Analysis and Performance Results of Computing Betweenness Centrality on IBM Cyclops64

Guangming Tan†, *Andrew Russo*†, *Vugranam Sreedhar*††, *Guang R. Gao*†

CAPSL Technical Memo 083
April 9, 2008

†Email: {guangmin,russo,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 (BC) – on a many-core architecture IBM Cyclops64. Dynamically non-contiguous memory access, unstructured parallelism and low arithmetic intensity in BC program 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 an efficient strategy for achieving scalability on a massive multi-threading many-core architecture. We demonstrate how to explore multi-grain parallelism and just-in-time locality with explicit memory hierarchy, non-preemptive thread execution and fine-grain data synchronization. Comparing to 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 and designers 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 (statically) predictable patterns, and "irregular applications" where data access and control flow have statically (and often even dynamically) unpredictable patterns. Analysis and optimization of such irregular applications are notoriously difficult. With the advent of massive many-core architectures, such as Intel Tera-scale [33] and IBM Cyclops64 [14] that contain tens or even hundreds of on-chip cores, it is extremely important to tackle the difficult problem of optimizing and scaling irregular applications. On-chip memory hierarchy, limited on-chip memory per core, and other features in such architectures make the problem even more difficult. Researchers are realizing that for many-core architectures the problem of scaling and optimizing irregular applications have to be done at different phases, including algorithmic changes and improvements to take advantage of the many-core architecture features [20, 30]. Many irregular applications are often implemented using pointer data structures such as graphs and queues and recursive control flow to traverse and manipulate such pointer data structures. It is difficult and often impossible to capture the data access patterns at compilation time. For architectures that support memory hierarchy, unpredicatable data access patterns often lead to higher off-chip memory access latency, which in turn can degrade the performance and scalability of irregular applications.

Computing betweenness centrality (BC) [17] in graph analysis is a good example of such irregular problems. BC is a popular quantitative index for the analysis of large scale complex network. It has been used extensively to build protein interaction network, identify key actors in terrorist network and study sexual/AIDS network. BC measures the control a vertex has over communication in the network. Brandes' algorithm [5] is one of fast algorithms for computing BC. In this paper, we refer to BC algorithm as one proposed by Brandes [5]. In general, BC algorithm calculates the centrality through two steps: BFS (breadth first search) traversal and backtrace accumulation. Due to scale-free [1] sparse graph traversal in these important applications, BC algorithm exhibits *dynamically non-contiguous memory access* and *unstructured parallelism*. Another explicit characteristic is *low arithmetic intensity* – the ratio between arithmetic operations and memory operations, which obviously forces programmers to expose an adequate amount of parallelism to the underlying many-core architecture within an application, instead of using higher speed processor. In this paper, we leverage some key features on a many-core architecture – IBM Cyclops64 to improve the performance of computing betweenness centrality in the scale-free sparse graph. The motivation of behind this idea is to be able to identify not only how programmers will use mechanisms provided in the emerging many-core architecture, but relative usefulness of various mechanisms as evidenced by their impact on application performance.

In consideration of *dynamically non-contiguous memory access*, *unstructured parallelism* and *low arithmetic intensity* in a large class of irregular applications, we identify four key properties of IBM Cyclops64 (C64) to address the challenge of executing irregular programs on many-core architectures:

- **Massive hardware thread units .** C64 integrates 160 simple 64-bit, single issue, in-order RISC processor operating at a moderate clock size (500MHz). The design of light-weight cores leaves more space for integrating more cores in a chip. For an application with low arithmetic intensity, it is desirable to exploit more parallelism. We not only implement a multi-grained parallel BC algorithm, but unearth additional parallelism to address the issue of irregular memory access.

- **Non-preemptive thread execution model & Explicit memory hierarchy.** These properties are fundamental to the proposed strategy for optimizing dynamically non-contiguous memory access. The ”dynamic” of memory operations is caused by data dependences, i.e. the level-by-level graph traversal in BC algorithm leads to producer-consumer data flow. Our strategy decouples computation with memory operations so that the memory accesses are operated by separated thread units to achieve *just-in-time locality* – data are local to a processing core just before computational consumer threads are scheduled to the core. Once the data dependences are specified by programmers, in the non-preemptive multi-thread execution model a producer-consumer operations are completed within once thread scheduling slot. It avoids multiple additional data swappings through memory hierarchy, which often degrades performance. Meanwhile, C64 is configured with three levels of memory hierarchy, which can be directly addressed (load/store) by all processing cores. With the explicit memory hierarchy programmers can exactly specify which level of memory the data are. Combining with non-preemptive execution it is feasible to schedule computation threads only access lower latency memory space. Our re-structure of BC algorithm is an orchestration of the computation and memory threads in a pipelining way so that the higher latency memory accesses are hidden and just-in-time locality is achieved.

- **Fine-grain data synchronization.** The fine-grained parallelism of visiting neighbors of a vertex (we refer to it as an extension operation) in BC algorithm is limited by the degrees of vertices, most of which is low in the scale-free sparse graph [1]. The multi-grained parallel algorithm exploits the parallelism between multiple extension operation in the same level. When two extension operations share the same neighbors, they require a synchronization so that one vertex is processed only one times. C64 provides an architectural support of fine-grained data synchronization – synchronization state buffer (SSB) [38]. Our strategy to optimize irregular memory access also takes good advantage of the SSB mechanism, and show that it is valuable to support fine-grain data synchronization on many-core architectures.

By utilizing the key architectural properties to map parallel BC algorithm we obtained a performance improvement of 4-50 times and scalability of 16 times on IBM Cyclops64. To the best of our knowledge this paper is the first indepth study of implementing a high performance BC program on many-core architectures. 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 C64 to re-structure BC algorithm. Section 5 evaluates the performance on a many-core chip architecture and draws implications on many-core architecture and programming. In section 6, we discuss the existing related techniques. Finally, section 7 concludes this paper.

2 Irregular Characteristics of BC Algorithm

In this section, we will briefly describe BC algorithm (for the detailed algorithm, refer to [5]), then examine the important irregular characteristics. Given a graph $G = (V, E)$ where V denotes the set of vertices and E the set of edges in G, the betweenness centrality (BC) measure of a vertex v is given by

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

where $\delta_{st}(v)$ denotes the fraction of shortest paths between s and t that pass through a particular vertex v, and is sometimes called as the pair-dependency of (s, t) on v. The algorithm contains a BFS (breath-first search) traversal and a backtrace accumulation. In the BFS traversal, the set of *predecessors* of a vertex v on a shortest path from source vertex s is generated:

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

At the same time, the *dependency* of s on $v \in V$ is calculated:

$$\sigma_s(v) = \sum_{t \in V} \sigma_{st}(v) \tag{3}$$

In the backtrace accumulation, a partial bc value of a predecessor is accumulated according to its successors. Equation 4 and 5 describe the calculation.

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

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

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

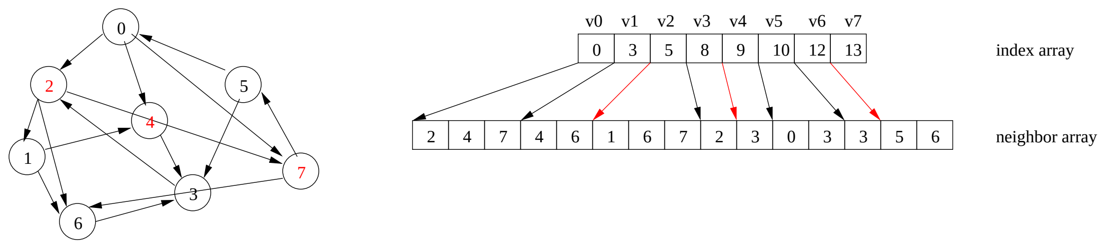

Figure 1: Adjacency array of a graph.

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

- *Dynamically non-contiguous memory access.* For instance, during the BFS phase, a queue is used to maintain the current vertices that is being extended (visiting the neighboring vertices of a vertex is referred to as an extension operation). The effectiveness of the existing locality optimization techniques such as prefetching and speculation rely on the continuity of the neighboring vertices and regular distance 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, we assume that the nodes v_2, v_4 and v_7 are currently in queue from which we pick nodes and process them. Notice that not only the neighboring nodes of v_2, v_4 and v_7 are located in different region in the adjacency array, but also the strides between the different regions are not constant. Also, the references to d, δ, σ, BC are almost random because the neighbors or predecessors of a node depends on the input graph. For an instance of visiting neighbors of v_2, v_4 and v_7, the sequence of reference to d is $d[1], d[6], d[7], d[3], d[5], d[6]$ (the same for σ). The dependence between $d, \delta, \sigma,$ and bc and the adjacency arrays means that the references to $d, \delta, \sigma,$ and bc are determined at runtime. Therefore, such non-contiguous or non-linear memory access pattern cannot benefit from current prefetching or speculation techniques.

- *Unstructured parallelism.* The available parallelism within an extension operation is proportional to the degrees of vertices. However, the degrees in the scale free graph obeys a power-law distribution [1], which means most of vertices has low degrees. Therefore, on many-core architectures with massive cores the parallelism will be very fine-grained. In order to utilize the ample processing units, an alternative way is to exploit more parallelism in BC algorithm, i.e. multi-grain parallel algorithms. The multiple extension operations at the same level can be finished in parallel only if they do not share the same neighboring vertices. In Figure 1 v_2, v_4, v_7 are at the same level of BFS tree. There exits parallelism during the extension of them, however, v_2 and v_7 share the same neighboring vertex v_6, a synchronization between two thread units processing v_2 and v_7 is required so that v_6 is touched by only one thread. Intuitively, a fine-grained mutex lock is a solution to the conflicts. We note that the size of memory storing lock is the number of vertices, which is usually so huge that the small local memory or cache can not hold it. Much worse, the memory access pattern to the lock array depends on that to the vertices, therefore, it is also dynamically non-contiguous.

- *Low arithmetic intensity.* The profiling of BC program execution shows that BFS traversal is the most time consuming. Looking at equation 2, 3 used in BFS traversal, an extension of one vertex needs two arithmetic (float point addition) operations, six memory operations. Although most of many-core designs do not resort to increase the speed of single core any more, the number of cores in a chip is increasing for a higher arithmetic performance. For traditional scientific computing applications with high arithmetic intensity and high parallelism, they naturally benefit from many-core architectures. In order to improve the performance of memory bound programs like BC algorithm, the key to a successful parallel program will be an efficient strategy to reduce the memory access overhead using the massive parallel thread units.

Most of current multi/many-core architectures are designed for regular scientific computing with high arithmetic intensity and highly explicit parallelism, the irregular applications like BC program

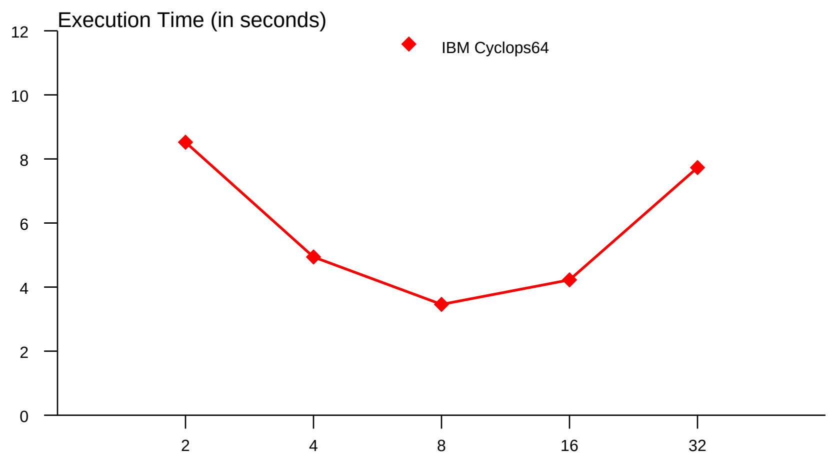

Figure 2: The performance of OpenMP implementation – HPCS SSCA2. The number of vertices and edges is 1024 and 8192, respectively

show different behaviors which do not match well with the many-core architecture. HPCS benchmark suite SSCA2 [2] specifies an OpenMP implementation of BC algorithm. Figure 2 shows its performance on IBM Cyclops64. As the number of threads is increased, the scalability and performance degrades. In order to achieve high performance on current many-core architectures, it is important to identify the characteristics impacting on application performance.

3 IBM Cylops64 Architecture

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

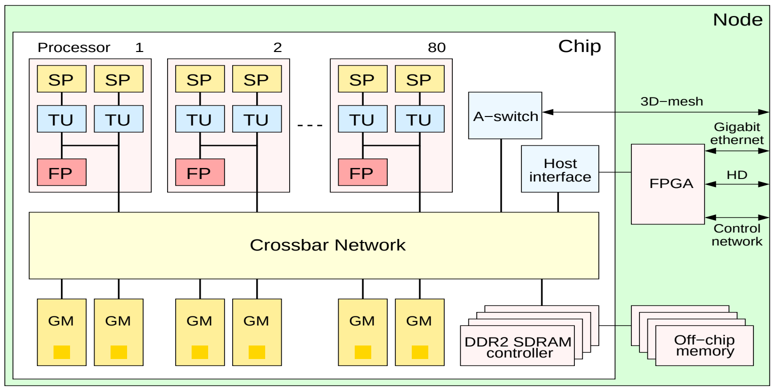

Figure 3: IBM Cyclops64 chip architecture

1. 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 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.

2. 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, which provides a fast temporary storage to exploit locality under software control. 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.

3. C64 incorporates 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.

4. C64 provides *synchronization state buffer (SSB)* to support fine-grain data synchronization (refer to [38] for details). SSB is 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. Thus, SSB avoids enormous memory storage cost and high latency memory access. the structure of an SSB is show in Figure 4. 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 lock synchronization.

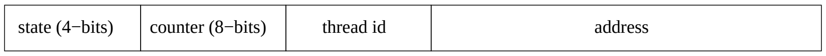

Figure 4: Structure of SSB entry

4 Mapping BC Algorithm to IBM Cyclops64

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

1. *Greedy parallelism.* 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 parallel programs. Therefore, we develop a multi-grain parallel BC algorithm, which implements coarse-, medium- and fine-grained parallelism simultaneously. The architectural support of efficient data synchronization is used in the multi-grained parallel algorithm. Being aware of low arithmetic intensity and memory hierarchy, we decouple computation (arithmetic) operations with memory operations, then exploit additional parallelism between them to make better use of massive parallel thread units.

2. *Just-in-time locality.* Like 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 is hard to optimize. We identify that the architectural characteristics of explicit memory hierarchy plus non-preemptive thread execution model make Dataflow-like execution model [15] possible. Although we can not eliminate the intrinsic data dependence (producer-consumer in BC algorithm), the decoupled computation operations and memory operations may be executed according to data-centric mechanism. Because data movement is under control of programmer and threads processing the data is not preemptive, computation threads consume *just-in-time locality* produced by memory threads, that is, data are local to computing core just before the core starts to processing the data.

Recall that there are two phases in the BC algorithm: the BFS traversal and backtrace accumulation (Section 2). Both phases have the similar computing behavior (although the backtrace accumulation is of high arithmetic intensity, our optimization strategies still work). To simplify the presentation we only describe the optimization for the BFS traversal phase.[1]

4.1 Preliminary Algorithm with Multi-grained Parallelism

Intuitively, the BC algorithm exhibits three level of parallelism itself. Each BFS from one source vertex can be started in parallel. For example in Figure 1, two BFS searches from vertex v_0 and v_2 can be dispatched to different parallel threads, respectively. The coarse-grained parallelism require multiple copies of the whole data structure. The medium-grained parallelism is exploited when two threads are visiting the neighboring vertices of different vertices in the same level. For example, v_2, v_4, v_7 lie in the same level of BFS tree. When visiting their neighbors, three parallel threads are activated to do the three task, respectively. However, if two vertices share the same neighboring vertices, a synchronization is forced to keep the shared neighbors being visited for just one times. The fine-grained parallelism is to visit the neighbors of a vertex in parallel. We may schedule three threads to visit the three neighboring

[1]algorithm for backtrace phase is very similar.

vertices v_2, v_4, v_7 of vertex v_0 in parallel. However, there are two factors hindering the various parallelism. First, The embarrassingly coarse-grained parallelism needs a copy of all data structures in the memory space of each thread. For a large scale graph in real world, the memory space usage is so huge that it often exceeds the physical memory even in traditional parallel computers, not speaking of current many-core architectures with small on-chip memory size. On the other hand, intensively concurrent memory accesses place burden on bandwidth, then slack the scalability. Second, note that the scale-free sparse graph has few vertices with high degrees. Both the available medium- and fine-grained parallelism depend on the degrees of vertices. Therefore, current parallel implementations [2–4] which only exploit either one of medium- and fine-grain parallelism can not achieve good performance on massive multi-threading architectures.

In the multi-grain parallel algorithm, the coarse-grain parallelism is easy to understand, we in detail present the combination of medium- and fine-grain parallelism. Let us denote the set of the vertices that is being extended in the current queue (the ith level of BFS tree) 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 neighboring set of vertices of a vertex v_{ij}. During execution the unvisited neighbor vertices u ($d[u] = -1$) are added to the current queue and the vertices being extended in the shortest path ($d[u] = d[v] + w[u][v]$) are added to the set of predecessor $P[u]$. The multi-grain parallel algorithm *logically* compacts all the neighbors in one level into one large set: $UN_i = \bigcup_{1 \leq j \leq n} N_j$, then partitions it among parallel threads. In the case of ignoring shared neighboring vertices, the multi-grain parallel algorithm achieves at least $p = \frac{|UN_i| = \sum_{j=1}^{k} |N_j|}{max_{j=1}^{k} |N_j|}$ times of parallelism than the fine-grain parallel algorithm at each level. However, in this initial multi-grain parallel BC algorithm there are two problems to be addressed:

- The multi-grain parallel algorithm achieves p times of parallelism at the cost of concurrently accessing p times of memory addresses. On C64 the small local storage is too small to hold the entire combined neighboring set, therefore a large number of high latency off-chip memory accesses happen. Meanwhile, the concurrent off-chip memory accesses make the contention of the limited off-chip bandwidth worse.
- Because the operations on one vertex are involved with only two arithmetic operations, the critical section protected by synchronization operations are so small that the synchronization overhead dilates the size of critical section. Thus, an efficient synchronization mechanism SSB on C64 will make a significant performance improvement. However, as noted in [38], SSB performance will degrade if the overflow of SSB happens when a synchronization operation is taken over by software. The multi-grain parallel algorithm may increase the number of activating synchronization memory addresses.

4.2 Achieving Just-in-time Locality

In the preliminary version of the multi-grain parallel BC program, we observe amount of off-chip memory accesses. In fact, the access to on-chip local storage have much higher bandwidth and lower latency than that to off-chip memory on many-core architectures including C64. Therefore, it is reasonable to schedule as many threads as possible to only 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 the data are in on-chip local storage or off-chip memory. In this paper, we refer to this thread execution model as a *weaker model*. Such weaker thread execution model may do well for regular applications, where there is an inherent cache/memory locality in the application. 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-schedule. Based on these architectural properties, programmers can specify the exact relationship between a thread execution and places of its data. Inspired by Dataflow model [15], we propose a strategy to achieve just-in-time locality for dynamically non-contiguous memory accesses on C64.

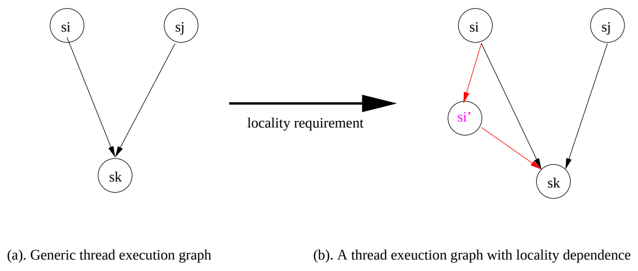
si
sj
sk
locality requirement
si
sj
si'
sk
(a). Generic thread execution graph
(b). A thread exeuction graph with locality dependence

Figure 5: thread execution graph

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 5). In a thread graph, a node s (i.e., a thread) is enabled if all its predecessor nodes have completed and the required data and control dependences 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 5). The locality requirements ensures 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.

Obviously, our strategy results in additional operations for ”creating” locality constraint. Note that the massive hardware thread units on 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 stronger 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 archi-

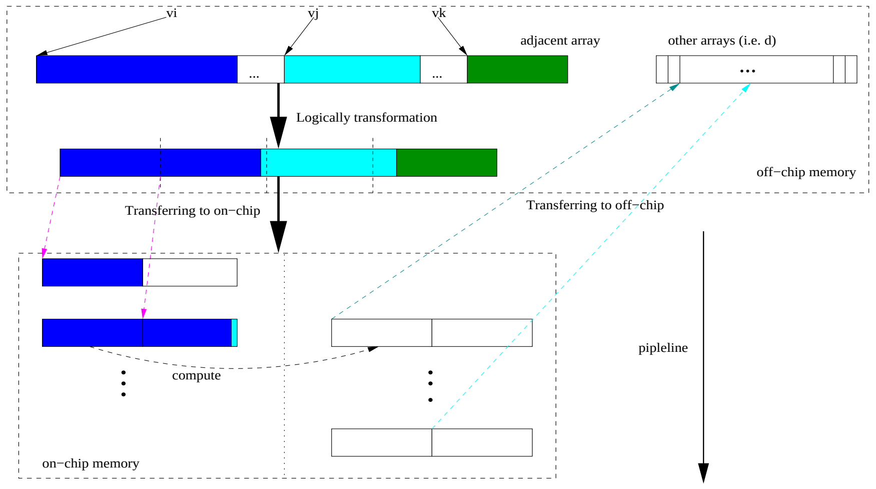

Figure 6: A demonstration of the parallel pipelining process for the BFS phase of the BC algorithm. For readability, the transformation (linearization/scatter) and data movement are depicted as two steps. A real implementation finishes them in single step by memory threads.

tectures (including many-core architectures) are designed to exploit "linear locality" [2] 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 $\sigma[1], \sigma[6]$, $\sigma[7]$ are contiguous, if we performed a linearization to these discrete memory locations just before they will be used to compute, then we achieve the "created" spatial locality. In an implementation, programs do *not* explicitly perform such a linearization operations in off-chip memory, but naturally complete it 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 size of the neighboring vertices region in adjacency array of each vertex, and collecting neighboring 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 neighboring vertcies and collection 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 scale-free graphs. Once computing the relevant information (d, σ) we write them back to off-chip memory using yet another memory threads.

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 $i-1$. Figure 6 illustrates the overall process across on-chip and off-chip memory. The threads operating multiple sub-blocks forms a pipeline, which achieves just-in-time locality for each

[2]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.

```
//Original serial Code
BFS(int v) {
        int dv = d[v];
        int sigmav = sigma[v];
        for (i = 0; i < NumEdges[v]; i++) {
            w = AdjacentArray[index[v]+i];
             if (d[w] < 0) {
                d[w] = dv + 1;
                 sigma[w] = 0;
             }
             if (d[w] = dv + 1)
                sigma[w] = sigmav + 1;
         }
}

//Code achieving just-in-time locality using three pipelined phase:
// (1) off-chip memory read (AdjacentArray);
//(2) compuation (accessing on-chip memory: buff, buff1, buff2, buff3);
// (3) off-chip memory write (AdjacentArray).
BFS(int v) {
  int offset = 0;
  int turn = 0;
  int dv = d[v];
  int sigmav = sigma[v];
  SPAWN_THERAD{
     for (i = 0; i < buffsize; i++)
           buff[turn][i] = AdjacentArray[index[v]+offset+i];
     offset += buffsize;
     turn ^= 1;};
  BARRIER_WAIT();
  while (offset < NumEdges[v]) {
        //1. off-chip memory read
        SPAWN_THERAD{
           for (i = 0; i < buffsize; i++)
              buff[turn][i] = AdjacentArray[index[v]+offset+i];
              offset += buffsize;
              }
           turn ^= 1;};
        SPAWN_THERAD{
            for (i = 0; i < buffsize; i++) {
             w = buff[turn][i];
             buff2[turn][i] = d[w];
             buff3[turn][i] = sigma[w];
             }
            turn ^= 1;};
        //(2). compuation (accessing on-chip memory);
        SPAWN_THERAD{
             for (i = 0; i < buffsize; i++) {
                 if (buff2[turn][i] < 0) {
                  buff2[turn][i] = dv+1;
                  buff3[turn][i] += 0;
                 }
                 if (buff2[turn][i] == dv+1)
                    buff3[turn][i] += sigmav;
                 }
               turn ^= 1;};
         //(3). off-chip memory write.
        SPAWN_THERAD{
             for (i = 0; i < buffsize; i++) {
                 w = buff[turn][i];
                 d[w] = buff2[i];
                 sigma[w] = buff3[i];
              }
             turn ^= 1;};
        BARRIER_WAIT();
   }
}
```

Figure 7: An illustration of BFS codes achieving just-in-time locality at one level sub-block. The complete multi-grain parallel for BFS phase is shown in Figure 7.

4.3 Using Architectural Support of Fine-grain Data Synchronization

Our previous work on C64 [11, 38] indicates that lock-based synchronization is better than lock-free one for explicit memory hierarchy. In fact, since there is no priority inversion and convoying problem in C64, performance and memory contention are the only factors of a lock-free data structure. For lock-free synchronization [19] in parallelizing betweenness centrality, due to irregular memory access pattern, we observed many failure of speculation and rollback. However, with software lock mechanism, we have to use additional lock array to assign one lock to each vertex. The size of lock array is the same with the number of vertices, which is usually huge in real world. Thus, for a large scale graph, it generates amounts of irregular off-chip memory accesses since these accesses are associated with that of vertices.

There are two remarkable features of the proposed parallel pipelining algorithm : 1). We explicitly separate computation from memory threads. The computation and memory threads 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 remarkable 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.

5 Evaluation

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

5.1 Methodology

We evaluate the performance characteristics of mapping approaches on a cycle-accurate C64 simulator [12] for the parallel BC program. The parameters of C64 architecture used in the experiments are summarized in Table 1. The toolchain on C64 consists of an optimized GCC compiler, a thread execution runtime systems TNT [13] (Pthread-like) and a TNT-based OpenMP [11]. By modifying HPCS SSCA2 bechmark [2], the proposed parallel algorithm is implemented using the TNT library. The TNT runtime always maintains as many threads as the number of cores. The directive SPWAN_THREAD in figure 7 is simply expressed as checking/assigning a idle thread.

We report the experimental results only for small problem sizes. Except for the limitation of simulator itself, note that C64 is devised as an accelerating engine for building a Petaflops supercomputer, and there will be massive C64 nodes in the systems. 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

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

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. It is also reasonable to evaluate performance of small size of problems on a simulator.

5.2 Empirical Results for Mapping Parallel BC Algorithm

In this section we present our empirical results. At first glance, we summarize an incremental optimization results of the parallel algorithm for just-in-time locality and synchronization using SSB. Figure 8 depicts 4-50 times reduction of execution time by comparing to the ported HPCS SSCA2 with OpenMP on IBM Cyclops64. The experimental data sets are generated by the program in HPCS SSCA2 benchmark.

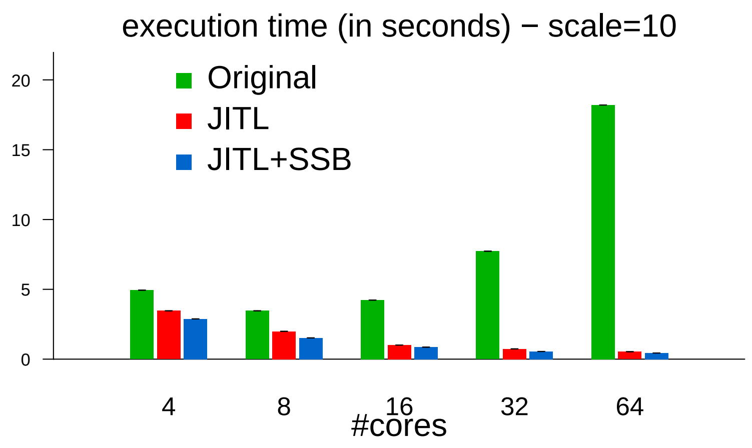

Figure 8: The incremental optimization results. JITL: just-in-time locality

For BC algorithm 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

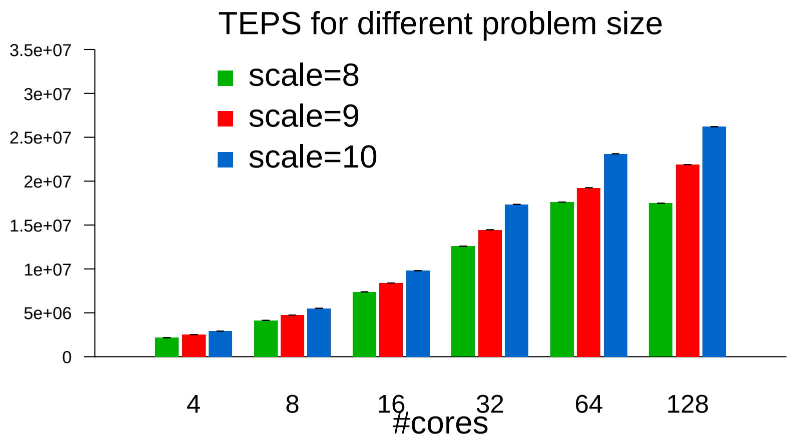

Figure 9: *Scalability* results of the parallel betweenness centrality algorithm (higher is better). The number of vertices $n = 2^{scale}$, $E(n) = 8n$.

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.

- For BC algorithm we represent the problem size in term of *scale*, where the number of vertices is $n = 2^{scale}$. Figure 9 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 (Figure 2), we can see that our optimization strategy shows a 16 times improvements of scalability. Using our approach we achieve a linear speedups for all test cases when the number of threads is less than 32. For the test case with a problem size $scale = 8$, 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 $scale = 9$ and 10. For BC algorithm the degree of a vertex determines the amount of parallelism that we can exploit. Although the multi-grain parallel algorithm reduces the number of idle threads, the maximum degree of a vertex is 64 for problem size $scale = 8$. So the available parallelism for this small problem size still leads to a little improvement on 128 threads. For $scale = 9$ and 10, where the maximum vertex degrees are 94 and 348, the performance and scalability is further improved.

- Figure 10 shows the effect of memory latency tolerance using the technique for creating just-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

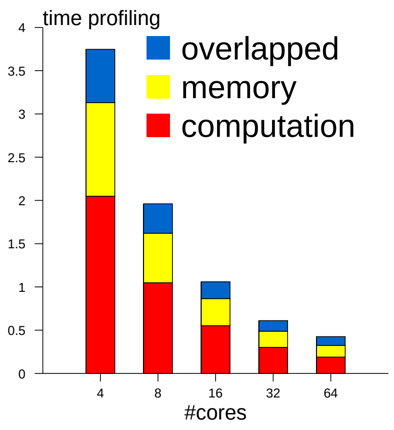

Figure 10: Time distribution and achieving off-chip memory latency tolerance.

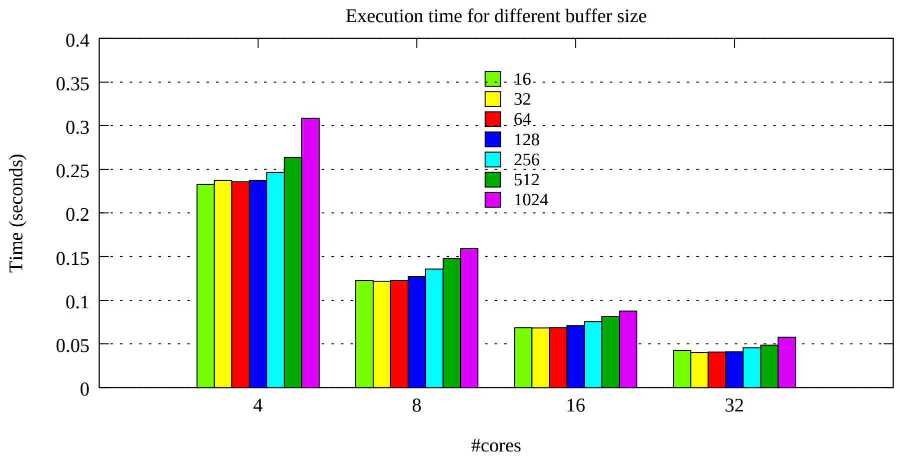

Figure 11: The comparison of running time using different sizes (bytes) of buffers.

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 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 11 shows that increasing buffer size does not achieve better performance.

- 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.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 the computation behavior of each vertex may be different. For example, if a vertex is not one of the predecessors of a neighbor that

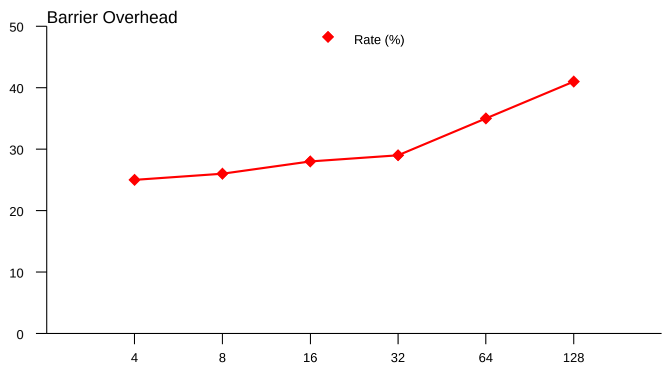

Figure 12: Overhead of barrier synchronization for scale = 10. The measured barriers include the barriers in both BFS and backtrace phase.

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 12 illustrates the percent of overhead of barrier synchronization with respect to the overall execution time.

- 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 reference, we implemented a highly optimized MCS [25] algorithm using in-memory atom operations on C64 [11, 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 13 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 highlight our joint study of architecture and algorithm, we compare the proposed parallel BC algorithm with both HPCS SSCA2 [2] on a Intel 4-way dual-cores Xeon SMP (8-processors) and a specific BC algorithm implemented by John Feo (previously in Cray Inc.) on 40-processors Cray MTA-2. 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., $scale = 22$. Unfortunately, we can

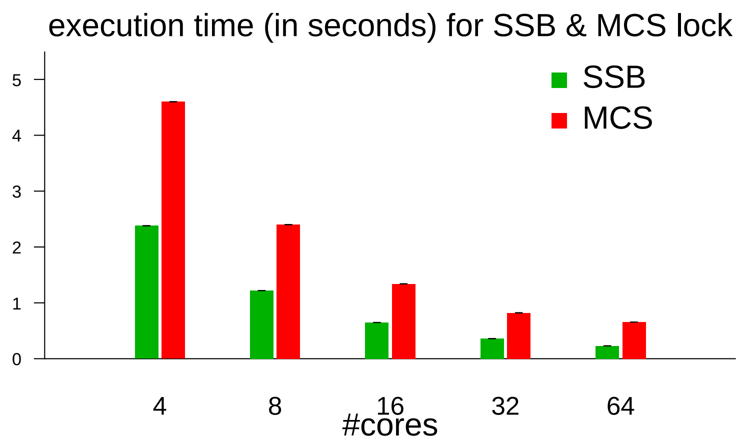

Figure 13: The comparison of software lock and SSB (BFS phase)

not run so large test sets on the C64 simulator), but the performance on the SMP is still weak (Due to space limitation, we do not present the detail here). This comparison indicates that the algorithm on MTA-2 achieves good weak scalability, but our algorithm on C64 achieves better strong scalability because we unearth more additional parallelism even for small problem sizes.

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

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

Although we present the results of one case of computing betweenness centrality, 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 on gives 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 bindwidth. For many-core architectures, an increasing gap between the number of cores and bindwidth is a serious obstacle to scalability of a parallel program. Configured with explicit memory hierarchy, 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. The results on Cray MTA-2 with flat memory (no cache) prove a similar point.

- 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 C64 is proven to be favorable. Similarly Cray MTA-2 provides full/empty mechanism at much more hardware cost.

- 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 dependences, a runtime system could parse the task graph and automatically determines 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 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 program 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, 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.

Our approach achieving just-in-time locality is inspired by Dataflow [15], it shares the same point with percolation, which was briefly discussed by Gao in the context of HTMT project [18]. 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. 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.

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 an old concept. Prefetching techniques [8, 21, 22, 24, 26, 35, 37] and thread speculation [6, 9, 10, 28, 31, 36] 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. There are several differences between our approach and prefetching. (1) In software prefetching order of execution of a program is predetermined and prefetch instructions are inserted to ensure that the data are available when they are needed by the computation. In other words, conceptually computation threads ”pull” the data locally using prefetch instructions. On the other hand, in our method the local data determines which computation thread is ready to execute. In other words, data that is local to a core will ”pull” computation thread to execute on the core. (2) In our threading model, a thread has to satisfy two requirements before it can be enabled and ready to run: (i) data/control dependencies and (ii) locality constraints. Our execution model is also non-preemptive and so we cannot 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. Besides, previous works do not discuss the impact of prefetching in the context of massive multithreading many-core. A variant of thread level speculation uses dependences by monitoring the reads and writes to memory locations. In producer-cosumer 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 if the vertices have been marked.

There have been several work on the optimization of irregular programs on parallel architectures. Recently, Williams et.al. [34]’s work on sparse matrix-vector multiplication on conventional multi-core platforms implies that new methods for parallel irregular computing is imperative. Erez et.al [16] performed a comprehensive study of 4 irregular scientific computing applications on a streaming processor. Both their work and ours share the streaming programming style of gather-compute-scatter. The way to gather data ahead make our approach different from theirs. In [16] the streaming processor uses a DMA-style transfer, our approach utilizes the ample hardware thread units, where to hide the overhead of transformation is easier and require less hardware cost. Salz et.al. [29] studied runtime methods to automatically parallelization and scheduling of loops. Trabado et.al [32] proposed a data parallel language extensions for exploiting locality in irregular problems, their work also focus on loops. Nikolopoulos et.al.’s [27] work tried to minimize the programming effort with OpenMP for irregular parallel codes. Lucco [23] developed a methodology for compiling and executing irregular parallel programs, the goal of his work is to achieve a optimal dynamic scheduling method. Based on objected-orented language, Chien [7] indicated that explicit management of namespace is efficient for irregular programs, but their experiments only reported the results for traditional scientific computing on conventional parallel computers.

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 such 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 on 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 make a good usage 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 in a many-core architecture. Our future work will focus on implementing a runtime systems for supporting programmability on many-core architectures.

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 Socialogy*, 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] A. Chien, J. Dolby, B. Ganguly, V. Karamcheti, and X. Zhang. Evaluating high level parallel programming support for irregular applications in icc++. In *Proceedings of International Scientific Computing in Object-Oriented Parallel Environments Conference (ISCOPE'97)*, 1997.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. In *ISCA '75: Proceedings of the 2nd annual symposium on Computer architecture*, pages 126–132, New York, NY, USA, 1975. ACM.

[16] 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.

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

[18] Guang R. Gao and et.al. Programming models and system software for future high-end compting systems: work in progress. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, 2003.

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

[20] M. Kulkarni, K. Pingali, B. Walter, G. Ramanarayanan, K. Bala, and P. Chew. Optimistic parallelism requires abstractions. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 211–222, 2007.

[21] 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.

[22] 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.

[23] Steven Lucco. A dynamic scheduling method for irregular parallel programs. In *PLDI '92: Proceedings of the ACM SIGPLAN 1992 conference on Programming language design and implementation*, pages 200–211, New York, NY, USA, 1992. ACM.

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

[25] 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.

[26] 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.

[27] D. S. Nikolopoulos, C. D. Polychronopoulos, and E. Ayguade. Scaling irregular parallel codes with minimal programming effort. In *SC'01*, 2001.

[28] 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.

[29] J. H. Salz, R. Mirchandaney, and K. Crowley. Run-time parallelization and scheduling of loops. In *IEEE Transactions on Computers*, volume 40, pages 603–612, 1991.

[30] Mikhail Smelyanskiy, Victor W. Lee, Daehyun Kim, Anthony Nguyen, and Pradeep Dubey. Scaling performance of interior-point method on large-scale chip multiprocessor system. In *IEEE/ACM SC'07*, 2007.

[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] Guillermo P. Trabado and Emilio L. Zapata. Data parallel language extensions for exploiting locality in irregular problems. In *Languages and Compilers for Parallel Computing*, pages 218–234, 1997.

[33] S. Vangal, J. Howard, G. Ruhl, S. Dighe, H. Wilson, J. Tschanz, D. Finan, P. Iyer, A. Singh, T. Jacob, S. Jain, S. Venkataraman, Y. Hoskote, and N. Borkar. An 80-tile 1.28tflops network-on-chip in 65 nm cmos. In *In Proceedings of IEEE International Solid-State Circuits Conference*, pages 98–589, 2007.

[34] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *IEEE/ACM SC'07*, 2007.

[35] 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.

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

[37] 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.

[38] 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.