Challenges in Mapping Graph Exploration Algorithms on Advanced Multi-core Processors

Oreste Villa[1,2], Daniele Paolo Scarpazza[1], Fabrizio Petrini[1] and Juan Fernández Peinador[1]

[1]Pacific Northwest National Laboratory
Computational & Information Sciences Division
Richland, WA 99352 USA

[2]Politecnico di Milano
Dipartimento di Elettronica e Informazione
Milano I-20133, Italy

Abstract

Multi-core processors are a shift of paradigm in computer architecture that promises a dramatic increase in performance. But multi-core processors also bring an unprecedented level of complexity in algorithmic design and software development.

In this paper we describe the challenges and design choices involved in parallelizing a breadth-first search (BFS) algorithm on a state-of-the-art multi-core processor, the Cell Broadband Engine (Cell BE).

Our experiments obtained on a pre-production Cell BE board running at 3.2 GHz show almost linear speedups when using multiple synergistic processing units, and an impressive level of performance when compared to other processors. The Cell BE is typically an order of magnitude faster than conventional processors, such as the AMD Opteron and the Intel Pentium 4 and Woodcrest, an order of magnitude faster than the MTA-2 multi-threaded processor, and two orders of magnitude faster than a BlueGene/L processor.

1 Introduction

Advanced multi-core processors promise a major advance in performance, coupled with a small form factor, limited power consumption and the driving force of the commodity market. The Cell BE processor, jointly designed by IBM, Toshiba and Sony, is an important example of such architectures, and it is rapidly gaining popularity as a basic building block for high performance clusters and supercomputers.

In the high performance computing community, graph exploration algorithms occupy a position of primary importance. Many areas of science (genomics, astrophysics, artificial intelligence, national security and information analytics) demand techniques to explore large-scale data sets which are, in most cases, represented by graphs. In these areas, search algorithms are the computational engines to discover vertices, paths, and groups of vertices with desired properties. Among graph search algorithms, Breadth-First Search (BFS) is probably the most common, and a building block for a wide range of graph analysis applications [2].

A good amount of literature deals with the design of BFS solutions, either based on commodity processors [11, 1, 8] or dedicated hardware [3]. Nevertheless, despite the important role of multi-core architectures in high performance computing and the relevance of large graph search problems in this domain, little attention has been devoted to exploring the potentials of multi-cores when applied to graph search algorithms. No studies have investigated how effectively the Cell BE can be employed to perform a BFS search on large graphs, and how it compares against other commodity or dedicated solutions in terms of performance.

The problem of searching large graphs alone poses difficult challenges, mainly due to the vast search space imposed by the sheer amount of data, combined with the lack of spatial and temporal locality in the data access pattern. Additionally, on the Cell BE the memory hierarchy must be explicitly managed at software level.

This paper describes the challenges and the design choices involved in mapping a BFS algorithm on the Cell BE. The choice of a conceptually simple algorithm such as the BFS allows for a complete, in-depth analysis.

This paper provides three primary contributions. (1) A detailed description of the BFS graph exploration algorithm for multi-core processors. We put emphasis on the peculiar

The research described in this paper was conducted under the Laboratory Directed Research and Development Program for the Data Intensive Computing Initiative at Pacific Northwest National Laboratory, a multi-program national laboratory operated by Battelle for the U.S. Department of Energy under Contract DEAC0576RL01830.

characteristics of this algorithm, such as the data-flow, the explicit management of a hierarchy of working sets and the data orchestration between them. (2) A detailed experimental evaluation of the algorithm that explains *how* its different components are integrated together, and an accurate comparison with other architectures. The goal is to provide insight on the performance impact of several architectural and design choices. (3) Perhaps the most interesting contribution is the parallelization methodology that we have adopted to design our algorithm and guide the software development process. Our work is inspired by the Bulk-Synchronous Parallel (BSP) model [9, 5].

Our methodology is based on the following cornerstones: (1) a high-level algorithmic design where the user focuses on the essential *machine-independent* aspects of the algorithm that we believe can guarantee portability with performance to other multi-core processors, (2) a *machine-dependent* part that refines the initial high-level algorithmic description to embed the specific optimizations of a particular multi-core processor, (3) a BSP-style global coordination, used by both the machine-dependent and machine-independent parts, that allowed us to implement and tests the various steps of our algorithm in a modular way, (4) an accurate analytical performance model, facilitated by the BSP programming style, that helped us determine upper and lower bounds on the execution time of each step of the algorithm and (5) the enforcement of a deterministic behavior of the algorithm that, for a given input set and number of processing elements, can be re-played as a sequential program.

Our work provides a valuable contribution for application developers, by identifying a software path that can be followed by other applications. We expect that many of these hand-crafted techniques described in this paper will eventually migrate into parallelizing tools and compilers. Processor designers might also find interesting information to develop the new generation of streaming processors, that will likely target from the very beginning the computing needs of scientific applications.

The remainder of this paper is organized as follows. Section 2 defines the BFS exploration problem. Section 3 presents our BFS implementation, especially conceived to exploit the features of the Cell BE, and Section 4 analyzes the performance of its components. Section 5 describes the experiments we have performed to measure the overall performance of our algorithm on a real Cell BE system and compares these results with the ones provided by other, state-of-the-art processors. Finally, Section 6 concludes the paper.

2 The Breadth-First Search algorithm

In this Section we present the methodology we used to parallelize the Breadth-First Search (BFS) algorithm. We first introduce the notation employed throughout the rest of this paper and a baseline, sequential version of BFS. We then describe a simplified parallel algorithm as a collection of cooperating shared-memory threads. Finally, we refine the level of detail with a second parallel algorithm that explicitly manages a hierarchy of working sets.

A graph $G = (V, E)$ is composed by a set of vertices V and a set of edges E. We define the *size* of a graph as the number of vertices $|V|$. Given a vertex $v \in V$, we indicate with E_v the set of neighboring vertices of v (or *neighbors*, for short), such that $E_v = \{w \in V : (v, w) \in E\}$, and with a_v the vertex arity, i.e. the number of elements $|E_v|$. We will denote as $\bar{a}$ the average arity of the vertices in a graph, $\bar{a} = \sum_{v \in V} |E_v|/|V|$.

Given a graph $G(V, E)$ and a root vertex $r \in V$, the BFS algorithm explores the edges of G to discover all the vertices reachable from r, and it produces a *breadth-first tree*, rooted at r, containing all the vertices reachable from r. Vertices are visited in *levels*: when a vertex is visited at level l it is also said to be at distance l from the root.

In Algorithm 1 we give the pseudo-code description of a sequential BFS algorithm. Despite its simplicity, this description is useful to illustrate the fundamentals of the algorithm, which are crucial to understand the parallel versions of the BFS that we will present later.

At any time, Q is the set of vertices that must be visited in the current level. Q is initialized with the root r (see line 4). At level 1, Q will contain the neighbors of r. At level 2, Q will contain these neighbors' neighbors (except those visited in level 0 and 1), and so on.

Algorithm 1 Sequential BFS exploration of a graph.

```
Input:      $G(V, E)$, graph;
            $r$,       root vertex;
Variables:  $level$,   exploration level;
            $Q$,       vertices to be explored in the current level;
            $Qnext$,   vertices to be explored in the next level;
            $marked$,  array of booleans: $marked_i \ \forall i \in [1...|V|]$;

 1 $\forall i \in [1...|V|] : marked_i$ = false
 2 $marked_r$ = true
 3 $level \leftarrow 0$
 4 $Q \leftarrow \{r\}$
 5 repeat
 6     $Qnext \leftarrow \{\}$
 7     for all $v \in Q$ do
 8         for all $n \in E_v$ do
 9             if $marked_n$ = false then
10                 $marked_n \leftarrow$ true
11                 $Qnext \leftarrow Qnext \cup \{n\}$
12             end if
13         end for
14     end for
15     $Q \leftarrow Qnext$
16     $level \leftarrow level + 1$
17 until $Q = \{\}$
```

During the exploration of each level, the algorithm scans the content of Q, and for each vertex $v \in Q$ it adds the corresponding neighbors to *Qnext*. *Qnext* is the set of vertices to visit in the next level. At the end of the exploration of a level, the content of *Qnext* is assigned to Q, and *Qnext* is emptied. The algorithm terminates when there are no more neighbors to visit, i.e. Q is empty (line 17).

To algorithm visits a vertex only once. To do so, it maintains an array of boolean variables $marked_v$ $\forall v \in V$, where each variable $marked_v$ tells whether vertex v has already been visited. Neighboring vertices are added to *Qnext* only when they have not been marked before.

A straightforward way to parallelize the algorithm just presented is by exploring the vertices in Q concurrently with all the available processing elements (PEs). The **for all** statement of Algorithm 1 (line 7) can be executed in parallel by different threads. The only constraint is that access to the array *marked* must be protected with some synchronization mechanism, such as a multiple locks [6]. This is the conventional solution in a cache-coherent shared-memory machine with uniform memory access time and a limited number of hardware threads, which unfortunately cannot scale well with a larger number of processing elements [7].

In our algorithm we adopt a different approach, illustrated in Algorithm 2. We partition V in disjoint sets V_i, one per each PE. We say that PE i *owns* the vertices in its partition V_i. Each PE i is only allowed to explore and mark the vertices it owns, and it must forward any other vertices to the respective owners. As indicated in line 9, all steps are globally synchronized across the processing elements. We denote synchronization points with a horizontal line, which imposes a sequential order between the phases.

The steps of Algorithm 2 are executed in parallel by all the available PEs. PE i accesses its own private Q_i and $Qnext_i$, and its partition of the *marked* array, which includes only the variables associated to the vertices V_i that it owns.[1] Additionally, each PE i has a set of private outgoing and incoming queues, called $Qout_{i,1}, Qout_{i,2}, ...Qout_{i,N}$ and $Qin_{i,1}, Qin_{i,2}, ...Qin_{i,N}$, respectively. Through these queues, PEs can forward the vertices to their respective owners.

At initialization time, the root vertex r is assigned to its owner's Q_i. During the exploration, each PE i examines the vertices v in Q_i and dispatches the vertices in E_v which belong to PE p to $Qout_{i,p}$. Then, when all the PEs have completed this phase, an all-to-all personalized exchange takes place, and the contents of each $Qout_{i,p}$ are transferred to $Qin_{p,i}$. Each outgoing queue of PE i having PE p as a recipient is copied into the incoming queue residing at PE p and having i as a sender. This exchange delivers the vertices to their respective owners.

[1]Also *level* is a private variable and should be denoted with a subscript indicating the PE. The subscript can be avoided without ambiguity because the value of *level* is kept the same across all PEs.

Algorithm 2 bulk-synchronous parallel version of a breadth first graph exploration.

```
Input:      $G(V,E)$,              graph;
            $r$,                   root vertex;
            $N$,                   available processing elements (PE);
            $V_1, V_2, ..., V_n : (\bigcup_{i:1...N} V_i) = V$,
                                   $\forall (i,j) \in [1...N]^2 :\ V_i \cap V_j = \{\}$ if $i \neq j$;
Variables:  $Q_i$,                 vertices to be explored in the current level;
            $Qnext_i$,             vertices to be explored in the next level;
            $level$,               exploration level;
            $marked_v$,            $\forall v \in V_i$;
            $Qout_{i,p}$,          $\forall p \in [1...N]$, outgoing queues;
            $Qin_{i,p}$,           $\forall p \in [1...N]$, incoming queues;

Processing element $i$:

 1 $level \leftarrow 0$
 2 $Q_i \leftarrow \{\}$
 3 $\forall v \in V_i : marked_v \leftarrow$ false
 4 if $r \in V_i$ then
 5     $Q_i \leftarrow \{r\}$
 6     $marked_r \leftarrow$ true
 7 end if
 8
 9 repeat in lockstep across the processing elements:
10     $Qnext_i \leftarrow \{\}$
11     $\forall p \in [1...N] : Qout_{i,p} \leftarrow \{\}$
12     ------------------------------------------------
13     // Gather and Dispatch
14     for all $(p,v) \in [1...N] \times Q_i$ do
15         $Qout_{i,p} \leftarrow Qout_{i,p} \cup \{(v, E_v \cap V_p)\}$
16     end for
17     ------------------------------------------------
18     // All-to-All
19     $\forall p \in [1...N] : Qin_{p,i} \leftarrow Qout_{i,p}$
20     ------------------------------------------------
21     // Bitmap
22     for all $n \in E_v$ where $(v, E_v) \in \left(\bigcup_{p:1...N} Qin_{i,p}\right)$ do
23         if $marked_n$ = false then
24             $marked_n \leftarrow$ true
25             $Qnext_i \leftarrow Qnext_i \cup \{n\}$
26         end if
27     end for
28
29     $Q_i \leftarrow Qnext_i$
30     $level \leftarrow level + 1$
31     ------------------------------------------------
32 until $\forall p \in [1...N] : Q_p = \{\}$
```

Note: horizontal lines indicate barrier-synchronization points.

Next, each PE examines the queues of incoming vertices, marks them and adds those that have not been visited to its private $Qnext_i$, as done in the previous algorithm. By construction $Qnext_i$, which will become Q_i during the next level, consistently contains only vertices owned by PE i.

The parallel algorithm we have just presented (Algorithm 2) does not consider any size limitations of Q, *Qnext*, *Qin*, and *Qout*. If the private data structures are entirely allocated in the local storage of each processing element, Algorithm 2 can quickly overflow the memory available to each core. Local memories in a multi-core processor are relatively small, and the problem is likely to intensify in the future: while advances in chip integration promise tens, perhaps hundreds of cores in a silicon die, it is unlikely that on-chip memory size will follow the same trend [10]. For this reason, we believe that application developers should design their algorithms taking explicitly into account application working sets and data orchestration between them.

Algorithm 3 is a refined version of the parallel algorithm that explicitly distinguishes between variables allocated in main memory and in the local memory of each single PE. Local memory variables can be subject to size constraints, but the algorithm can access their contents at any granularity (element or block). On the other hand, variables allocated in main memory do not have any size constraint, but they can be accessed only via explicit operations, preferably at a coarser granularity.

In algorithm 3, the graph G and queues Q_i and $Qnext_i$ are allocated in main memory, while *marked*, *Qin* and *Qout* are now allocated in the local memory. The algorithm does not access elements in Q directly; rather it *fetches* blocks of Q_i into a smaller, size-constrained queue named bQ_i (the b prefix intuitively identifies local buffers), via an explicit *fetch* operation (see line 10). Symmetrically, it does not add elements directly to $Qnext_i$, but to a small buffer $bQnext_i$, which is then *committed* to *Qnext* via an explicit operation (see line 39). Adjacency lists E_v are also explicitly loaded into the local data structure *bG* during the *gather* (see line 15).

The algorithm can operate on graphs of arbitrary size and arity, provided that all the local variables fit in local memory, that *bG* is at least as large as the longest adjacency list E_v and each $Qin_{i,p}$ is at least as large as $Qout_{i,p}$. Overflows of *bG* can be easily managed at graph creation time by splitting a single adjacency list in multiple lists having the same father, and incorporating minor algorithmic changes to load-balance the exploration of these *heavy* vertices across multiple PEs.

Each partition of the *marked* variables (which are altogether as many as $|V|$) must fit in the local memory of each PE. This raises an additional constraint on the maximum size of graphs explorable with the above algorithm on a given architecture, which we will discuss later. Except for the newly-introduced Fetch, Gather and Commit steps, the new algorithm is only slightly more sophisticated that the previous one, but incorporates what we believe are the essential features to achieve optimal efficiency on the existing and future generations of multi-core processors.

3 Implementation of the Algorithm

In this section we describe how we parallelized the Algorithm 3 on the Synergistic Processing Elements (SPEs) of

Algorithm 3 bulk-synchronous parallel exploration of a graph, with limited-storage constraints.

```
Input:  $G(V, E)$,          graph (allocated in main memory);
        $r$,                root vertex;
        $N$,                available processing elements (PE);
        $V_1, V_2, ..., V_n : (\bigcup_{i:1...N} V_i) = V$,
                            $\forall (i, j) \in [1...N]^2 : V_i \cap V_j = \{\}$ if $i \neq j$;
Variables allocated in main memory:
        $Q_i$,              vertices to be explored in the current level;
        $Qnext_i$,          vertices to be explored in the next level;
Variables allocated in the memory of the $i$th PE:
        $level$,            exploration level;
        $marked_v$          $\forall v \in V_i$;
        $Qout_{i,p}$        $\forall p \in [1...N]$, outgoing queues;
        $Qin_{i,p}$         $\forall p \in [1...N]$, incoming queues;
        $bQ_i$,             a size-constrained subset of $Q$;
        $bQnext_i$,         a size-constrained subset of $Qnext$;
        $bG_i$,             a size-constrained subset of $E$;
Processing element $i$:
 1 $level \leftarrow 0$;
 2     $Q_i \leftarrow \{\}$;
 3 $\forall v \in V_i : marked_v \leftarrow$ false
 4 if $r \in V_i$ then
 5     $Q_i \leftarrow \{r\}$
 6     $marked_r \leftarrow$ true
 7 end if
 8
 9 repeat in lockstep across the processing elements:
10     // 1. Fetch
11     load $bQ_i \subset Q_i$
12     $Q_i \leftarrow Q_i - bQ_i$
13     while $bQ_i \neq \{\}$ do
14         ----------------------------------------
15         // 2. Gather
16         determine a subset $\{v_1, v_2, ..., v_n\} \subset bQ_i$ such that:
17             $|E_{v_1}| + |E_{v_2}| + ... + |E_{v_n}| <$ max allowed $|bG_i|$
18         $bQ_i \leftarrow bQ_i - \{v_1, v_2, ..., v_n\}$
19         load $bG_i \leftarrow \{E_{v_1}, E_{v_2}, ..., E_{v_n}\}$
20         ----------------------------------------
21         // 3. Dispatch
22         $\forall p \in [1...N] : Qout_{i,p} \leftarrow \{\}$
23         for all $(p, v) \in [1...N] \times \{v_1, v_2, ..., v_n\}$ do
24             $Qout_{i,p} \leftarrow Qout_{i,p} \cup \{(v, E_v \cap V_p)\}$
25         end for
26         ----------------------------------------
27         // 4. All-to-All
28         $\forall p \in [1...N] : Qin_{p,i} \leftarrow Qout_{i,p}$
29         ----------------------------------------
30         // 5. Bitmap
31         $bQnext_i \leftarrow \{\}$
32         for all $n \in E_v$ where $(v, E_v) \in \left(\bigcup_{p:1...N} Qin_{i,p}\right)$ do
33             if $marked_n =$ false then
34                 $marked_n \leftarrow$ true
35                 $bQnext_i \leftarrow bQnext_i \cup \{n\}$
36             end if
37         end for
38         ----------------------------------------
39         // 6. Commit
40         $Qnext_i \leftarrow Qnext_i \cup bQnext_i$
41     end while
42     $Q_i \leftarrow Qnext_i$
43     $level \leftarrow level + 1$
44     ----------------------------------------
45 until $\forall p \in [1...N] : Q_p = \{\}$
```

Cell BE. At this stage we analyze lower-level details, such as remote DMAs, double buffering, data alignment, etc.

Figure 1 presents a schematic overview of the steps composing this implementation, and the data structures they operate on. From a software engineering point of view, each of these steps can be designed, tested and optimized in isolation. A detailed description of each step follows.

Fetch. Step 1 fetches a portion of Q into bQ. The fetch is implemented by a DMA transfer, in a double buffering fashion. This means that there are two data structures associated with bQ, and Step 1 waits for the previous transfer (if any) to complete, it swaps the two buffers to make the newly-arrived data available to the subsequent steps, and it starts a new transfer for the next block of Q, using the other buffer as a destination. Because of the much higher latency associated with the remaining steps in the algorithm, Step 1 never has to actually wait for bQ to arrive, except for the very first fetch at the beginning of each level of exploration. In our implementation bQ is a relatively small buffer, only 512 bytes.

Gather. Step 2 explores the vertices in bQ and loads their respective adjacency lists in bG, until bG is full, using a DMA list. The Cell BE architecture provides DMA lists as a low-overhead means to orchestrate a sequence of transfers (up to 2,048), which are carried out without further intervention of the processor, obtaining almost optimal overlap with the computation. It is necessary to know the size of a data structure before loading it with a DMA list, and there is no obvious way to know the length of an adjacency list *before* loading it. For this reason, rather than representing vertices with their vertex identifiers, we represent them with *vertex codes*. A vertex code is 32-bit a word (or a larger binary representation, if the cardinality of the graph requires it) where a certain number of bits are reserved for the vertex identifier, and the others are reserved to encode the length of its adjacency list, possibly expressed in a quantized form if there is a limited amount of bits available. In detail, a vertex code has two fields, the vertex identifier (which is v) and the vertex length, which is an encoded representation of $|E_v|$. With the help of the length field, Step 2 can prepare a DMA list to transfer as many adjacency lists as possible into bG, minimizing the amount of space wasted and optimizing the accesses to main memory. Step 2 operates in a double-buffering style. Hence, its actual code consists in waiting for the in-flight bG transfer to complete, swap buffers, prepare another DMA list for the next bG transfer and initiate it. The same considerations about wait times stated for Step 1 apply here.

Dispatch. The purpose of Step 3 is to split the adjacency lists previously gathered by Step 2 into the respective $Qout$ queues. To expedite this step, we adopt an optimized encoding format for the adjacency lists. In detail, at graph generation time, adjacency lists are encoded in a per-SPE split form. Additionally, each adjacency list comprises a header which specifies the offset and length of each per-SPE portion of that list. Each portion is padded to a multiple of a 4 words size, in such a way that Step 3 can dispatch adjacency lists operating one quadword at a time, which is the size of registers and the width of the loads from local store. To increase the efficiency of Step 3, multiple iterations can be unrolled: in this case, the step may load and process more quadwords at a time, thus requiring adjacency lists to be padded to larger quantities.

All-to-all. Step 4 is the all-to-all personalized exchange in which each SPE delivers the $Qout$ queues to their destinations. It is not necessary to transfer the $Qout_{i,i}$, and $Qin_{i,i}$ is simply an alias of $Qout_{i,i}$.

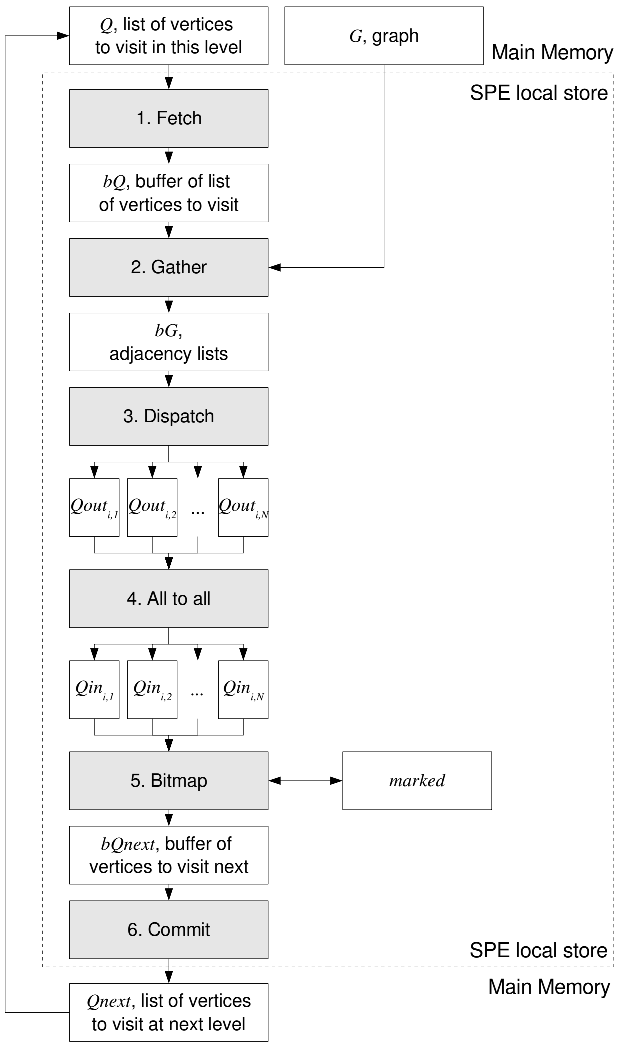

Figure 1. The data flows involved in the different steps of the algorithm.

Step 4 requires an appropriate synchronization mechanism to detect the presence of valid data in a *Qin*. The most efficient way to implement this mechanism is through *communication guards*. A communication guard is a flag that the receiver resets before the transfer is started, and that the sender sets when the transfer is complete. At any time during the transfer, the receiver can determine the status of the transfer by reading this flag. For the mechanism to work properly, a guard must be reserved for each outgoing queue, and appropriate hardware support must be employed to guarantee that the guards are not transferred before the payload has completely reached its destination.

To allow maximum efficiency, transfers are organized according to a predefined schedule in a circular fashion using a predefined DMA list. In fact, both the sequence of transfers and their coordinates are known in advance at program initialization, and the DMA list can be prepared at that time. Therefore, the actual implementation of Step 4 is a simple invocation of the `mfc_putlb` intrinsic, which takes only a few clock cycles at the source.

Bitmap. In Step 5, each SPE *i* scans the vertices contained in the incoming *Qin* queues, and adds them to its private $bQnext_i$ if they had not been marked before. Despite the simple description, Step 5 is the most computationally expensive part of the algorithm, and it was the hardest to optimize. For sake of space efficiency, we choose to implement the *marked* data structure with a bitmap, stored in the local memory of each SPE. This bitmap is a boolean data structure where a single bit represents the status (marked or not marked) of one of the vertices owned by the current SPE. Given the limited size of the local store in the Cell BE architecture (256 kbytes), it is hard to allocate more than 160 kbytes for the bitmap. This limits the maximum graph size to 10 million vertices (i.e. the cumulative number of bits stored in the bitmaps of 8 SPEs) on one Cell BE processor. The algorithm can be simply generalized to larger graphs by *gang-scheduling* the graph exploration on subsets of a larger bitmap, but the discussion of this enhancement is beyond the scope of this paper.

Commit. Step 6 commits the content of $bQnext_i$ accumulated during the last execution of Step 5 and writes them to $Qnext_i$. $bQnext_i$ buffers are managed with a double-buffering technique as bQ_i and bG_i, and the same considerations made above apply. The only additional complexity is due to the fact DMA transfer must be aligned on a 128 byte boundary, and that the programmer must explicitly guarantee this alignment. Unlike what happens with bQ_i and bG_i, $bQnext_i$ is not naturally aligned. In fact, blocks from Q can be loaded with arbitrary alignment, so it is not enough to choose the size of bQ as a multiple of 128 bytes to guarantee the alignment of subsequent load operations. Similarly, bG loads adjacency lists which can be easily forced to begin at aligned locations at graph generation time. On the

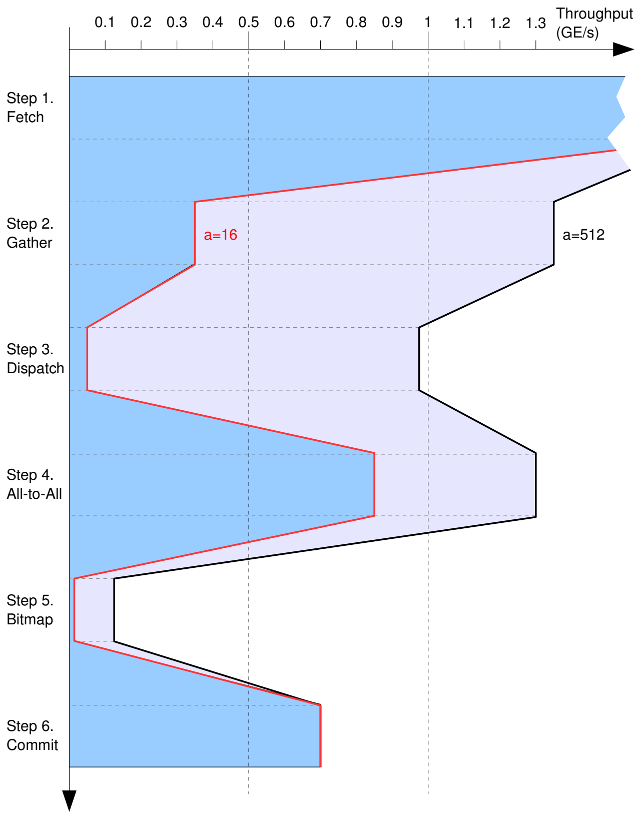

Figure 2. Throughput attainable by each step in the algorithm, per SPE (GE/s).

other hand, *bQnext* may contain an unpredictable number of vertices which may be misaligned. Appropriate techniques are needed to pad blocks of irregular size, and to rewrite the padding with new data at the subsequent commit steps avoiding unnecessary loads.

At the end of Step 6, a barrier synchronizes all the processing elements. The algorithm terminates when all SPEs have no vertices left in their Q_i queues. The actual implementation of this check is obtained via an *allreduce* primitive, which executes a distributed sum of the length of all the Q_i queues $\forall i$. If this sum is zero, the algorithm can terminate.

4 Performance Analysis and Optimization

Thanks to the modular software design described in the previous sections, we can easily develop an accurate analytical performance model of our application. This model has guided our implementation and various levels of performance optimization.

First, we derive lower and upper bounds on the performance of each step, on the basis of benchmarks performed on the Cell BE and simple models derived from the experimental results. This preliminary analysis exposes a primary bottleneck, the bitmap implementation, which is discussed in more detail later in this section.

Since any BFS implementation must visit all the edges of the given graph which are connected to the root vertex, a natural way to express the performance is through the number of edges visited per unit of time. We call this quantity *throughput*, we indicate it with the symbol Th and we measure it in edges per second (E/s). With ME/s and GE/s we indicate a million and a billion edges per second respectively.

As a *final* step of our algorithmic design, we release some of the strict synchronization bounds imposed by the BSP design, to fully overlap computation with on-chip and off-chip communication and achieve almost optimal performance. We consider this as a key point of our methodology: we allow the concurrent execution of these activities –arguably the most difficult part to debug and analyze, only *after* having a reasonably accurate understanding of all the components of our algorithm.

We now derive performance bounds of the maximum throughput achievable by each step of the algorithm. The entire algorithm can only be as fast as the slowest stage of the performance pipeline. The result of this analysis is the performance diagram described in Figure 2.

All the values have been either measured or analytically derived for a Cell processor with a clock running at 3.2 GHz. Whenever the throughput depends on the available bandwidth of a data-transfer operations, we have assumed the worst traffic conditions: i.e., all the 8 SPE are used, and they are all contending for the communication resources at the same time. When the bandwidth varies significantly depending on which block size is transferred, we have reported a meaningful minimum and maximum estimates.

Fetch. Step 1 is a transfer of a single, contiguous block from main memory to the local store. The available bandwidth depends on the size of this block. Figure 3 shows this dependence. The size of bQ in our implementation is always larger than 1024 bytes, which guarantees high bandwidth, and in steady-state conditions bQ is always full. Therefore, the available aggregate bandwidth is 22.06 Gbyte/s, i.e. 2.76 Gbyte/s per each SPE.

Step 1 is different from the others because it transfers vertex identifiers rather than adjacency lists. For each vertex identifier transferred in Step 1, the remaining steps will transfer an entire adjacency list, which will be arity times larger, on average. Therefore, Step 1 can transfer 689.38 M vertices/s. In the worst case, when the average arity $\bar{a} = 1$, this yields a throughput $Th = 689.38$ ME/s.

Gather. Step 2 is another get from main memory. Since it transfers a sparse set of adjacency lists, the most efficient way to perform the step is via a DMA list. Setting

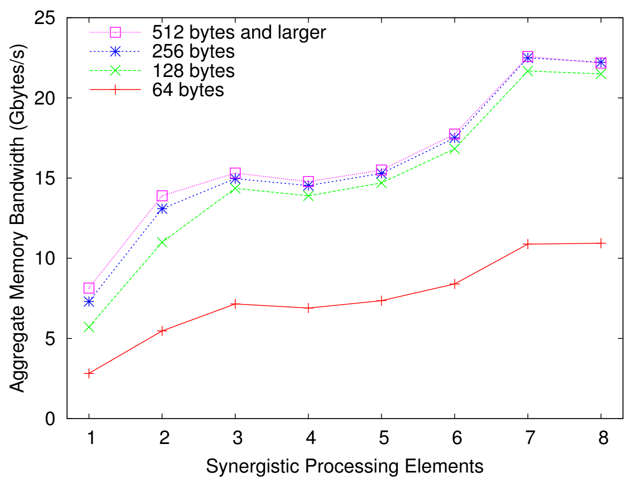

Figure 3. Aggregate bandwidth available when loading from main memory.

up the DMA list requires a negligible amount of computation. DMA lists can specify up to 2,048 transfers and, once set up, are handled by the Cell BE architecture without any additional operation or overhead. When the arity is large enough to transfer blocks of more than 512 bytes (i.e. $\bar{a} > 128$), the available bandwidth is 2.76 Gbyte/s (see Figure 3) with a sustained throughput of 689.38 ME/s. In the worst case, with blocks of 64 bytes, the bandwidth is 1.36 Gbyte/s, which yields $Th = 341.76$ ME/s.

Dispatch. Step 3 is a computational step. Appropriate design of the data structures reduces this step to a minor control portion, plus a local data transfer. Studied in isolation, this phase is able to process 42.34 ME/s with small graph arity ($\bar{a} = 16$), and 984.23 ME/s with larger arity ($\bar{a} = 512$).

All-to-all. Step 4 transfers each $Qout_{p,i}$ inside each SPE p into queue $Qin_{i,p}$ inside SPE i. Unnecessary transfers are avoided, i.e. when $p = i$. We prepare a DMA list where a communication guard appears after each queue, and we transfer it with the `mfc_putlb` intrinsic (put DMA list with barrier). This makes sure that the guard transfer is initiated only after the queue transfer is completed. To minimize the computational cost associated with this step, we set up this DMA list at initialization time, which is possible because the addresses of the Qin and $Qout$ queues in memory do not vary during execution. Assuming a queue cumulative size of 36 kbyte, and that queues are, on the average, exploited between 75% and 99.2% of their available capacity depending on the chosen arity (100% is not reachable because of the space occupied by data headers), the corresponding throughput is between 853.3 ME/s and 1.13 GE/s per SPE.

Bitmap. Step 5 is a computational step. This is the primary bottleneck of our implementation. Since the bitmap is the performance bottleneck of the entire algorithm, its optimization is crucial. Starting from a baseline implementation we explored 8 gradual refinements. Each refinement aims at improving the throughput by either removing overhead or exploiting potential sources of instruction-level or data-level parallelism provided by the Cell BE architecture. Thanks to these optimization, we have been able to lower the edge processing time from 96 to 26 clock cycles. Our best implementation is able to ensure a throughput between 35.17 ME/s (with $\bar{a} = 16$) and 113.73 ME/s (with $\bar{a} = 512$) per SPE. This version has been obtained using a combination of function inlining, selective SIMDization, loop unrolling, branch elimination through speculation and the use of restricted pointers.

Commit. Step 6 is a main memory communication. This results in a single transfer of a large block (> 512 byte) to main memory. This always allows for the maximum bandwidth, which is 2.76 Gbyte/s, leading to a throughput Th = 689.38 ME/s per SPE.

To summarize the results of this analysis, the maximum performance achievable by the algorithm is upper bounded by Step 5 between 35.17 and 113.73 ME/s per each SPE. A more realistic upper bound on the throughput can be obtained by considering not only Step 5, but Step 3 and 5 jointly, because these two computational phases cannot be overlapped. Their joint throughput $Th_{3,5}$ is

$$Th_{3,5} = \frac{Th_3 \cdot Th_5}{Th_3 + Th_5} .$$

The above formula yields new performance bounds for the overall algorithm between 19.21 ME/s (for $\bar{a} = 16$) and 101.95 ME/s (for $\bar{a} = 512$) per SPEs.

In the final step of our implementation we release some of the constraints of the BSP scheduling to fully overlap computation with on-chip and off-chip communication, as shown in Figure 4. It is worth noting that even this unconstrained version of the algorithm is completely deterministic, for a given input graph, root vertex and number of processing elements. This proved to be a major advantage during the functional and performance debugging.

In Figure 5, we can see that for every combination of problem parameters (e.g., arity and size of the graph) all the transfer latencies of the Gather and Commit phases together are less than the sum of the computational phases. Moreover the time required to transfer a single queue in the All-to-All phase is less than the time required to process it in the Bitmap phase.

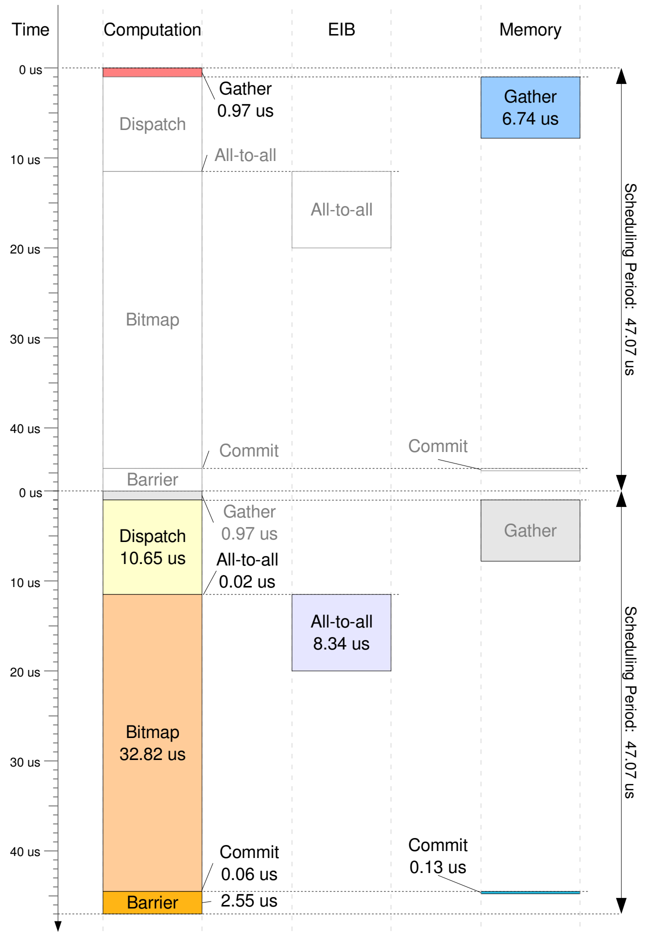

Figure 4. An iteration of the main loop (Steps 2–6) spans two scheduling periods.

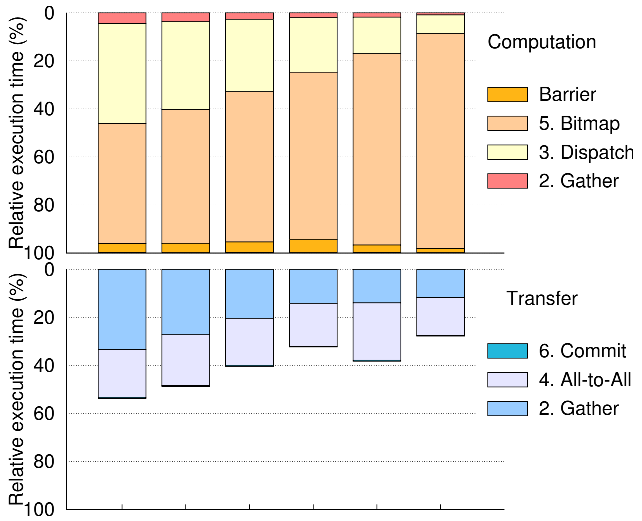

Figure 5. Time spent in computation (above) and data-transfer (below).

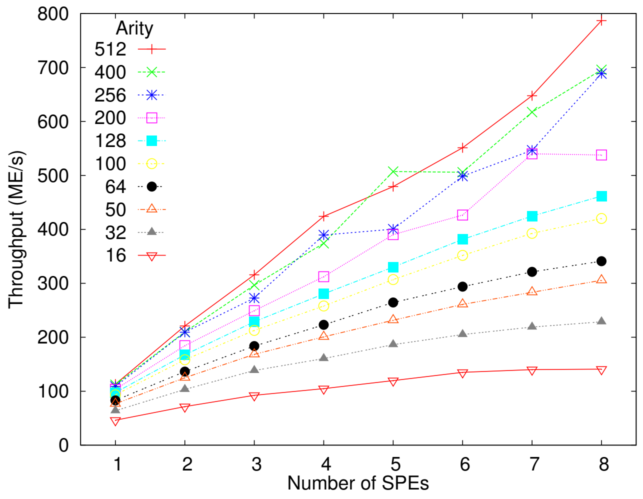

Figure 6. Throughput scaling for graph arity $\bar{a}$ varying from 16 to 512.

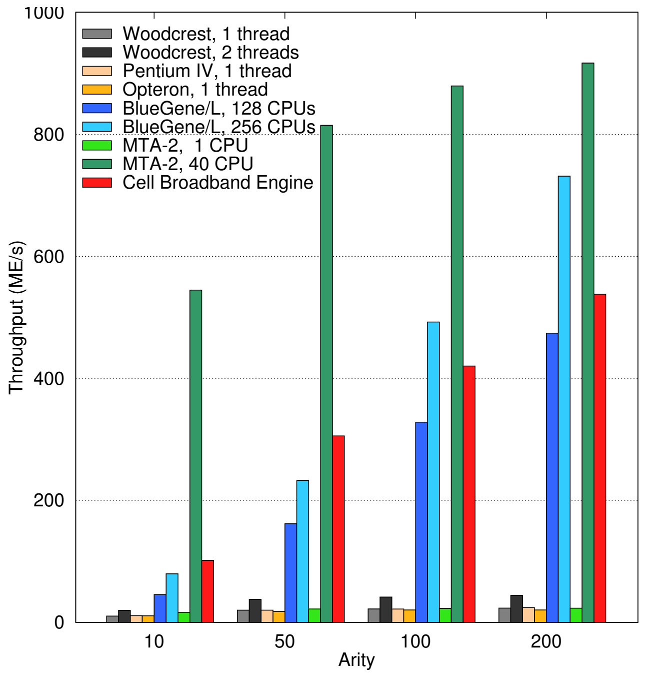

Figure 7. Performance comparison of our Cell BE implementation with other processors.

5 Experimental Results

This section describes the experimental results and compares these results with other architectures. We have implemented the algorithm in C language using the Cell BE, and compiled it with GNU GCC 4.0.2. We have run the experiments on an IBM DD3 blade with 2 Cell BE processors running at 3.2 GHz, 1 Gbyte of RAM and a Linux kernel version 2.6.16.

In the experiments we have measured the average throughput that our BFS exploration can deliver with different graph arities $\bar{a}$, and a variable number of SPEs N. To do that, we have executed our algorithm on randomly generated graphs. The arities of the vertices are independent, identically distributed random variables, taken from a uniform distribution in the range $[0...2\bar{a}]$. For each given $\bar{a}$, we have empirically determined the maximum graph size $|V|$ that can be allocated in main memory, without swapping to disk. It is worth noting that the performance of our implementation is *insensitive to the size of the random graphs*, as shown in Figures 5 and 4, because all on- and off-chip communication can be fully overlapped with computation. The only limits are the TLB addressability of the SPEs, 32 Gbytes using *huge page tables*, and the amount of physical memory available.

Figure 6 reports the results of these tests when $\bar{a}$ varies from 16 to 512, and N from 1 to 8. For each value of $\bar{a}$, the aggregate throughput is plotted as a function of N. Our implementation shows a good scaling behavior, which is virtually linear at high arities, and with a limited saturation effect at small ones.

Finally, in Figure 7 we compare our BFS implementation with other implementations, running on different architectures. For the BlueGene/L, we derive throughput values from Yoo et al. [11]. The results for the MTA-2 are taken from a personal communication with Feo [4] (Figure 7 reports data coming from John Feo). We also compare our implementation with an in-house single-processor BFS implementation running on the AMD Opteron and the Intel Pentium 4, and a scalable `pthread` implementation on the Intel Woodcrest.

When the graphs have low average arity ($\bar{a}$=10 in our experiments) the Cell BE shows a throughput equal to only 101.6 ME/s. This reduced performance is due to the extensive padding of the data structures, which need to be filled to quadwords.

Also, the smaller adjacency lists are, the less efficient is their transfer via DMA. In fact, with $\bar{a}$=10 adjacency lists occupy blocks around 64 bytes in size, which drops the aggregate memory bandwidth from 22 to 10 Gbps (see Figure 3). Nevertheless, we can still achieve complete overlapping of data transfers and computation. Conventional processors have little cache locality, and they are, on average, 9 times slower then the Cell BE. The best performance in this class is obtained by the 2 cores of the Intel Woodcrest which are between 5 and 12 times slower.

The comparison between the Cell BE and the MTA-2 and BlueGene/L is not an apple-to-apple one because of the limited amount of memory available on the Cell blade, only 1 Gbyte versus several Gbytes. This is mostly a technological limitation that will be addressed by future generations of Cell blades.

With fine arity, BlueGene/L combines the lack of cache locality with the communication overhead of small packets, and a single Cell is two orders of magnitude faster, reaching the same scaled performance of 325 BlueGene/L processors with arity $\bar{a}$=50.

The performance of our algorithm is also compared with a BFS implementation on the Cray MTA-2 provided by John Feo [4]. With $\bar{a}$=10 a Cell BE is approximately equivalent to 7 MTA-2 processors. Larger arities enhance the effectiveness of the SIMD-ized bitmap manipulations in the Cell BE. With $\bar{a}$ = 200, the Cell BE is 22 times faster than the Pentium and the Woodcrest (12 times faster than two Woodcrest cores), 26 times faster than the AMD Opteron, and at the same level of performance of 128 BlueGene/L processors and an MTA-2 system with 23 processors.

6 Conclusions

Together with an unprecedented level of performance, multi-core processors are also bringing an unprecedented level of complexity in software development. We see a clear shift of paradigm from classical parallel computing, where parallelism is typically expressed in a single dimension (i.e, local vs. remote communication, or scalar vs. vector code), to the complex, multi-dimensional parallelization space of multi-core processors, where several levels of control and data parallelism *must* be exploited in order to gain the expected performance.

This paper describes a figurative journey we took to obtain the highest level of performance of the BFS graph exploration on the Cell Broadband Engine Processor (Cell BE). On this journey, we discovered many important properties of the Cell BE, such as the importance of a careful algorithmic design that takes in explicit consideration a hierarchy of working sets and the data orchestration between these levels.

The experimental evaluation has shown that the Cell BE can obtain impressive performance in this class of algorithms: a performance speedup of one order of magnitude when compared to other commodity and special-purpose processors, reaching two orders of magnitude with BlueGene/L.

We believe that the key to achieve this level of performance is a clear understanding of the characteristics of each component of the algorithm. Our task has been simplified by the adoption of a Bulk-Synchronous model that has allowed us to implement and test the various steps of the algorithm in a modular fashion, and to develop an accurate performance model to determine upper and lower bounds on the run time of each part of the algorithm.

The work presented in this paper shows that it is possible to achieve high performance and ease of programming by attacking the real problem first –the unmanageable complexity derived by many concurrent activities.

Acknowledgments

We would like to thank Deborah Gracio, Troy Thompson, Dave Thurman, Mike Kistler, John Feo, David A. Bader, Karen L. Karavanic and Samuel W. Williams for their help and fruitful suggestions.

References

[1] D. A. Bader and K. Madduri. Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2. In *Proc. Intl. Conf. on Parallel Processing (ICPP'06)*, Columbus, OH, August 2006.

[2] A. Clauset, M. E. J. Newman, and C. Moore. Finding Community Structure in Very Large Networks. *Physical Review E*, 6(70), December 2004.

[3] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. J. Knight, and A. DeHon. GraphStep: A System Architecture for Sparse-Graph Algorithms. In *Proc. Symposium on Field-Programmable Custom Computing Machines (FCCM'06)*, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[4] J. Feo. Optimized BFS Algorithm on the MTA-2 Architecture. Personal Communication, November 2006.

[5] J. Fernández, E. Frachtenberg, and F. Petrini. BCS MPI: A New Approach in the System Software Design for Large-Scale Parallel Computers. In *Proc. Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SuperComputing'03)*, Phoenix, AZ, November 2003.

[6] J. Mellor-Crummey and M. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(1):21–64, February 1991.

[7] D. S. Nikolopoulos and T. S. Papatheodorou. The Architectural and Operating System Implications on the Performance of Synchronization on ccNUMA Multiprocessors. *Intl. Journal of Parallel Programming*, 29(3), October 2001. 249–282.

[8] V. Subramaniam and P.-H. Cheng. A Fast Graph Search Multiprocessor Algorithm. In *Proc. of the Aerospace and Electronics Conf. (NAECON'97)*, Dayton, OH, July 1997.

[9] L. G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, 33(8):103–111, 1990.

[10] W. A. Wuld and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *ACM Computer Architecture News*, 23(1), March 1995.

[11] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek. A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L. In *Proc. Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SuperComputing'05)*, Seattle, WA, November 2005.