Efficient Breadth-First Search on the Cell/B.E. Processor

Daniele Paolo Scarpazza, Oreste Villa, Fabrizio Petrini, Member, IEEE

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

In this paper we describe the challenges involved in designing a Breadth-First Search (BFS) algorithm for the Cell/B.E. processor. The proposed methodology combines a high-level algorithmic design that captures the machine-independent aspects, to guarantee portability with performance to future processors, with an implementation that embeds processor-specific optimizations. Using a fine-grained global coordination strategy derived by the Bulk-Synchronous Parallel (BSP) model, we have determined an accurate performance model that has guided the implementation and the optimization of our algorithm.

Our experiments show an almost-linear scaling over the number of used synergistic processing elements in the Cell/B.E. platform, and compares favorably against other systems. On graphs which offer sufficient parallelism, the Cell/B.E. is typically an order of magnitude faster than conventional processors such as the AMD Opteron and the Intel Pentium 4 and Woodcrest, and custom-designed architectures, such as the MTA-2 and BlueGene/L.

Index Terms—Multi-core processors, Parallel Computing, Cell Broadband Engine, Parallelization Techniques, Graph Exploration Algorithms, Breadth-First Search, BFS.

I. INTRODUCTION

Over the last decade, high-performance computing has ridden the wave of commodity technology, building clusters that were able to leverage the tremendous growth in processor performance fueled by the commercial world. As this pace slows down, processors designers are facing complex problems when increasing gate density, reducing power consumption and designing efficient memory hierarchies. The scientific and industrial communities are looking for alternative solutions that can keep up with the insatiable demand of computing cycles and yet have a sustainable market.

Traditionally, performance gains in commodity processors have come through higher clock frequencies, an exponential increase in the number of devices integrated on a chip, and other architectural improvements. Power consumption is increasingly becoming the driving constraint in processor design: processors are more and more power-limited rather than area-limited. Current general purpose processors are optimized for single-threaded workloads, and spend a significant amount of resources (and power) to extract parallelism. Common techniques are out-of-order execution, register renaming, dynamic schedulers, branch prediction, reorder buffers, etc. Experiencing diminishing returns, processor designers are turning their attention to thread-level, VLIW and SIMD parallelism. Explicitly parallel techniques, where a user or a compiler express the available parallelism as a set of cooperating threads, offer a more efficient means of converting power into performance than techniques which speculatively discover the implicit—and often limited—instruction level parallelism hidden in a single thread.

Another important trend in computer architecture is the implementation of highly integrated chips. Several design avenues have been explored both in the academia, such as the Raw multiprocessor [47] and TRIPS [43], and in the industrial world, with notable examples being the AMD Opteron, IBM Power5, Sun Niagara, Intel Montecito and others [32], [27], [28], [34].

The Cell Broadband Engine (Cell/B.E.) processor, jointly developed by IBM, Sony and Toshiba is a heterogeneous chip with nine cores (one control processor coupled with eight independent synergistic processing elements) capable of massive floating point processing, optimized for compute-intensive workloads and broadband, rich media applications. The processing power of the Cell/B.E., that with a frequency of 3.2 GHz peaks at 204.8 single-precision Gflops/second, has not passed unobserved.

Intel has also recently announced its Tera-Scale research initiative, by connecting eighty simple cores on a simple test chip. These chips will be able to deliver, most likely by the end of the decade, teraflop-level performance. Figure 1 provides some intuition on what the near future might look like (courtesy of Doug Carmean, “The Evolution of Computing Architectures”, Intel slideshow).
A. The Programming Challenge

To fully exploit the potential of multi-core processors, we need a significant shift of paradigm in the software development process. Unfortunately, for some classes of applications, this implies the radical redesign of algorithms.

Together with the initial excitement of early evaluations, several concerns have emerged. More specifically, there is an interest in understanding the complexity of developing new applications and parallelizing compilers, whether there is a clear migration path for existing legacy software, and what fraction of the peak performance can actually be achieved by real applications.

Several recent works have provided insight into these problems. In fact, to develop efficient multi-core algorithms one must understand in depth multiple algorithmic and architectural aspects. The list includes (1) identifying the available dimensions of parallelism, (2) mapping parallel threads of activity to a potentially large pool of processing and functional units, (3) using simple processing cores with limited functionalities, (4) coping with the limited on-chip memory per core and (5) the limited off-chip memory bandwidth (when compared to the rich resources available on-chip), and many others.

The programming landscape of these advanced multi-core processors is rather sophisticated. Many similarities appear with cluster computing: in both fields we need to extract explicit parallelism, deal with communication, and take care of how threads of computation are mapped onto the physical machine. But there are also differences, mostly in the data orchestration between processing cores and main memory, which demand a fresh look to the problem, and the design of new algorithms.

B. Graph Exploration Algorithms

Many areas of science (genomics, astrophysics, artificial intelligence, data mining, national security and information analytics, to quote a few) demand techniques to explore large-scale data sets which are represented by graphs. Among graph search algorithms, Breadth-First Search (BFS) is probably the most common, and a building block for a wide range of graph applications. For example, in the analysis of semantic graphs the relationship between two vertices is expressed by the properties of the shortest path between them, given by a BFS search. BFS is also the basic building block for best-first search, uniform-cost search, greedy-search and A\textsuperscript{*}, which are commonly used in motion planning for robotics. A good amount of literature deals with the design of BFS solutions, either based on commodity processors or dedicated hardware. Some recent publications describe successful parallelization strategies of list ranking and phylogenetic trees on the Cell/B.E. But, to the best of our knowledge, no studies have investigated how effectively the Cell/B.E. can be employed to perform a BFS search on large graphs, and how it compares with other commodity processors and supercomputers. The Cell/B.E. with its impressive amount of parallelism (multiple cores, SIMD) promises interesting speedups in the BFS exploration of the many graph topologies which can benefit from it.

Searching large graphs poses difficult challenges, because the vast search space is combined with the lack of spatial and temporal locality in the data access pattern. In fact, few parallel algorithms outperform their best sequential implementations on clusters due to long memory latencies and high synchronization costs. These issues call for even more attention on multi-cores like the Cell/B.E., because of the memory hierarchy which must be managed explicitly.

C. Contribution

This paper provides four primary contributions.

1) A detailed description of a BFS graph exploration algorithm for multi-core processors. We put emphasis on its peculiar characteristics, such as the data-flow and data layout, the explicit management of a hierarchy of working sets and the data orchestration between them.

2) An experimental evaluation of the algorithm on the Cell/B.E. processor that discusses how its different components are integrated, and an accurate comparison with other architectures. The goal is to provide an insight on the performance impact of the possible architectural and software 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) methodology.

4) An arsenal of low-level optimizations to exploit the full potential of the Cell/B.E. processor.

Our methodology is based on the following cornerstones:

- a high-level algorithmic design which focuses on essential machine-independent aspects, which guarantee the portability of performance to other multi-core processors,
- a machine-dependent refinement of the initial algorithm which embeds the specific optimizations of the chosen target multi-core architecture,
- a BSP-style global coordination, used by both the machine-dependent and machine-independent parts, that allows implementing and validating the individual steps of the algorithm in a modular way,
- 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 the enforcement of a deterministic behavior within the steps of the algorithm that, for a given input set and number of processing elements, can be re-played as a sequential program.

The proposed algorithm blends the theoretical analysis of graph algorithms of Dehne et al. on constrained parallel models (Coarse-Grained Multicomputer (CGM) and BSP), with the parallel implementation of the Boost Graph Library, to take advantage of the specific hardware capabilities of multi-core processors.
As already pointed out by academic [31] and industrial projects [11], non-determinism should be judiciously and carefully introduced where needed, or eliminated altogether when possible, to attack the programming wall of multi-core processors. We think that the optimal level of performance and ease of programming can be achieved at the same time, by attacking the real problem first: i.e., the unmanageable complexity caused by many concurrent activities.

The remainder of this paper is organized as follows. Section II discusses the high-level, machine-independent part of our parallel BFS algorithm for multi-core processors. In Section III we describe the machine-dependent parallelization on the Synergistic Processing Elements (SPEs) of Cell/B.E. Section IV gives a detailed performance analysis of the algorithm, pinpoints its bottlenecks and proposes several types of optimizations. Section V describes the experiments we have performed to measure the performance of our algorithm on a Cell/B.E. board, and Section VI compares these results against the ones provided by other, state-of-the-art processors and supercomputers. Finally, Section VII concludes the paper. The appendix is devoted to describe the low-level optimizations of the performance-critical sections of our algorithm.

II. THE BREADTH-FIRST SEARCH ALGORITHM

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. Then, we describe a simplified parallel BFS algorithm as a collection of cooperating shared-memory threads. Finally, we refine it into an algorithm that explicitly manages a hierarchy of working sets.

A. Notation

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 $E_v$ its adjacency list, i.e. the set of neighboring vertices of $v$ (or neighbors, for short), such that $E_v = \{w \in V : (v,w) \in E\}$. We indicate with $d_v$ the degree, i.e. the number of elements $|E_v|$. We will denote as $d$ the average degree of the vertices in a graph, $d = \frac{\sum_{v \in V} |E_v|}{|V|}$.

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.

B. Sequential BFS algorithm

Algorithm 1 presents a sequential BFS algorithm. 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.

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 $Q_{next}$. $Q_{next}$ is the set of vertices to visit in the next level. At the end of the exploration of a level, the content of $Q_{next}$ is assigned to $Q$, and $Q_{next}$ is emptied. The algorithm terminates when there are no more neighbors to visit, i.e. $Q$ is empty (line 17).

The algorithm does not visit a vertex more than once. To do so, it maintains an array of boolean variables marked, $\forall v \in V$, where each variable marked, tells whether vertex $v$ has already been visited. Neighboring vertices are added to $Q_{next}$ only when they have not been marked yet.

C. A Parallel BFS

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, provided that access to the array marked is protected with some synchronization mechanism, such as a multiple locks. This is the conventional solution in a cache-coherent shared-memory machine with uniform memory access time and a limited number of hardware threads. Unfortunately, it does not scale well with larger numbers of PEs.

We adopt a different approach, illustrated in Algorithm 2. We partition $V$ in disjoint sets $V_i$, one per 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 2 all steps are globally synchronized across the PEs. Horizontal lines denote synchronization points, 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 private $Q_i$ and $Q_{next}$, and its partition of the marked array. Additionally, each

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; $Q_{next}$, vertices to be explored in the next level; marked, array of booleans: marked, $\forall i \in [1...|V|];$

1. \textbf{for all} $i \in [1...|V|] :$ marked$_i = \text{false}$
2. marked$_0$ = \text{true}
3. level $\leftarrow 0$
4. $Q \leftarrow \{r\}$
5. \textbf{repeat}
6. $Q_{next} \leftarrow \{\}$
7. \textbf{for all} $v \in Q$ \textbf{do}
8. \textbf{for all} $n \in E_v$ \textbf{do}
9. \textbf{if} marked$_n$ = \text{false} \textbf{then}
10. marked$_n$ = \text{true}
11. $Q_{next} \leftarrow Q_{next} \cup \{n\}$
12. \textbf{end if}
13. \textbf{end for}
14. \textbf{end for}
15. $Q \leftarrow Q_{next}$
16. level $\leftarrow$ level $+ 1$
17. \textbf{until} $Q = \{\}$
Algorithm 2 bulk-synchronous parallel version of a breadth first graph exploration.

Input: \( G(V,E) \), graph;
\( r \), root vertex;
\( P \), available processing elements (PE);
\( V_1, V_2, \ldots, V_p : (\bigcup_{i=1}^p V_i) = V \);
\( \forall (i, j) \in [1..P]^2 : V_i \cap V_j = \{ \} \) if \( i \neq j \);
Variables:
\( Q_i \), vertices to be explored in the current level;
\( Q_{next,i} \), vertices to be explored in the next level;
\( Q_{marked,i} \), vertices marked during exploration;
\( Q_{in,i} \), incoming queues;
\( Q_{out,i} \), outgoing queues;
\( Q_{next} \), vertices to be explored in the next level;
\( Q_{marked} \), vertices marked during exploration;
\( Q_{in} \), incoming queues;
\( Q_{out} \), outgoing queues;

Processing element \( i \):
1. \( \text{level} \leftarrow 0 \)
2. \( Q_i \leftarrow \{\} \)
3. \( \forall v \in V_i : \text{marked}_i \leftarrow \text{false} \)
4. \( \text{if } r \in V_i \text{ then } \)
5. \( Q_i \leftarrow \{r\} \)
6. \( \text{marked}_i \leftarrow \text{true} \)
7. \( \text{end if} \)
8. \( \) repeat in lockstep across the processing elements:
9. \( \) for all \( (p,v) \in [1..P] \times Q_i \) do
10. \( Q_{in,p} \leftarrow \{\} \)
11. \( Q_{next,p} \leftarrow \{\} \)
12. \( \) end for
13. // Gather and Dispatch
14. \( \) for all \( n \in E \), \( (v,E_i) \in (\bigcup_{p=1}^P Q_{in,p}) \) do
15. \( \) if \( \text{marked}_i = \text{false} \) then
16. \( \text{marked}_i \leftarrow \text{true} \)
17. \( Q_{next,i} \leftarrow \{\} \)
18. \( \) end if
19. \( \) end for
20. \( \) until \( \forall p \in [1..P] : Q_p = \{\} \)

Note: horizontal lines indicate barrier-synchronization points.

PE \( i \) has a set of private outgoing and incoming queues, called \( Q_{out,i} \), \( Q_{in,i} \), \( Q_{next,i} \), and \( Q_{marked,i} \). Through these queues, PEs can forward the vertices to their respective owners.

At initialization time, the root vertex \( r \) is assigned to its owner \( Q_i \). During the exploration, each PE \( i \) examines the vertices \( v \) in \( Q_i \) and dispatches the vertices in \( E \), which belong to PE \( p \) to \( Q_{out,p} \). Then, when all the PEs have completed this phase, an all-to-all personalized exchange takes place, and the contents of each \( Q_{out,p} \) are transferred to \( Q_{in,p} \). This exchange delivers the vertices to their respective owners.

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

D. A Parallel BFS for Multi-core Processors

The parallel algorithm just presented (Algorithm 2) does not consider size limitations of \( Q_i \), \( Q_{next} \), \( Q_{in} \), and \( Q_{out} \). If private data structures are entirely allocated in the local storage of each PE, Algorithm 2 can overflow the memory available in each core. Local memories in a multi-core processor are small, and the issue won’t disappear in the future: whereas chip integration promises tens of cores on a die, on-chip memory will not increase as fast [50]. For this reason, application developers should design their algorithms taking explicitly into account application working sets and data orchestration among 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 \( Q_{next,i} \) are still in main memory, while \( Q_{marked} \), \( Q_{in} \), and \( Q_{out} \) are now allocated in the local memory. The algorithm does not access \( Q_i \) 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 \( Q_{next,i} \), but to a small buffer \( bQ_{next} \), which is then committed to \( Q_{next} \) via an explicit operation (see line 39). Adjacency lists \( E_v \) are also explicitly loaded into the local data structure \( bG \) during the gather step (see line 15).

The algorithm can operate on graphs of arbitrary size and vertex degree, 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 \( Q_{marked} \) is at least as large as \( Q_{next,i} \). Overflows of \( bG \) can be managed at graph creation by splitting the single adjacency list in multiple lists having the same vertex, and incorporating minor algorithmic changes to load-balance the exploration of these heavy vertices across multiple PEs. Each partition of the \( marked \) variables 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 than 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.

III. IMPLEMENTATION OF THE ALGORITHM

In this section we describe how we parallelized Algorithm 3 on the Synergistic Processing Elements (SPEs) of Cell/B.E. We can consider this part as the machine-dependent one of our parallelization process, which in the future could find its best place in a run-time system or a compiler. At this stage we analyze lower-level details, such as remote DMAs, double buffering, data alignment, etc.
Figure 3 presents a schematic overview of the steps of the 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/B.E. 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 a 32-bit 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 not enough bits are available.

In detail, a vertex code has two fields, the vertex identifier (which is $i$) and the vertex length, which is an encoded representation of $|E_i|$. 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.** Step 3 splits the adjacency lists previously gathered into the respective $Qout$ queues. To expedite this task, we encode the adjacency lists as shown in Figure 2. 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 sub-list is padded to a multiple of a 4 words size to enable dispatch of it one quadword at a time (a quadword is the size of registers and of 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 $Qout_{i,j}$; therefore, $Qin_{i,j}$ is simply an alias of $Qout_{i,j}$.

Step 4 requires an appropriate synchronization mechanism to detect the presence of valid data in a $Qin$. An 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.

This is done by preparing a DMA list where the transfers of queues and guards are interleaved, and employing the \texttt{mfc\_putlb} intrinsic (put DMA list with barrier), which guarantees that each transfer in the list is completed before the next one is issued\footnote{Simpler synchronization mechanisms, e.g. tagging the data payload with a special marker, are insufficient. They lead to potential race conditions and consequent data corruption. In fact, the on-chip network of the Cell/B.E. may re-arrange segments of a DMA transfer in arbitrary order, causing the marker to reach the receiving end before the whole transfer has been completed.}

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 \texttt{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 \(Q_{in,i}\) queues, and adds them to its private \(Q_{next,i}\), 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 have implemented 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/B.E. 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/B.E. processor. The algorithm can be simply generalized to larger graphs by gang-scheduling —scheduling in a coordinated way the graph exploration on subsets of a larger bitmap which is loaded under program control in different phases— but the discussion of this enhancement is beyond the scope of this paper.

**Commit.** Step 6 commits the content of \(bQ_{next}\), accumulated during the last execution of Step 5 and writes them to \(Q_{next}\). \(bQ_{next}\) buffers are managed with a double-buffering technique as \(bQ_i\) and \(bG_i\), and the same considerations made about them 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 \(bG_i\), \(bQ_{next}\) 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 other hand, \(bQ_{next}\) may contain an unpredictable number of vertices which may be misaligned. We used some simple heuristics to pad blocks of irregular size, and to rewrite the padding with new data during the subsequent commit steps, avoiding unnecessary local memory 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 \(Q\) queues. The actual implementation of this check is obtained via an \texttt{allreduce} \footnote{Simpler synchronization mechanisms, e.g. tagging the data payload with a special marker, are insufficient. They lead to potential race conditions and consequent data corruption. In fact, the on-chip network of the Cell/B.E. may re-arrange segments of a DMA transfer in arbitrary order, causing the marker to reach the receiving end before the whole transfer has been completed.} primitive, which executes a distributed sum of the length of all the \(Q\) queues.

The encoding format adopted for the adjacency lists: (a) each adjacency list of a vertex in the graph is encoded in split sub-lists. In each of them, all the vertices belong to a given SPE. A header specifies the vertex code, and the length and offset of each sub-list; (b) how adjacency sub-lists are encoded in \(Q_{out}\) (and \(Q_{in}\)) queues, as a result of the dispatch step. "4 words" marks a block of data which must be padded to a quadword, "4* words" marks a block of data which must be padded to a quadword or a multiple of quadwords, depending on the unroll factor chosen for the Bitmap step.
The data flows involved in the different steps of the algorithm.

∀i. If this sum is zero, the algorithm has completed.

IV. PERFORMANCE ANALYSIS AND OPTIMIZATION

We determine lower and upper bounds on the performance of each step, on the basis of benchmarks performed on the Cell/B.E. This analysis exposes a primary bottleneck, the bitmap implementation, whose optimization is discussed 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 algorithm can only be as fast as the slowest stage of the performance pipeline. Also, the bounds we present in this section are solely derived from computer-architecture constraints: in fact, the nature of the input graph and its topological properties may set additional, lower performance boundaries, as we show in Section V.

The result of this analysis is the performance diagram described in Figure 4. All the values have been either measured or analytically derived for a Cell/B.E. 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 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 aggregate bandwidth is a function of the block size, as shown in Figure 5. 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. It can transfer 689.38 M vertices/s (i.e. the above bandwidth divided by 4, which is the number of bytes per vertex identifier). For each vertex identifier transferred in Step 1, the remaining steps will transfer an entire adjacency list, which will be $\bar{d}$ times larger, on average. In the worst case, when the $\bar{d}$=1, this yields a throughput $Th = 689.38$ ME/s.

**Gather.** Step 2 is another get from main memory, implemented with a DMA list. DMA lists can specify up to 2,048 independent transfers that can be handled by the Cell/B.E. architecture without any additional operation or overhead. When the average degree is large enough to transfer blocks of more than 512 bytes (i.e. $\bar{d} > 128$), the available bandwidth is 2.76 Gbyte/s (see Figure 5) 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. The optimized design of the data structures simplifies 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 degree ($\bar{d}$=16), and 984.23 ME/s with larger degree ($\bar{d}$=512).

**All-to-all.** Step 4 transfers each $Qout_{p,i}$ inside each SPE $p$ into queue $Qin_{p,i}$ 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. Figure 6 shows how the latency required to complete this step varies with the number of SPEs. Since the cumulative queue size is independent from the number of SPEs, when fewer SPEs are employed this results in more efficient, larger transfers. With 8 SPEs, the latency is 8.1 µs. 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 degree (100% is not reachable because of the 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/B.E. architecture. These optimizations have lowered the edge processing time from 96 to 26 clock cycles. Our best implementation ensures a throughput between 35.17 ME/s (with $\bar{d}$=16) and 113.73 ME/s (with $\bar{d}$=512) per SPE. We have obtained this level of performance using a combination of function inlining, selective SIMDization, loop unrolling, branch elimination through speculation and the use of restricted pointers, as discussed in more detail below. The impact of optimizations is outlined in Table I.

**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 set 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 $T_{h3,5}$ is given by

$$T_{h3,5} = \frac{T_{h3} \cdot T_{h5}}{T_{h3} + T_{h5}},$$

which formula yields new performance bounds between 19.21 ME/s (for $\bar{d}=16$) and 101.95 ME/s (for $\bar{d}=512$) per SPEs.

A. Overlapping Computation and Data Transfers

In the final stage of our implementation we release the constraints of the BSP scheduling to fully overlap computation on-chip and off-chip communication. This leads to a pipelined schedule for the main loop of our algorithm (Steps 2–6 plus a barrier), which is represented in Figure 7. Note that an iteration of the main loop spans two scheduling periods, but at the end of each period an iteration is completed, as typical with pipelined designs. Steps depicted in color belong to the current iteration, while the remaining ones do not (the white ones belong to the previous iteration, and the grey ones to the next). The time values reported in the figure are worst-case

and refer to an exploration with $d=128$; they are insensitive to the number of vertices in the graph.

Note that this version of the algorithm is still completely deterministic: a given input graph, root and number of processing elements always results in the same visit. This proved to be a major advantage during the functional and performance debugging.

When the average degree of the input graph varies, the duration of each step varies accordingly, and the relative impact of each step also changes. For example, larger degrees make the bitmap step more and more predominant. We represent the fraction of the time spent in the various steps when $\bar{d}$ changes in Figure 8 (steps are color-coded as in Figure 7; results are insensitive to the number of vertices). Data transfer latencies are always smaller than the computational latencies, irrespectively of the degree, thus ensuring overlapping of computation and data transfer as desired.

V. Experimental Results

We have implemented the algorithm in C language using the Cell/B.E., and compiled it with GNU GCC 4.0.2. We have run the experiments on an IBM blade with two Cell/B.E. DD3 processors running at 3.2 GHz, 1 Gbyte of RAM and a Linux kernel version 2.6.16.

In the experiments, we have measured throughput and scalability of our BFS algorithm. The input sets we have considered are two classes of graphs:

- synthetic graphs whose vertex degrees are random variables extracted from a uniform distribution over the range $[0...2\bar{d}]$, for a set of chosen values of $\bar{d}$;
- graphs from real-world problems in a variety of domains (structural engineering, computational fluid dynamics,
electromagnetics, thermodynamics, materials, acoustics, computer graphics, robotics, optimization, circuit simulation, networks, economic modeling, theoretical and quantum chemistry, chemical process simulation, etc.).

The graphs in the first class offer a useful benchmarking ground, where the impact of variable degree can be studied in isolation from other effects, such as load imbalance. In fact, the uniform distribution of degrees guarantees a good load balancing among the processing elements. Instead, the graphs in the second class (100 randomly-chosen graphs from the University of Florida Sparse Matrix Collection [10]) exhibit diverse topological and statistical properties, and their exploration is subject to the joint impact of two factors: the amount of available parallelism (which can be small, due to small average degree), and the load imbalance (which can be significant, due to power-law degree distributions). We have not considered worst-case graph instances, e.g., chains of degree-1 vertices. These graphs do not offer any parallelism and locality exploitable by our implementation.

Tests on the first class of graphs are reported in Table I. For each considered $d$ between 16 and 512, we generated and explored the largest synthetic graphs (in terms of number of vertices) that would fit the available main memory on the blade (1 GB). Results are in good agreement with the upper bounds estimated in Section IV, especially when $d$ is large, so that the impact of the Bitmap step is predominant, as shown in Figure 8. For example, when $d=512$ and $P=8$, the aggregate throughput is 786.85 ME/s, i.e. 98.36 ME/s per SPE, which is very close to the estimate of 101.95 ME/s obtained in Section IV. The 3.5% difference accounts for the computational part of the other steps, the overhead of flow control and the load imbalance. When $d=16$, the aggregate throughput is 140.99 ME/s, i.e. 17.62 ME/s per SPE, which is close to the estimate of 19.21 ME/s (the difference is 8.6%). Within the assumptions made above, the performance of our implementation is independent from the number of vertices in the graph.

Table I shows how the throughput scales when $P$ varies from 1 to 8. Our implementation shows a good scaling behavior, which is virtually linear at large values of $d$, with a limited saturation effect at small ones. When $d$ is as low as 10, the throughput is only 101.6 ME/s. This reduced performance is due to the insufficient parallelism, which causes padding to be introduced in the data structures which are subject to SIMD operations. Also, the smaller adjacency lists are, the less efficient is their transfer via DMA. In fact, with $d=10$ adjacency lists occupy blocks around 64 bytes in size, which lowers the aggregate memory bandwidth from 22 to 10
Gbytes/s (see Figure 5). Nevertheless, complete overlapping of data transfers and computation is still ensured. Moreover, adjacency sub-lists also need to be padded to a quadword or its multiples, depending on the unroll factor (as shown in Figure 2). Therefore, when more SPEs are used, the length of adjacency sub-lists decreases, and they need a higher amount of padding, thus leading to lower performance, as Figure 9 shows.

On the second class of graphs, these performance considerations hold too. Additionally, the non-uniformly distributed degree of the vertices can cause load imbalance, and subsequent performance degradation.

We assume as a reference the performance reported for the first class of graphs. Figure 10 reports the fraction of reference performance (%) that our algorithm achieves when exploring 100 graphs in the second class, with a single SPE (points in red) and with 8 SPEs (points in green). Although graphs of similar degree can exhibit significantly different performance, a clear trend correlates large degrees with high efficiency: the coefficient of correlation between $\log(\bar{d})$ and $Th$ is 0.727.

Finally, in Figure 11 we compare our BFS implementation with others, running on different architectures. Values for BlueGene/L and MTA-2 come respectively from Yoo et al. [51] and from a personal communication with Feo [19]. We have measured values for the Intel Pentium 4 and AMD Opteron with an in-house, single-processor BFS implementation, and values for the Intel Woodcrest with a similar, in-house a scalable pthread-based implementation. For sake of consistency, all values represent the peak performance provided by each implementation, i.e. in the case of the Cell/B.E., graphs are from the first class presented above.

Conventional processors have little cache locality, and they are, on average, 9 times slower then the Cell/B.E. 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/B.E. and the MTA-2 and BlueGene/L is not an apple-to-apple one because of the limited amount of memory available on a Cell/B.E. blade, only 1 Gbyte versus several Gbytes. This is mostly a technological limitation that will be addressed by future generations of blades.

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

Our BFS implementation compares well with one provided by John Feo for the Cray MTA-2 [19]. With $\bar{d}$=10, a Cell/B.E. is approximately equivalent to 7 MTA-2 processors. A larger $\bar{d}$ enhances the effectiveness of the SIMD bitmap operations in the Cell/B.E. With $\bar{d}$=200, the Cell/B.E. is 22× faster than the Pentium and the Woodcrest (12× faster than two Woodcrest cores), 26× 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.

VI. DISCUSSION

Our analysis, summarized in Table IV, suggests that high performance comes at the price of increased hardware complexity or human effort (roughly expressed in lines of code) required to develop the software application.

Given a fixed amount of real estate on a chip, one can adopt the traditional approach of the Intel Woodcrest, with fewer coarse-grained computational cores, or the more aggressive design of the Cell/B.E. with a higher number of fine-grained cores. With the current state of the art, the former solution allows two cores, and the latter 8 cores. It is easy to expect, as technology progresses, to see the same architectural dilemma, fewer core with HW speculation, deep pipelines, standard cache-coherence protocols or a larger number of simpler cores.
TABLE III
How our BFS implementation on the IBM Cell/B.E. processor compares with other reference architectures.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>ME/s 10 Speed-up</th>
<th>ME/s 50 Speed-up</th>
<th>ME/s 100 Speed-up</th>
<th>ME/s 200 Speed-up</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>IBM Cell/B.E. @3.2 GHz</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 PPE + 8 SPE</td>
<td>101.6</td>
<td>–</td>
<td>305.7</td>
<td>–</td>
</tr>
<tr>
<td>Intel Woodcrest @3.0 GHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>· 1 core</td>
<td>10.5</td>
<td>9.68x</td>
<td>20.1</td>
<td>15.22x</td>
</tr>
<tr>
<td>· 2 cores</td>
<td>19.8</td>
<td>5.13x</td>
<td>37.7</td>
<td>8.11x</td>
</tr>
<tr>
<td>Intel Pentium 4 HT @3.4 GHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>· 1 core</td>
<td>11.1</td>
<td>9.13x</td>
<td>20.2</td>
<td>15.10x</td>
</tr>
<tr>
<td>AMD Opteron 250 @2.4 GHz</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>· 1 core</td>
<td>11.0</td>
<td>9.26x</td>
<td>17.9</td>
<td>17.05x</td>
</tr>
<tr>
<td>Cray MTA-2 @220 MHz [2]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>· 1 CPUs</td>
<td>17.7</td>
<td>5.75x</td>
<td>17.7</td>
<td>17.31x</td>
</tr>
<tr>
<td>· 40 CPUs</td>
<td>512.0</td>
<td>0.20x</td>
<td>512.0</td>
<td>0.60x</td>
</tr>
<tr>
<td>Cray MTA-2 @220 MHz [19]</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>· 1 CPUs</td>
<td>16.7</td>
<td>6.09x</td>
<td>22.0</td>
<td>13.93x</td>
</tr>
<tr>
<td>· 40 CPUs</td>
<td>544.7</td>
<td>0.19x</td>
<td>814.8</td>
<td>0.38x</td>
</tr>
<tr>
<td>IBM BlueGene/L</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>· 128 CPUs</td>
<td>45.7</td>
<td>2.22x</td>
<td>162.0</td>
<td>1.89x</td>
</tr>
<tr>
<td>· 256 CPUs</td>
<td>79.8</td>
<td>1.27x</td>
<td>232.7</td>
<td>1.31x</td>
</tr>
</tbody>
</table>

TABLE IV
A comparison among the multiple architectural and programming solutions which have been employed to tackle the BFS problem, in terms of features and effort involved.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Parallel Algorithm</th>
<th>Programming Language</th>
<th>Type of Parallelism</th>
<th>Parallel Data Decomposition</th>
<th>Explicit Synchronization</th>
<th>Lines of Code</th>
<th>Programming Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>AMD Opteron 250</td>
<td>no</td>
<td>C</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>100</td>
<td>Low</td>
</tr>
<tr>
<td>· 1 CPU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Pentium 4 HT</td>
<td>no</td>
<td>C</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>100</td>
<td>Low</td>
</tr>
<tr>
<td>· 1 CPU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Woodcrest</td>
<td>no</td>
<td>C</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>100</td>
<td>Low</td>
</tr>
<tr>
<td>· 1 core</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>· 2 cores</td>
<td>yes</td>
<td>C + pthread</td>
<td>Thread</td>
<td>No</td>
<td>Yes [2] / No [19]</td>
<td>200</td>
<td>Medium Low</td>
</tr>
<tr>
<td>Cray MTA-2</td>
<td>yes</td>
<td>C + pragmas</td>
<td>Thread</td>
<td>No</td>
<td>Yes [2] / No [19]</td>
<td>100</td>
<td>Medium Low</td>
</tr>
<tr>
<td>· 128 HW threads</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BlueGene/L</td>
<td>yes</td>
<td>C + MPI</td>
<td>Message Passing</td>
<td>Yes</td>
<td>Yes</td>
<td>?</td>
<td>Medium</td>
</tr>
<tr>
<td>· 2 cores</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cell/B.E.</td>
<td>yes</td>
<td>C + intrinsics</td>
<td>Mixed</td>
<td>Yes</td>
<td>Yes</td>
<td>600</td>
<td>Medium High</td>
</tr>
</tbody>
</table>

TABLE V
Performance achieved at BFS by commodity processors, when the size of the graph is comparable with the cache memory size.

<table>
<thead>
<tr>
<th>Average degree</th>
<th>10</th>
<th>16</th>
<th>32</th>
<th>50</th>
<th>100</th>
<th>200</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertices</td>
<td>105200</td>
<td>16536</td>
<td>32768</td>
<td>21000</td>
<td>10480</td>
<td>5240</td>
</tr>
<tr>
<td>AMD Opteron 250</td>
<td>27.87</td>
<td>36.44</td>
<td>47.63</td>
<td>53.57</td>
<td>64.73</td>
<td>71.57</td>
</tr>
<tr>
<td>(1 MB cache)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Pentium 4 HT</td>
<td>24.58</td>
<td>45.85</td>
<td>76.25</td>
<td>96.88</td>
<td>125.58</td>
<td>152.77</td>
</tr>
<tr>
<td>(2 MB cache)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Woodcrest (1 core)</td>
<td>42.59</td>
<td>65.41</td>
<td>106.15</td>
<td>132.21</td>
<td>167.65</td>
<td>196.00</td>
</tr>
<tr>
<td>(4 MB cache)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Intel Woodcrest (2 cores)</td>
<td>70.50</td>
<td>101.76</td>
<td>193.79</td>
<td>210.35</td>
<td>257.39</td>
<td>337.20</td>
</tr>
<tr>
<td>(4 MB cache)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
that require SW speculation and explicit data orchestration in the local storage.

The programming advantage of the Woodcrest is clearly outlined on Table V [V]. A simple program of roughly 100 lines of code can efficiently implement a graph exploration algorithm on a single core. In order to use both cores, the user needs to extract explicit thread parallelism with a shared-memory programming model, in our case using the pthread library. The user must also explicitly deal with the potential race conditions that can happen when multiple threads update data structures of the same vertex at the same time, a consequence of the inherent non-determinism of the graph exploration. Even if the parallelization is very efficient, with almost optimal speed-up on two threads, the aggregate processing rate is lower than the Cell/B.E., in some cases by more than an order of magnitude.

Table V provides some insight on this performance gap. With small graphs, that can fit in the L2 cache, the Woodcrest (and also the other conventional processors) can achieve a much higher throughput, reaching 196 ME/s with one and 337 ME/s with two cores.

The HW speculation of these processors, throttled by the limited number of outstanding memory requests that can be issued at any given time, is not as powerful as the software speculation, carefully organized in DMA transfers that can be issued at a user-specified granularity. In fact, the Cell/B.E. could process graphs at a wire-speed of a few GE/s (in Figure 6 [6] we have seen that it is possible to achieve aggregate communication bandwidth in excess of 20 Gbytes/s when using DMAs as small as 128 bytes), if it were not limited by the processing rate of the bitmap manipulation, which is the bottleneck of our implementation.

On the other hand, the Woodcrest and the other conventional processors can execute a bitmap update in 4.6 ns, 2 times faster than the Cell/B.E., with a significantly simpler implementation. This suggests a potential direction for performance improvements in future generations of Cell/B.E. processors and other multi-core processors.

Another interesting comparison is between the Cell/B.E. and the MTA-2 multi-threaded machine. The MTA-2 is a cache-less architecture that hides the memory latency with a large number of memory requests, 1,024 per processor, that can be issued by 128 HW threads. The absence of a layered memory hierarchy makes the performance of this machine relatively insensitive to the average degree of the graph, as shown in Table III [III]. The graph exploration algorithm presented by Bader [2] is remarkably simple, and only incrementally more complex than the sequential pthread version. Using #pragma, the user needs to identify the loops that contain enough parallelism, estimate the number of threads that can be executed in parallel and consider possible race conditions, that are avoided using presence bits in each memory locations [2] or a int_fetch_add [19]. Using these spare bits, each memory location can have associated a producer-consumer semantic, and HW threads can synchronize at the fine granularity of the single memory word.

The clock cycle of the MTA-2 is only 220 MHz, with an expected clock cycle of 500 MHz in the upcoming Cray Threadstorm processor that is the building block of the Cray XMT [20]. Surprisingly, the Cell/B.E. compares well with a 20-CPU cluster with 2,560 HW threads, providing similar performance with graphs of relatively large average degree. It is worth noting that many of these architectural features have been adopted by General Purpose Graphical Processing Units (GPGPU), such as the NVidia GeForce 8800 [39], that can take advantage of a larger market, and therefore employ more aggressive and expensive integration technologies.

The work on BlueGene/L done by Yoo et al. [51] take the difficult step of parallelizing the graph exploration on a distributed memory machine, using an explicit message-passing paradigm. The level of complexity is substantially higher than the other implementations, and it also requires a sophisticated pre-processing of the graph. Quite remarkably, a single Cell/B.E. processor is two orders of magnitude faster than the BlueGene/L, across various graph configurations, in particular those with small average degree where there is very little data locality. In a specific case, with $d = 50$, the scaled performance of a single Cell/B.E. is equivalent to 336 BlueGene/L processors.

The programming effort that is needed to achieve optimal performance on the Cell/B.E. is still high. This is in part due to the lack of a clear and simple abstract machine model to develop new algorithms. And in part to the lack of run-time systems and compilers that can help the program development and optimization. In our case, for example, we had to explicitly implement barrier synchronization and reduction, collective
communication primitives that are typically included in a runtime libraries such as MPI, and optimize at almost assembly level the computational kernels of our application. In Table IV we can see that the actual algorithm required 600 lines of code, plus 600 lines to implement run-time functionalities such as barrier, allreduce, initialization and termination.

VII. CONCLUSIONS
Together with an unprecedented level of performance, multicore processors are also bringing an unprecedented level of complexity in terms of 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.

With this paper we proved that, for the specific case of the breadth-first search graph exploration, it is possible to tame the algorithmic and software development process and achieve at the same time an impressive level of performance.

The explicit management of the memory hierarchy, with emphasis on the local memories of the multiple cores, is a fundamental aspect that needs to be captured by the high-level algorithmic design, to guarantee portability of performance across existing and future multi-core architectures. Programmability greatly benefits from the separation of computation and communication, and from a Bulk-Synchronous Parallel (BSP) parallelization, which also leads to accurate performance models and guides the implementation and optimization effort.

Our experiments show that the Cell/B.E. can obtain high performance in this class of algorithms: a speedup of one order of magnitude when compared to other commodity and special-purpose processors, reaching two orders of magnitude with BlueGene/L.

A major strength of the Cell/B.E. is the possibility of overcoming the memory wall: the user can explicitly orchestrate the memory traffic by pipelining multiple DMA requests to main memory. This is a unique feature that is not available on other commodity multiprocessors, that cannot efficiently handle working sets that overflow the cache memory. The major limitation is the extraction of the SIMD parallelism, a non-trivial effort with multiple concurrent activities.

VIII. ACKNOWLEDGEMENTS
We thank Deborah Gracio, Troy Thompson and Dave Thurman for their support. We thank Mike Kistler of the Austin IBM Research Laboratory for his insightful technical advices, and John Feo for the description of his BFS parallelization on Cray MTA-2

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.


