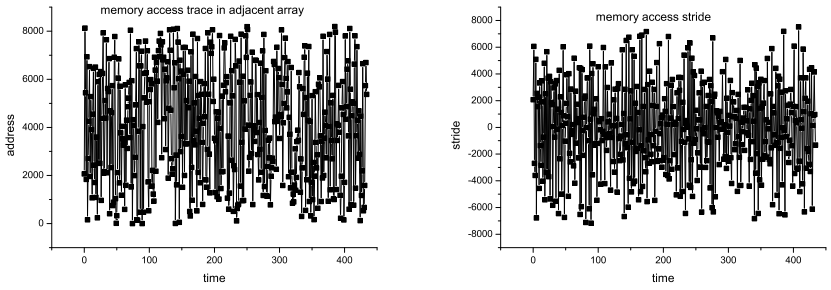
Figure 3: The execution time distribution of critical sections. An value in x-axis is a critical section number

Unlike regular applications where the inherent locality and parallelism are apparent and easy to be exploited, it takes careful understanding of the locality and parallelism behavior of irregular applications before one can achieve high performance and scalability. We summarize three important features of BC algorithm, which represent a large class of irregular applications' computation behaviours.

- *Unstructured parallelism.* For instance, during the BFS phase, a queue is used to maintain the current vertices being extended (In the following context, visiting neighbors of a vertex is referred to as an *extension* operation). The available parallelism within an extension operation is proportional to degrees of vertices. However, the degrees in a scale free graph obey a power-law distribution [1], which shows a high variance and means that most of vertices have low degrees. In the same level of BFS tree multiple extension operations may proceed in parallel if they do not share neighbor vertices. In Figure 1 the parallel extensions of v_2, v_4, v_7 require a synchronization mechanism. A fine-grained mutex lock associated with each vertex is one way to protect from conflict. Note that the number of locks scales with the number of vertices, which is often too huge to be held in memory. For a very fine-grained parallel algorithm on a many-core architecture, an additional concern is the very small critical section. Figure 3 compares each lock synchronization overhead to the span of each critical section (only plots about 8000 sections). The critical section is so small that the synchronization overhead dilates the size of critical section.

- *Dynamically non-contiguous memory access.* The effectiveness of existing locality optimization techniques such as prefetching and speculation rely on the continuity of the neighboring vertices and regular distances of different region of neighboring vertices in adjacency arrays. In a scale-free sparse graph, the degrees or neighbors of vertices are highly variable. Considering the simple example shown in Figure 1 again, we observe that not only the neighbor nodes of v_2, v_4 and v_7

are located in different regions in the adjacency array, but also the strides between the different regions are not constant. Figure 4(a) records the trace of accesses in the adjacency array, and Figure 4(b) depicts the distance between two consecutive accesses to neighbor regions. Also, the references to other linear arrays d, δ, σ, bc have the similar behaviors. Such non-contiguous memory access pattern cannot benefit from current prefetching or speculation techniques.



(a) (b)

Figure 4: Memory access pattern in the adjacent array

- *Low arithmetic intensity.* The profiling of BC program execution reports that BFS traversal is the most time consuming. Looking at equation 2, 3, an extension of one vertex needs two arithmetic (float point addition) operations, six memory operations. Thanks to high arithmetic intensity and parallelism, most of traditional scientific computing programs naturally benefit from many-core. However, in order to improve the performance of a memory bound program like BC, the key to a successful parallel program will be an efficient strategy to reduce the memory access overhead by utilizing the massive parallel thread units.

HPCS benchmark suite SSCA2 [2] specifies an OpenMP implementation of BC algorithm. Figure 5 reports its performance on IBM C64. As the number of threads is increased, the scalability and performance degrade. In order to achieve high performance on such many-core architectures, it is important to identify the characteristics impacting on application performance.

3 IBM Cylops64 Architecture

IBM Cyclops64 (C64) is a many-core architecture designed to serve as a dedicated petaflop computing engine for running high performance applications. In this section we describe its architecture, highlighting some of its core features that we exploit in improving the performance and scalability of irregular applications.

- *Massive light-weight hardware thread units.* A C64 chip employs a multiprocessor-on-a-chip design with 160 hardware thread units, half as many floating point units, embedded memory, an interface to the off-chip SDRAM memory and bidirectional inter-chip routing ports. On-chip

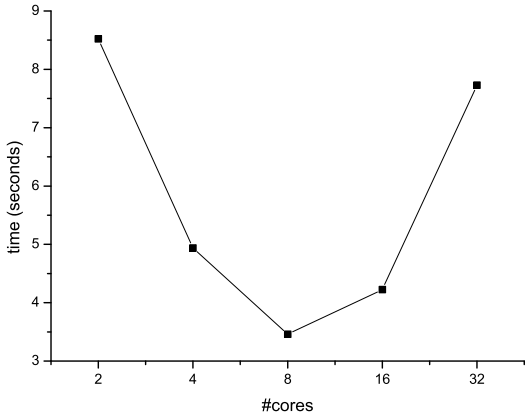


Figure 5: The performance of OpenMP implementation – HPCS SSCA2. The number of vertices and edges is 1024 and 8192, respectively.

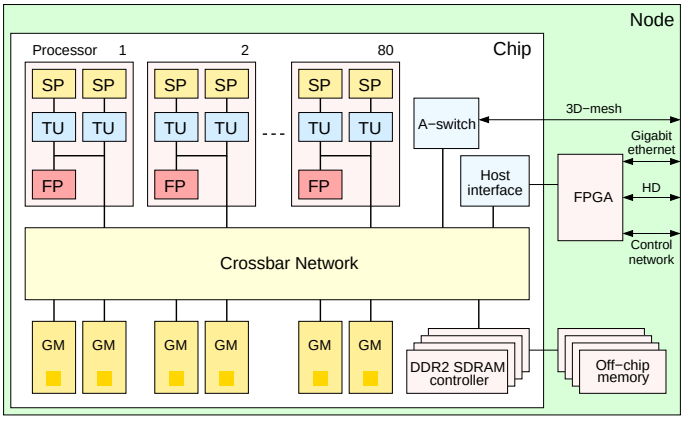


Figure 6: IBM Cyclops64 chip architecture

resources are connected to a 96-port crossbar network, which sustains all the intra-chip traffic communication. In regard to intra-chip communication bandwidth, each processor within a C64 chip is connected to a crossbar network that can deliver 4GB/s per port, totaling 384GB/s in each direction. The bandwidth provided by the crossbar supports intra-chip communication, i.e. access to other processor's on-chip memory.

- *Explicit memory hierarchy.* C64 chip has no data cache and features a three-level (scratchpad memory, on-chip SRAM, off-chip DRAM) memory hierarchy. A portion of each thread unit's corresponding on-chip SRAM bank is configured as the scratchpad memory (SP). Therefore, the thread unit can access to its own SP with very low latency through a backdoor. The remaining sections of all on-chip SRAM banks that together form the on-chip global memory (GM) that is uniformly addressable from all thread units. There are 4 memory controllers connected to 4 off-chip DRAM banks.

- *Non-preemptive thread execution model.* C64 incorporates an efficient support for thread level execution. For instance, a thread can stop executing instructions for a number of cycles or indefinitely; and when asleep it can be woken up by another thread through a hardware interrupt. All the thread units within a chip connect to a 16-bit signal bus, which provides a means to efficiently implement barriers. C64 provides no resource virtualization mechanisms: the thread execution is *non-preemptive* and there is no hardware virtual memory manager. The former means the OS will not interrupt the user thread running on a thread unit unless the user explicitly specifies termination or an exception occurs. The latter means the three-level memory hierarchy of C64 chip is *visible* to the programmer.

- *Fine-grain data synchronization.* C64 provides a *synchronization state buffer (SSB)* to support fine-grain data synchronization (refer to [36] for details). SSB is a small buffer attached to the memory controller of each memory bank. It records and manages states of active synchronized data units to support and accelerate word-level fine-grain synchronization. SSB avoids enormous memory storage cost and high memory access latency. The structure of SSB is shown in Figure 7. Each SSB entry consists of four parts: 1) address field that is used to determine a unique location in a memory bank, 2) thread identifier, 3) an 8-bits counter and 4) a 4-bits field that supports 16 different synchronization modes. SSB mechanism uses instructions of *ssb_lock/unlock* to implement fine-grain synchronization.

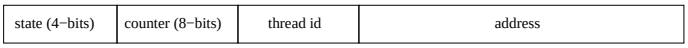


Figure 7: The structure of SSB entry

4 Mapping BC Algorithm to IBM Cyclops64

In this section, we discuss betweenness centrality with graph traversal and show how to map it to IBM C64 by leveraging the architectural features. As noted in section 2, due to highly variable degrees and data dependence, first, both low arithmetic intensity and unstructured parallelism lead to the low utilization of massive hardware units. Second, both high memory storage cost and dynamically non-contiguous memory access patterns incur high overhead of off-chip memory accesses. Our strategy combines the algorithmic re-structure with several key architectural properties:

1. *Greedy parallelism.* Since the underlying many-core architecture provides massive hardware thread units and efficient fine-grain data synchronization, we try to exploit as much parallelism as possible in application programs. Therefore, we develop a fine-grained parallel BC algorithm, which exploits multiple levels of parallelism simultaneously. With the fine-grained parallelism, it is easier to schedule the parallel tasks for better load balance. Further, it opens a door to achieve just-in-time locality (In fact, it is an extra source of fine-grained parallelism), which is proven to be critical to high performance on a C64-like many-core architecture in this work.

2. *Just-in-time locality.* Like a cache-based architecture, it is desirable to schedule most of threads to access low latency on-chip local storage. Due to dynamically non-contiguous memory access, traditional prefetching and speculation techniques are hard to take effect. We identify that the architectural characteristics of explicit memory hierarchy plus non-preemptive thread execution model make just-in-time locality [19, 20, 32] possible. Although the intrinsic data dependence (producer-consumer in BC algorithm) can not be eliminated, we may decouple computation operations with memory operations so that an additional parallelism between them is exploited. A more important fact is that the decoupled operations are scheduled to run according to data-centric mechanism. This approach is reasonable because data movement is under the control of programmer and threads processing the data are not preemptive, computation threads consume *just-in-time locality* produced by memory threads, that is, data are local to computing cores just before the cores start to process the data.

Recall that there are two phases in the BC algorithm: BFS traversal and backtrace accumulation (See section 2). Both phases have similar computing behavior (although the backtrace accumulation is of high arithmetic intensity, our optimization strategies still work). To simplify presentation we only describe the optimization for BFS traversal phase.

4.1 A Fine-grained Parallel Algorithm

BFS traverses a graph to generate a tree (subgraph) level by level. At current level each vertex is extended to produce the vertices at the next level. We observe two levels of parallelism: among all extensions and in an extension (the term of extension is defined in section 2). First, if multiple vertices at one level do not share the same neighbor vertices, these extension operations can be assigned to multiple parallel thread units. For example in Figure 1 v_2, v_4, v_7 are at the same level, the extensions of these vertices are finished by three parallel threads. However, if two vertices share the same neighbor vertices, a synchronization is forced to keep the shared neighbors being visited for just one time. Second, in an extension, the explorations on each neighbor vertex can be embarrassingly parallelized. For example the explorations on v_2, v_4, v_7 of vertex v_0 are assigned to three parallel threads. However, there are two factors hindering the scalability of parallelism. On one hand, the first level of parallelism requires concurrent memory accesses to their neighbor vertices, the memory size of which may exceed the small on-chip memory on many-core architectures. Besides, the intensively concurrent memory accesses place a burden on the limited off-chip bandwidth, which slacks the scalability of parallelism. On the other hand, the scale-free sparse graph has few vertices with high degrees, which determines the available parallelism. Therefore, the previous parallel implementations [2–4], which only exploit either one level of parallelism, can not achieve good performance on massive multi-threading architectures.

Let us denote the set of the extended vertices in current level as $V_i = \{v_{i1}, v_{i2}, ..., v_{ik}\}$. Let $N_j = \{u_{j1}, u_{j2}, ..., u_{jk_j}\}, 1 \leq j \leq k$ denote the neighbor vertices set of a vertex v_{ij}. During execution the unvisited neighbor vertex u ($d[u] = -1$) is 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 $P[u]$. We *logically* compact all their neighbors into one large set: $UN_i = \bigcup_{1 \leq j \leq n} N_j$, then partition it among parallel threads. In the

case of ignoring shared neighbor vertices, the compaction achieves at least $p = \frac{|UN_i| = \sum_{j=1}^{k} |N_j|}{max_{j=1}^{k} |N_j|}$ more times of parallelism than either one level of parallelism. For example in 1 we compact the neighbor vertices of v_2, v_4, v_7 into a larger set of $v_1, v_6, v_7, v_3, v_5, v_6$. The tasks of visiting the 6 neighbor vertices are evenly distributed to multiple threads. Because the working set of an extension operation depends on the highly variant degrees of vertices, the previous parallel algorithm [2] only schedules multiple extension operations so that load balance is hard to achieve. Thanks to the support of massive fine-grained parallel thread units, we combine and re-distribute the parallel tasks of fine-granularity.

However, in the initial fine-grained parallel BC algorithm there are two problems to be addressed:

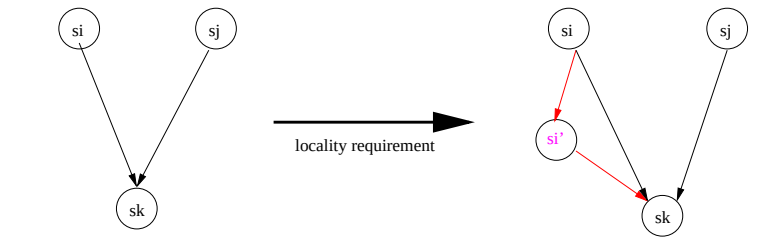
- The parallel algorithm achieves p times of parallelism at the cost of concurrently accessing p times of memory addresses. On IBM C64 the local storage is too small to hold the entire combined neighbor set, therefore a large number of high latency off-chip memory accesses happen. Much worse, the memory access patten is dynamically non-contiguous, it is difficult to adopt either perfetching or speculation automatically. Meanwhile, the concurrent accesses make the contention of the limited off-chip bandwidth worse.

- Because the operations on one vertex are involved with only two arithmetic operations, the critical section protected by synchronization operations is so small that the synchronization overhead dilates the size of critical section. An efficient synchronization mechanism SSB on IBM C64 may help. However, as noted in [36], SSB performance will degrade if the overflow of SSB happens when a synchronization operation is taken over by software. Therefore, we should reduce the number of conflicts at an instance to avoid SSB overflow.

4.2 Achieving Just-In-Time Locality

In the preliminary version of the fine-grained parallel BC program, we observe amount of off-chip memory accesses. In fact, the access to on-chip local storage has much higher bandwidth and lower latency than that to off-chip memory. Therefore, it is reasonable to schedule as many threads as possible only to access on-chip local memory space. Because both on-chip and off-chip memory are addressed by all threads in a uniform space with different latency, in a conventional execution model a thread is activated as soon as its data/control dependencies are satisfied, regardless of where the data are. Such a thread execution model does well for regular applications, where there is an inherent cache/memory locality. Unfortunately, irregular applications like BC program often have dynamically non-contiguous memory access. Note that C64 is configured with explicit memory hierarchy and non-preemptive thread execution model. Programmers can explicitly state where the data are in explicit memory hierarchy. Non-preemptive thread execution model forces a thread to finish consuming its data without re-scheduling. Based on these architectural properties, programmers can specify the exact relationships between a thread execution and places of its data. Based the features of IBM C64, we propose a *data-centric* strategy to achieve just-in-time locality for dynamically non-contiguous memory accesses.

We represent a program as a directed acyclic thread graph, where each node is a thread, and a direct arc between two nodes represents a precedence relation between threads (See Figure 8). In a



(a). Generic thread execution graph (b). A thread exeuction graph with locality dependence

Figure 8: thread execution graph

thread graph, a node s (i.e., a thread) is enabled if all its predecessor nodes have completed and the required data and control dependence have been satisfied. We call a thread that satisfies both data and control dependence requirements as being *logically enabled*. In order to achieve just-in-time locality for a thread execution, it is not sufficient for a logically enabled thread to run. We introduce locality constraint in addition to data and control dependence requirements to overcome the latency gap through memory hierarchy. Using locality constraint a logically enabled thread often cannot immediately run since the data may still be in off-chip memory hierarchy or in the local memory of other cores. All data referenced by the thread should become local before a thread can begin execution. We call a logically enabled thread as *locality enabled* if it also satisfies locality constraints (See Figure 8). The locality requirements ensure that the corresponding data of the candidate thread are resident in the same level of memory hierarchy where it is to be enabled. The stronger constraint on thread execution is data-centric, that is, the local data enables a thread execution.

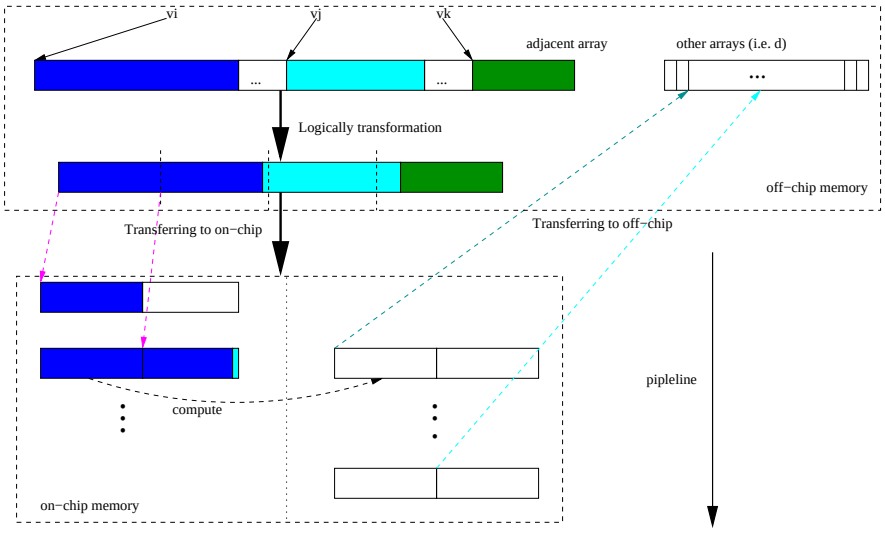


Figure 9: A demonstration of the parallel pipelining process for the BFS phase of the BC algorithm. For readability, the transformation (linearization/scatter) and data movement are depicted as two steps. A real implementation finishes them in single step by memory threads.

This strategy results in additional operations for "creating" locality constraint. Note that the massive hardware thread units on IBM C64 and low arithmetic intensity of BC program, we separate several threads to complete locality constraint. Meanwhile, the computation operations and memory operations are decoupled so that the parallel program is mapped to such a thread execution model. Within memory hierarchy, the memory operations may involve either collecting the data toward the cores where the thread is enabled, or sending/migrating the data away from the cores. Since most architectures (including many-core architectures) are designed to exploit "linear locality" [1], it is important to transform non-linear locality into linear locality just in time for the computation. For instance, consider the example shown in Figure 1, and assume that v_2, v_4 and v_7 are currently in queue, and assume we extend (during BFS) node v_2 to bring in nodes v_1, v_6 and v_7. Since these three nodes are contiguous we exploit the locality among them and arrange them in a linear contiguous memory (in-core memory). However, neither $d[1], d[6]$, $d[7]$ nor $\sigma[1], \sigma[6]$, $\sigma[7]$ is contiguous, if we performed linearization to these discrete memory locations just before they are used to compute, then we achieve the "created" spatial locality. In an implementation, programs do *not* explicitly perform such linearization operations in off-chip memory, but naturally complete them during data movement through memory hierarchy. For example, a memory thread, which transfers data from off-chip memory to on-chip memory, consists of computing the start address and the size of the neighbor vertices region in adjacency array of each vertex, and collecting neighbor vertices dispersed in the off-chip memory address (adjacency array) into a contiguous on-chip memory address. It also collects the corresponding elements in d, σ into a contiguous on-chip memory address. Notice that there is a producer-consumer relationship between the collection of neighbor vertices and that of d, σ. Also, the memory references of d, σ are discrete because the distribution of the neighboring vertices obeys a law of power in a scale-free graph. Once computing the relevant information we write them back to off-chip memory using yet another memory thread.

In order to tolerate the overhead of "creating" locality constraint, we exploit parallelism between computation threads and memory threads. The multi-grain parallel algorithm partitions union set UN_i into multiple sub-blocks. When computation threads are processing the data in block i, some memory threads gather the data in block $i + 1$ and other memory threads scatter the results that are generated using the data in block $i - 1$. The threads operating multiple sub-blocks form a pipeline, which achieves just-in-time locality for each sub-block (See Figure 9). Figure 10 and 11 describe a pesudocode example of BFS with and without percolation, respectively.

4.3 Using Architectural Support of Fine-grain Data Synchronization

Our previous work on IBM C64 [10, 36] indicates that lock-based synchronization is better than lock-free one for explicit memory hierarchy. In fact, since there is neither priority inversion nor convoying problem in IBM C64, performance and memory contention are the only factors of a lock-free data structure. For lock-free synchronization [22] in parallelizing betweenness centrality, due to irregular memory access pattern, we observed many failures of speculation and rollback. However, with software lock mechanism, we have to use an additional lock array to assign one lock to each vertex. The size of

[1] We use the term linear locality to mean that data access have constant strides and for contiguous accesses the strides have one unit value, typically one word length.

```
1  BFS(int v) {
2    int dv = d[v]; //length of the shortest path
3    int sigmav = sigma[v]; //the number of the shortest path
4    for (i = 0; i < NumEdges[v]; i++) {
5      w = Adjacent[index[v]+i];
6      if (d[w] < 0) {
7        d[w] = dv + 1;
8        sigma[w] = 0;
9      }
10     if (d[w] = dv + 1)
11       sigma[w] = sigmav + 1;
12   }
13 }
```

Figure 10: The sequential BFS codes without percolation.

lock array is the same with the number of vertices, which is usually huge in real world. Thus, a large scale graph generates amounts of irregular off-chip memory accesses since these accesses are associated with that of vertices.

There are two advantages of the proposed parallel pipelining algorithm : 1). We explicitly separate computation threads from memory ones. The computation threads and memory ones access different memory locations at any instance t implemented by double-buffering. 2). The algorithm accesses the arrays in a chunking way, that is, in each pipelining stage, only small blocks reside in on-chip memory at any instance t. Note that the on-chip memory of C64 is organized in multiple banks way where each process is associated with a memory bank. Let $N = M \times B$ be the number of memory locations, where M is the size of each memory bank and B is the number of memory bank. At any instance t, let $S(t)$ be the amount of synchronization by all threads. Since the two features of the algorithm and the number of active threads $T \ll N$, the program is easily adaptive to satisfy an important constraint:

$$S(t) \ll N \tag{6}$$

Therefore, at any instance only a small fraction of memory locations are actively participating in synchronization. This observation exactly satisfies the condition of no overflow in SSB [36].

5 Evaluation

In this section we report experimental results and show the architectural and algorithmic impact on program performance, then summarize several implications on many-core architecture and programming.

5.1 Methodology

We evaluate the performance characteristics of mapping approaches on a cycle-accurate IBM C64 simulator [11] for the proposed parallel BC program. The architectural parameters used in the experiments

```
1  /*three pipelined phase: (1) off-chip memory read; (2) compuation (accessing on-chip memory);
2  off-chip memory write.*/ BFS(int v) {
3    int offset = 0;
4    int turn = 0;
5    int dv = d[v];
6    int sigmav = sigma[v];
7    SPAWN_TASK{
8    for (i = 0; i < buffsize; i++)
9      buff[turn][i] = Adjacent[index[v]+offset+i];
10   offset += buffsize;
11   turn ^= 1;};
12   BARRIER_WAIT();
13   while (offset < NumEdges[v]) {
14     //1. off-chip memory read
15     SPAWN_TASK{
16     for (i = 0; i < buffsize; i++)
17       buff[turn][i] = Adjacent[index[v]+offset+i];
18     offset += buffsize;
19     turn ^= 1;};
20     SPAWN_TASK{
21     for (i = 0; i < buffsize; i++) {
22       w = buff1[i];
23       buff2[turn][i] = d[w];
24       buff3[turn][i] = sigma[w];
25     }
26     turn ^= 1;};
27     //(2). compuation (accessing on-chip memory);
28     SPAWN_TASK{
29     for (i = 0; i < buffsize; i++) {
30       if (buff2[turn][i] < 0) {
31       buff2[turn][i] = dv+1;
32       buff3[turn][i] += 0;
33     }
34     if (buff2[turn][i] == dv+1)
35       buff3[turn][i] += sigmav;
36     }
37     turn ^= 1;};
38     //(3). off-chip memory write
39     SPAWN_TASK{
40     for (i = 0; i < buffsize; i++) {
41       w = buff[turn][i];
42       d[w] = buff2[i];
43       sigma[w] = buff3[i];
44     }
45     turn ^= 1;};
46     BARRIER_WAIT();
47    }
48 }
```

Figure 11: BFS codes with percolation on IBM C64. *SPAWN_TASK* is implemented as dispatching a thread from a thread pool. *BARRIER_WAIT()* is a global barrier synchronization supported by hardware mechanism.

are summarized in Table 1. The toolchain consists of an optimized GCC compiler, a thread execution runtime systems TNT [12] (Pthread-like) and a TNT-based OpenMP [10]. By modifying HPCS SSCA2 benchmark [2], the proposed parallel algorithm is implemented using the TNT library. The TNT runtime always maintains as many threads as the cores.

Table 1: Simulation parameters of C64.

Component	# of units	Params./unit
Threads	128	single in-order issue, 500MHz
FPUs	64	floating point/MAC, divide/square root
I-cache	16	32KB
SRAM (on-chip)	128	32KB (20 cycles load,10 cycles store)
DRAM (off-chip)	4	256MB (36 cycles load,18 cycles store)
Crossbar	1	96 ports, 4GB/s port

We report experimental results only for small problem sizes. Except for the limitation of simulator itself, IBM C64 is devised as an accelerating engine for building a Petaflops supercomputer, and there are massive IBM C64 nodes in the system. In a massive parallel algorithm the working-set in each node may be usually small. 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 and the shared on-chip memory is still small. 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. It is also reasonable to evaluate performance of small size of problems on a simulator.

5.2 Results for Mapping Parallel BC Algorithm

In this section we present our experimental results. The experimental data sets are generated by the program in HPCS SSCA2 benchmark. We represent the problem size in term of *S*, where the number of vertices is $n = 2^S$. At first glance, we summarize incremental optimization results of the parallel algorithm for just-in-time locality and synchronization using SSB. Figure 12 depicts 4-50 times reduction of execution time by comparing with the naively ported HPCS SSCA2.

Next we focus on four different performance characteristics. First we focus on performance and scalability as we increase both the problem (graph) size and the number of threads. Second we focus on understanding locality and memory latency as we increase the number of threads. Third we focus on the effect of barrier synchronization on the performance. Lastly, we present the performance improvement by SSB lock synchronization.

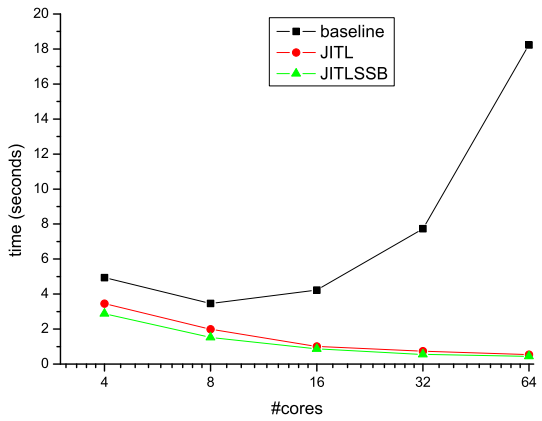


Figure 12: The incremental optimization results (S = 10). baseline: OpenMP version of HPCS SSCA2. JITL: SSCA2 with just-in-time locality. JITLSSB: SSCA2 with both just-in-time and synchronization state buffer.

- Figure 13 illustrates the performance and scalability as we increase the number of threads for three different scales (i.e., the problem size). We refer to the number of traversed edges per second (TEPS) as a performance metric, i.e. $TEPS = \frac{n*E(n)}{T(n)}$, where n is the problem size. Comparing the result with the OpenMP implementation, we can see that our optimization strategy shows a 16 times improvements of scalability. Using our approach we achieve almost linear speedups for all test cases when the number of threads is less than 32. For the test case with a problem size $S = 8$, the performance stops increasing when the number of threads reaches 128 because the number of available parallel sub-tasks is less than the number of hardware thread units. However, we improve the performance when the problem size is increased, i.e for $S = 9$ and 10. For BC algorithm the degree of a vertex determines the amount of parallelism that we can exploit. Although the multi-grained parallel algorithm reduces the number of idle threads, the maximum degree of a vertex is 64 for problem size $S = 8$. So the available parallelism for this small problem size still leads to a little improvement on 128 threads. For $S = 9$ and 10, where the maximum vertex degrees are 94 and 348, the performance and scalability are further improved.

- Figure 14 shows the effect of memory latency tolerance using the technique for creating just-in-time locality. Recall that the main purpose for creating just-in-time locality is to transform non-linear off-chip memory access to linear on-chip memory access in such a way that the overhead of the transformation is hidden. The implementation on C64 uses on-chip double buffers to hide the off-chip memory latency. The memory threads are used to transfer data between two memory levels. The overlapping of memory operations and computation operations is important to achieve high performance. In order to figure out the overlapping time, we profiled the execution time of computation and memory operations. Although the computation only access on-chip memory, the overall execution time of computation tasks is more than that of the memory tasks due to synchronization that is required among the computation tasks for computing the shortest

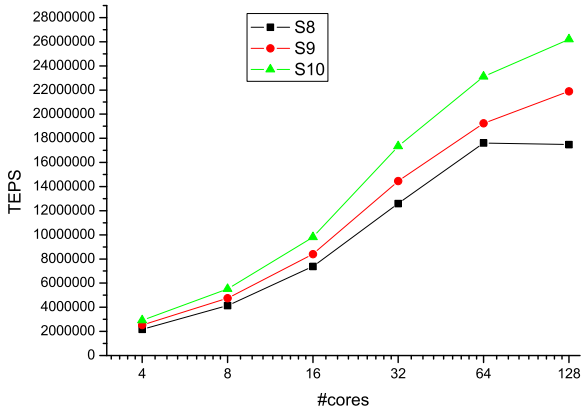


Figure 13: *Scalability* results of the parallel betweenness centrality algorithm (higher is better). The number of vertices $n = 2^S$, $E(n) = 8n$.

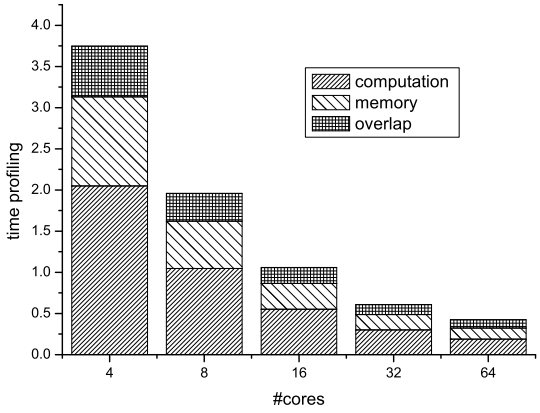


Figure 14: Time distribution and achieving off-chip memory latency tolerance.

path information. Next we wanted to understand the effect of increasing buffer size used in the parallel pipeline on the overall scalability. Interestingly increasing the buffer had little effect on the scalability. Note that degrees of most of vertices in scale-free graph are low so that we can not hide more off-chip memory access by increasing the buffer size. Figure 15 shows that increasing buffer size does not achieve better performance. Our strategy to hide memory access is pipelining between computation and memory operations. The larger buffer can contain more vertices, then the length of pipeline becomes so short that the memory operations can not be hidden.

- When implementing the parallel pipelining algorithm, we insert a barrier synchronization operation at the end of each pipeline. The overhead of a barrier is determined mostly by load balance and the number of barriers. Recall that the algorithm loads the adjacency array into the on-chip buffers one block at a time, it is important to note in the BC algorithm that the computation be-

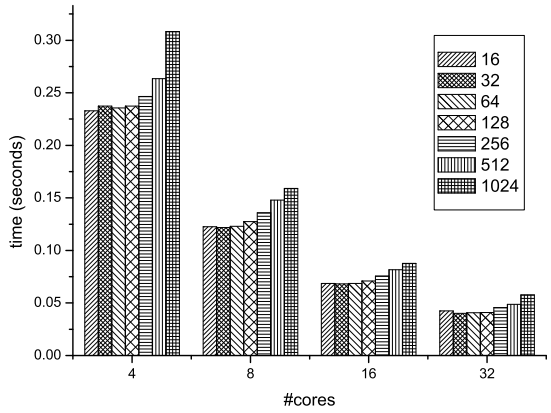


Figure 15: The comparison of running time using different sizes (bytes) of buffers.

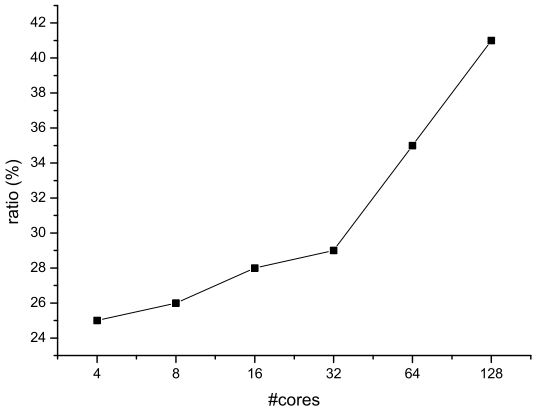


Figure 16: Overhead of barrier synchronization for scale = 10. The measured barriers include the barriers in both BFS and backtrace phase.

havior of each vertex may be different. For example, if a vertex is not one of the predecessors of a neighbor that is currently loaded into the on-chip memory, we do not have to insert this vertex into the predecessor set of the neighbor (otherwise we unnecessarily incur several additional memory accesses). Therefore, the execution time of each block may be different and so workload may not be balanced among multiple threads. Also, we cannot achieve a perfect overlap between computation and memory tasks at every stage of the pipeline. On the other hand by increasing the number of tasks, the workload on each thread decreases so that the difference of the workloads is not so significant, and we achieve more overlapping time. Unfortunately such fine grain partition may increase the depth of the pipeline and the number of barrier synchronization. Figure 16 illustrates the percent of overhead of barrier synchronization with respect to the overall execution time.

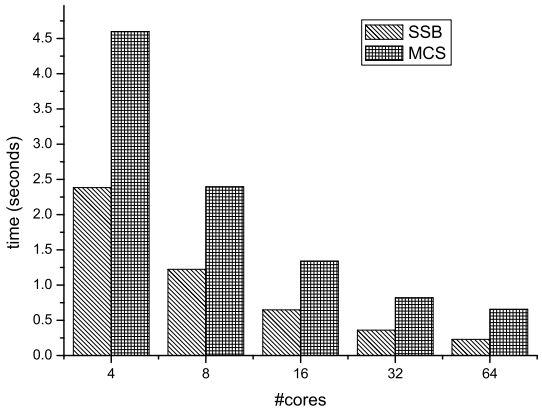


Figure 17: The comparison of software lock and SSB (BFS phase)

- Since in our parallel algorithm we only access a small portion of data during the computation phase to create just-in-time locality, only a small portion of memory participate in data synchronization. Using SSB for data synchronization seems very effective. As a reference, we implemented a highly optimized MCS [26] algorithm using in-memory atom operations on C64 [10,14]. As shown in section 3 for C64 architecture, each core accesses his own SP with very low latency. Thus, we use it as a "local memory" in MCS algorithm. Figure 17 compares the performance of the parallel programs with MCS lock to that with SSB. SSB further reduces the execution time and is very effective for the parallel algorithm.

5.3 Discussion

In order to evaluate our joint study of architecture and algorithm, we also compare the proposed parallel BC algorithm with both HPCS SSCA2 [2] on a Intel 4-way dual-cores Xeon SMP (8-processors) and a highly optimized implementation on Cray MTA-2, which is available through a personal communication with John Feo (previously in Cray Inc.). The MTA-2 is configured with 40-processors. Table 2 reports the TEPS performance on the three platforms. Although the L2 cache size of the SMP is 2MB, which can contain the whole graph data structure for the small problem size, the performance still is low because an efficient lock synchronization is unavailable. The low performance on MTA-2 is caused by low utilization of thread streams for the small problem size. In fact, we observed a sub-linear scalability on MTA-2 when the problem size is large enough (i.e., $scale = 22$), but the performance on the SMP is poor [13] (Unfortunately, due to limitation of simulator, we can not run so large test sets for IBM C64). This comparison indicates that the algorithm on MTA-2 achieves good weak scalability, but our algorithm on IBM C64 achieves better strong scalability because we unearth more additional parallelism even for small problem sizes.

Although this paper focuses on the performance and scalability of an on-chip many-core architecture, we also give a prospect of performance and scalability of the proposed algorithm on a multi-

Table 2: The comparison of TEPS on three platforms. $scale = 10$.

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

chips system connected with high latency network since IBM C64 will serve as a compute engine in a Petaflops supercomputer. Because a real system or simulator of multi-chips is not available, we present a qualitative analysis of the parallel algorithm. Note that the just-in-time locality decouples computation with memory operations, then the percolation strategy use some helper threads to hide latency to off-chip memory. This idea can be naturally extended to a multi-chip system with high latency network. Therefore, we expected that scalability of a system with multiple IBM C64 chips is comparable with that of Cray MTA-2.

In this paper we present the results for one case of computing betweenness centrality, however, it represents a class of general applications with irregular memory access, 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:

- A performance critical application with irregular memory access prefers to no-cache mechanism memory hierarchy. Hardware-managed memory (cache) automatically exploit locality in programs. The irregular memory access pattern in BC sets an obstacle to the capability of cache and incurs a large number of cache misses which hurt the memory bandwidth. For many-core architectures, an increasing gap between the number of cores and bandwidth is a serious obstacle to scalability of a parallel program. Configured with explicit memory hierarchy, IBM C64 provides an architectural support to programmer for precisely orchestrating memory movement for just-in-time locality at algorithmic level using multiple simple hardware thread units.

- Architectural support of fine-grain synchronization is reasonable. In a fine-grain parallel program on a many-core architecture, the overhead of synchronization is more sensitive since working-set of each thread is small. Using software synchronization it becomes worse because the memory access in a irregular program is unpredictable. The SSB on IBM C64 is proven to be favorable.

- A runtime system supporting programmers to utilize just-in-time locality is promising. The experimental results show that achieving just-in-time locality in programs is an efficient alternative technique for developing high performance algorithms on many-core architectures. In algorithmic level, programmers separate memory from computation and pipeline multiple memory-computation stages. In parallel programming model, users specify the tasks and their dependence, a runtime system could parse the task graph and automatically determine the granularity of decoupling and a way of pipelining so that the program would be optimally adaptive to memory hierarchies. Another advantage of the runtime system may be to provide virtualization of non-

preemptive execution model, which is one of the conditions to just-in-time locality on IBM C64. The resource virtualization is important for easy programming.

6 Related Work

Due to the importance of computing betweenness centrality, there have been several works on parallelization on conventional parallel architectures [2–4]. These parallel programs exploited inherent parallelism and solved a large scale graph on several parallel computers with huge memory storage. Our work focused on optimizing the irregular memory access using multi-threading many-core architectures with small on-chip memory, which propose different challenges on parallelizing a performance critical application. On the other hand, our work paid more attention to a joint study of architecture and algorithm. It is helpful to give some implications on many-core architecture design in the future.

The main contribution of this work is reflected in successfully mapping an important irregular application to an emerging many-core architecture IBM Cyclops64. Although our work focus on a joint study of algorithm and architecture, not a programming tool, the BC program represents a large class of irregular applications, thus the optimization techniques here may be applied to other programs on a many-core architecture. Therefore, we want to clarify the connection between our optimization strategy and the previous techniques.

- *Percolation:* Our approach achieving just-in-time locality is data-centric, and it shares the same point with percolation, which was briefly discussed by Gao in the context of HTMT project [19, 20]. In Gao's work a percolation process was proposed to pack the code and data into a tiny thread. Since there is no implementation of Gao's percolation model, it is unclear whether his approach was effective in practice for many-core architectures. We have also implemented our approach in C64 and also used sophisticated fine-grain synchronization (such as SSB) to improve performance and scalability of irregular applications on a many-core architecture.

- *Prefetching & Speculation:* In our parallel pipelining algorithm we overlap computation task with memory task. The concept of overlapping computation with I/O, network, and other long latency operations is old. Prefetching techniques [7, 23–25, 27, 33, 35] and thread speculation [6, 8, 9, 29, 31, 34] also use such overlapping concept. Most previous work on prefetching also focused on moving data (mostly contiguous data) from main memory to local memory (either to register or cache) prior to execution. Prefetching collects and performs an analysis of information of a program's instruction stream. Conceptually it is computation who *pulls* the data locally using prefetch instructions. In our method the local data determines which computation is ready to execute. In other words, data that is local to a core will *pull* computation to execute on the core. Burtscher et.al [17, 18] proposed to use extra cores to execute prefetching threads, which is a shared point with our approach. However, their framework is event-driven helper threading and future execution with speculation. Our approach for achieving just-in-time locality is data-driven, which is an efficient for executing irregular program. The underline execution model of our work is non-preemptive so that we may not bring in more data than what can be consumed. In

prefetching there is no control on how much data to prefetch—prefetching too much or too less data can impact the performance.

A variant of thread level speculation uses dependence by monitoring the reads and writes to memory locations. In producer-consumer loop iterations, the speculative execution leads to a violation of dependence, then must roll back. For the irregular memory access in the BC algorithm, in addition to the random reference to arrays d, σ, and δ, the references in the next iteration depends on the results in the current iteration. If we speculate the references based on the remaining neighbor vertices, it can lead to a large number of roll backs.

- *Inspector-Executor:* In inspector-executor paradigm [28, 30], an inspector translates global indices to local indices, identifies non-local references and generates communication schedules, an executor prefetches non-local data using schedules and performs computation. Although it is similar to our idea of decoupling/pipelining computation with memory, the underline model in inspector-executor is task-driven, whereas the computation tasks determine the schedule of data communication/prefetching.

- *Streaming programming:* In [15, 21] the authors performed a comprehensive study of regular/irregular scientific computing applications on streaming programming model. Both their work and ours share the streaming programming style of gather-compute-scatter. The way to gather/scatter data ahead makes our approach different from theirs. The streaming programming uses a DMA-style transfer, our approach utilizes the ample hardware thread units, where the way to hide the overhead of transformation is more flexible and requires less hardware cost.

7 Conclusion

Emerging future microprocessor chip technology unveils a new generation of many-core chip architectures that may contain 100 to 1,000 processing cores using a shared memory organization with large number of on-chip memory banks. 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. IBM Cyclops64 represents a new class of many-core architecture featuring with shared address space for on-chip memory between cores and explicit addressing without cache. This paper presents a study of evaluating the new many-core architectural features and shows how these features can be effectively exploited when executing challenging irregular applications in practice.

Because of the irregular behavior of BC algorithm, it is difficult to achieve high performance on a parallel architecture. By leveraging the key properties of explicit memory hierarchy and non-preemptive execution model, we propose a parallel pipelining algorithm to implement just-in-time locality for BC program on IBM Cyclops64. The parallel algorithm makes good use of the architectural support of fine-grain data synchronization. Our experimental results show that our methods are promising to improve scalability and performance of irregular application on a many-core architecture. Our future work will focus on implementing a runtime systems for supporting programmability on many-core architectures.

We would like to thank many members of the Computer Architecture and Parellel Systems Laboratory (CAPSL) at University of Delaware: Andrew Russo, Weirong Zhu and Ge Gan for helpful discussions.

References

[1] David Alderson, John C. Doyle, Lun Li, and Walter Willinger. Towards a theory of scale-free graphs: Definition, properties, and implications. *Internet Math*, 2(4):431–523, 2005.

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

[3] 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.

[4] 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.

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

[6] nones Carlos García Qui Carlos Madriles, Jesús Sánchez, Pedro Marcuello, Antonio González, and Dean M. Tullsen. Mitosis compiler: an infrastructure for speculative threading based on pre-computation slices. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 269–279, 2005.

[7] Trishul M. Chilimbi and Martin Hirzel. Dynamic hot data stream prefetching for general-purpose programs. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 199–209, New York, NY, USA, 2002. ACM.

[8] J. D. Collins, D. M. Tullsen, H. Wang, and J. P. Shen. Dynamic speculative precomputation. In *the 34th Annual International Symposium on Microarchitecture*, 2001.

[9] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, D. Lavery, and J. P. Shen. Speculative precomputation: Long-range prefetching of delinquent loads. In *The 28th International Symposium on Computer Architecture*, 2001.

[10] 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.

[11] 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.

[12] 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.

[13] Guangming Tan Dengbiao Tu. Characterizing betweenness centrality algorithm on multi-core architectures. In *the 2009 IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA'09)*, 2009.

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

[15] Mattan Erez, Jung Ho Ahn, Jayanth Gummaraju, Mendel Rosenblum, and William J. Dally. Executing irregular scientific applications on stream architectures. In *ICS '07: Proceedings of the 21st annual international conference on Supercomputing*, pages 93–104, New York, NY, USA, 2007. ACM.

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

[17] I. Ganusov and M. Burtscher. Future execution: A hardware prefetching technique for chip multiprocessors. In *2005 International Conference on Parallel Architectures and Compilation Techniques*, pages 350–360, 2005.

[18] I. Ganusov and M. Burtscher. Efficient emulation of hardware prefetchers via event-driven helper threading. In *2006 International Conference on Parallel Architectures and Compilation Techniques*, pages 144–153, 2006.

[19] Guang Gao, Jose Nelson Amaral, Andres Marquez, and Kevin Theobald. A refinement of the "htmt" program execution model. Technical report, CAPSL,University of Delaware, 1998.

[20] Guang R. Gao, Konstantin K. Likharev, Paul C. Messina, and Thomas L. Sterling. Hybrid technology multi-threaded architecture,. In *Proceedings of Frontiers '96: The Sixth Symposium on the Frontiers of Massively Parallel Computation*, pages 98–105, 1996.

[21] Michael Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, October 2006.

[22] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, 1991.

[23] Yuan Lin and David Padua. Compiler analysis of irregular memory accesses. In *PLDI '00: Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 157–168, New York, NY, USA, 2000. ACM.

[24] Jiwei Lu, Abhinav Das, Wei-Chung Hsu, Khoa Nguyen, and Santosh G. Abraham. Dynamic helper threaded prefetching on the sun ultrasparc cmp processor. In *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, pages 93–104, Washington, DC, USA, 2005. IEEE Computer Society.

[25] Chi-Keung Luk and Todd C. Mowry. Automatic compiler-inserted prefetching for pointer-based applications. *IEEE Transactions on Computers*, 48(2), 1999.

[26] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. In *ACM Transactions on Computer Systems*, volume 9, page 1, 1991.

[27] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *Journal of Parallel and Distributed Computing*, 12(2):87–106, 1991.

[28] Ravi Ponnusamy, Joel Saltz, and Alok Choudhary. Runtime-compilation techniques for data partitioning and communication schedule reuse. In *Supercomputing'93.*, 1993.

[29] L. Rauchwerger, Y. Zhan, and J. Torrellas. Hardware for speculative run-time parallelization in distributed shared memory multiprocessors. In *Proceedings of the 4th International Symposium on High-Performance Computer Architecture*, page 162, 1998.

[30] Shamik Sharma, Ravi Ponnusamy, Bongki Moon, Yuan Hwang, Raja Das, and Joel Saltz. Runtime and compile-time support for adaptive irregular problems. In *Supercomputing'94*, 1994.

[31] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture*, 2000.

[32] Guangming Tan, Vugranam C. Sreedhar, and Guang R. Gao. Just-in-time locality and percolation for optimizing irregular applications on a manycore architecture. In *21st Annual Languages and Compilers for Parallel Computing Workshop*, 2008.

[33] Youfeng Wu. Efficient discovery of regular stride patterns in irregular programs and its use in compiler prefetching. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 210–221, New York, NY, USA, 2002. ACM.

[34] W. Zhang and D. M. Tullsen. Accelerating and adapting precomputation threads for efficient prefetching. In *3th International Symposium on High Performance Computer Architecture*, 2007.

[35] Zheng Zhang and Josep Torrellas. Speeding up irregular applicaitons in shared-memory multiprocessors: Memory binding and group prefetching. In *22nd International Symposium on Computer Architecture*, 1995.

[36] 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.