Scalable Graph Exploration on Multicore Processors

Virat Agarwal[1] Fabrizio Petrini[1] Davide Pasetto[2] David A. Bader[3]

[1]IBM TJ Watson, Yorktown Heights, NY 10598, USA
[2]IBM Computational Science Center, Dublin, Ireland
[3]College of Computing, Georgia Tech, Atlanta, GA 30332, USA

viratagarwal@us.ibm.com, fpetrin@us.ibm.com, pasetto_davide@ie.ibm.com

***Abstract*—Many important problems in computational sciences, social network analysis, security, and business analytics, are data-intensive and lend themselves to graph-theoretical analyses. In this paper we investigate the challenges involved in exploring very large graphs by designing a breadth-first search (BFS) algorithm for advanced multi-core processors that are likely to become the building blocks of future exascale systems. Our new methodology for large-scale graph analytics 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. We present an experimental study that uses state-of-the-art Intel Nehalem EP and EX processors and up to 64 threads in a single system. Our performance on several benchmark problems representative of the power-law graphs found in real-world problems reaches processing rates that are competitive with supercomputing results in the recent literature. In the experimental evaluation we prove that our graph exploration algorithm running on a 4-socket Nehalem EX is (1) 2.4 times faster than a Cray XMT with 128 processors when exploring a random graph with 64 million vertices and 512 millions edges, (2) capable of processing 550 million edges per second with an R-MAT graph with 200 million vertices and 1 billion edges, comparable to the performance of a similar graph on a Cray MTA-2 with 40 processors and (3) 5 times faster than 256 BlueGene/L processors on a graph with average degree 50.**

I. Introduction

With the advent of multicore processors, and their widespread use in the data centers of numerous industrial, business and governmental institutions, parallelism is here to stay.

One of the biggest challenges in effectively using this unprecedented level of parallelism is in supplying high-level and simple approaches to develop parallel applications. There are many hurdles with developing such applications, but certainly one of the main difficulties is dealing with the communication costs between processing nodes and within the memory hierarchy of each node. The typical way programmers and algorithmic designers improve performance is through locality, trying to re-use data as much as possible. This is a complex task because there are different types of locality: hierarchical networks, distributed, shared and local caches, local and non-local memory, and various cache-coherence effects. In an ideal spectrum of parallel applications, those that have high locality, rely on limited communication and closely match the cache hierarchy, can readily take advantage of existing multicore processors and accelerators [1], [2]. At the other end of the spectrum there are problems, such as graph exploration, with very little data reuse and a random access pattern. Searching large graphs poses difficult challenges, because the potentially vast data set is combined with the lack of spatial and temporal locality in the access pattern. In fact, few parallel algorithms outperform their best sequential implementations on clusters due to long memory latencies and high synchronization costs [3]. Additionally, these difficulties call for more attention if dealing with commodity multicore architectures because of the complexity of their memory hierarchy and cache-coherence protocols.

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, in most cases, represented by graphs. In these areas, search algorithms are determinant to discover nodes, paths, and groups of nodes with desired properties. Among graph search algorithms, Breadth-First Search (BFS) is often used as 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. Applications in community analysis often need to determine the connected components of a semantic graph [4], [5], [6], [7], and connected components algorithms [8] often employ a BFS search. BFS is also the basic building block for best-first search, uniform-cost search, greedy-search and A*, which are commonly used in motion planning for robotics [9], [10].

A good amount of literature deals with the design of BFS solutions, either based on commodity processors [11], [12] or special purpose hardware [13], [14], [15], [16]. Some recent publications describe successful parallelization strategies of list ranking [17] and phylogenetic trees on the Cell BE [18]. Recently Xia et al. [19] have achieved high-quality results for BFS explorations on state-of-the-art Intel and AMD processors.

This paper describes the challenges we have faced to implement an efficient BFS algorithm on the latest families of Intel Nehalem processors, including a newly-released system based on the 8-core Nehalem EX. The choice of a conceptually simple algorithm, such as the BFS exploration, allows for a complete, in-depth analysis of the locality and communication protocols.

SC10 November 2010, New Orleans, Louisiana, USA 978-1- 4244-7558-2/10/$26.00

The primary contribution of this paper is the development of a simple and scalable BFS algorithm for multicore, shared-memory systems that can efficiently parse graphs with billions of vertices and beyond. To the best of our knowledge, as discussed in detail in Section IV, the results presented in this paper outperform in absolute value any other commodity or specialized architecture on the same classes of problems, reaching aggregate processing rates that in some cases exceed a billion edges per second. The main aspects of the algorithm are:

- an innovative data layout that enhances memory locality and cache utilization through a well-defined hierarchy of working sets;
- a software design that decouples computation and communication, keeping multiple memory requests in flight at any given time, taking advantage of the hardware capabilities of the latest Nehalem processors;
- an efficient, low-latency channel mechanism for inter-socket communication that tolerates the potentially high delays of the cache-coherent protocol;
- the extreme simplicity: the proposed algorithm relies on a handful of native mechanisms provided by the Linux operating system, (namely `pthreads`, the atomic instructions `__sync_fetch_and_add()` and `__sync_or_and_fetch()`, and the thread and memory affinity libraries).

Other contributions include an in-depth performance evaluation that considers different classes of graphs and analyzes scalability, processing rate sensitivity to vertex arity and graph size, and a comprehensive comparative analysis of the related work in graph exploration in Section IV.

By combining our innovative algorithmic design with new architectural features, such as the capability of keeping many memory requests in flight that for the first time is available in commodity processors, we are able to achieve rates that are competitive with supercomputing results in the recent literature. More specifically, a 4-socket Nehalem EX is:

- 2.4 times faster than a Cray XMT with 128 processors when exploring a random graph with 64 million vertices and 512 million edges [15],
- capable of processing 550 million edges per second with an R-MAT graph with 200 million vertices and 1 billion edges, comparable to the performance of a similar graph on a Cray MTA-2 with 40 processors [16],
- 5 times faster than 256 BlueGene/L processors on a graph with average degree 50 [20].

Finally the paper exposes important properties of the latest families of Nehalem processors and gives a fresh look to the recently announced Nehalem EX. In our experiments we used a four-socket eight-core system (a total of 64 threads, based on dual-threading per each of the 32 cores) equipped with 256 GB of memory.

Our work provides a valuable contribution for application developers, by identifying a software path that potentially can be followed by other applications, in particular in the security and business analytics domains. We expect that the 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 streaming and irregular applications. We believe that the results presented in this paper can be used as algorithmic and architectural building blocks to develop the next generation of exascale machines.

The rest of this paper is organized as follows. Section II describes the speeds and feeds of the Nehalem EP and EX systems used in our experimental evaluation. The essential aspects of the algorithmic design are summarized in Section III, while a rich body of experimental results is presented in Section IV. Finally, some concluding remarks are given in Section V.

II. System Architecture and Experimental Platforms

We focus our design and experimental evaluation on two systems: a dual-socket Xeon 5500 (Nehalem-EP) and a four socket Xeon 7560 (Nehalem-EX). Figure 1 provides a visual overview of the system architecture of the two processors, and describes how larger systems with 4 and 8 Nehalem-EX sockets can be assembled to build a shared-memory SMP; table I summarizes system parameters.

The Xeon 5570 (Nehalem-EP) is a 45nm quad core processor. Each core has a private L1 and L2 cache, while the L3 cache is shared across the cores on a socket. Each core supports Simultaneous Multi Threading (SMT), allowing two threads to share processing resources in parallel on a single core. Nehalem EP has a 32KB L1, 256KB L2 and a 8MB L3 cache. As opposed to older processor configurations, this architecture implements an inclusive last level cache. Each cache line contains 'core valid bits' that specify the state of the cache line across the processor. A set bit corresponding to a core indicates that the core may contain a copy of this line. When a core requests for a cache line that is contained in the L3 cache, the bits specify which cores to snoop, for the latest copy of this line, thus reducing the snoop traffic. The MESIF cache-coherence protocol extends the native MESI protocol to include 'forwarding'. This feature enables forwarding of unmodified data that is shared by two cores to a third one.

The Xeon 7560 (Nehalem-EX), on the other hand, contains 8 cores per socket. The configuration of each core is identical to the Xeon 5600 but they operate at lower frequency. This architecture expands the shared L3 cache size to 24 MB per processor with a fast kilobit-wide ringbus between the different cache segments to boost access speed. Nehalem EX also implements two-way SMT per processing core and contains 4 DDR3 channels per chip, each capable of simultaneous read and write transactions, effectively doubling memory bandwidth. The 4-channel QPI interconnect at 6.4 GigaTransactions per second gives over a 100 GB/sec bandwidth to the other neighbouring sockets in a blade. In Table I we can see some interesting architectural trade-offs. The Nehalem EP can run

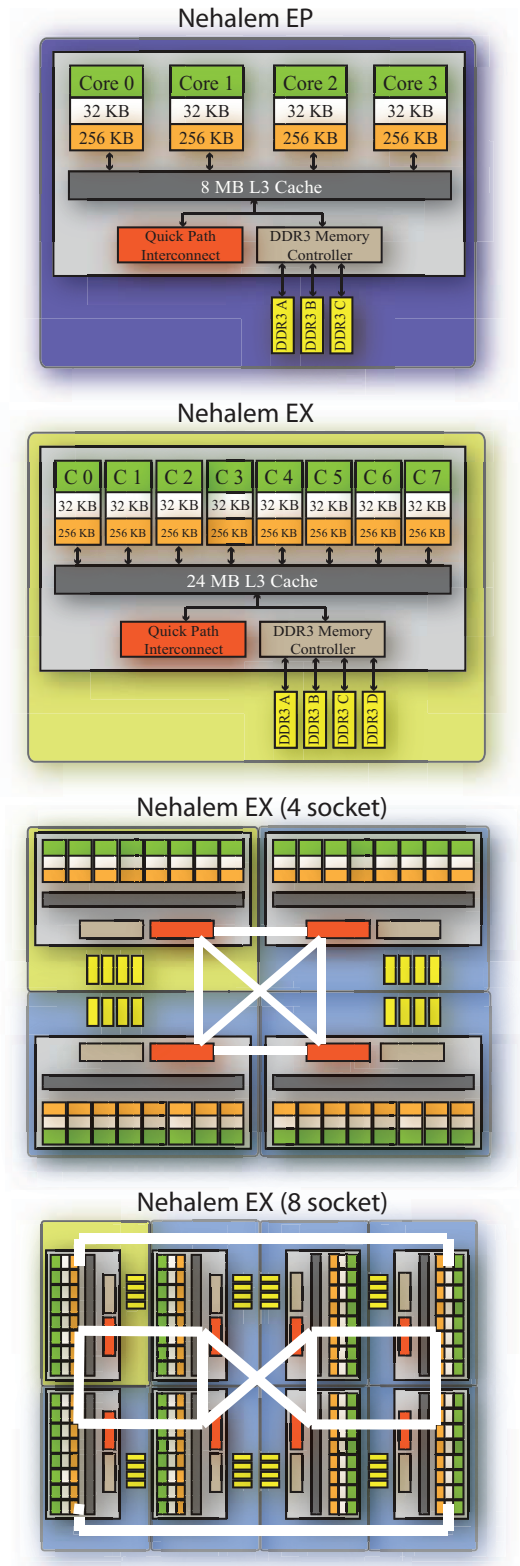


Fig. 1: Nehalem EP and EX: architectural overview and topology of 4- and 8-core Nehalem-EX systems.

at higher frequencies (high-end EP processors can reach 3.4 GHz), while the top frequency for an EX is only 2.26 GHz. The EP has three DDR3 memory channels for four cores, while the EX has four channels for eight cores, with a less favorable communication to computation ratio.

Processors	Nehalem-EP	Nehalem-EX
Core affinities	Proc 0 : 0-3 & 8-11 Proc 1 : 4-7 & 12-15	Proc 0 : 0-7 & 32-39 Proc 1 : 8-15 & 40-47 Proc 2 : 16-23 & 48-55 Proc 3 : 24-31 & 56-63
Cores per socket	4	8
Core frequency	2.93 GHz	2.26 GHz
L1 cache size	32 KB/32 KB	32 KB/32 KB
L2 cache size	256KB	256 KB
L3 cache size	8 MB	24 MB
Cache line size	64 Bytes	64 Bytes
Memory type	3 channels per socket, DDR3-1066	4 channels per socket, DDR3-1066

TABLE I: System configuration

Figure 2 reports the results of a simple benchmark. We consider a collection of data arrays of increasing size, ranging from 4KB to 8GB, that are accessed in read-only mode by a single core using a pseudo-random pattern. The core issues a batch of up to 16 memory requests and then waits for the completion of all of them before continuing to the next iteration.

The graph clearly shows the well known performance impact of the memory hierarchy: a sequence of performance-degrading steps that happen when we overflow each level of cache memory. As expected, by narrowing the working set size to fit in one of caches we can greatly increase performance.

The graphs also shows an important property of the new Nehalem processors: we can hide the memory latency by keeping a number of read requests in flight, as traditionally done by multi-threaded architectures [16], [15]. Surprisingly, with a simple software pipelining strategy we can increase by a factor of eight the number of transactions per second: for example, with a working set of 8MB, the memory subsystem can satisfy up to 160 millions reads per second, and with 2 GB we can achieve 40 millions of random reads per second. In accordance with the results presented in [21], we have experimentally determined that the maximum number of outstanding requests is about 10 for both Nehalem EP and EX. When we consider the aggregate behavior of all cores in the socket and we add SMT threads we can keep up to 50 and 75 requests in flight, respectively for the Nehalem EP and EX.[1] So, we can take advantage of one of the pillars of multi-threaded processors with commodity processors.

In Figure 3 we refine the previous experiment by running atomic fetch-and-adds, another important building block of our

[1]The maximum number of outstanding requests is influenced by the working set size and type of memory operation. While a full analysis of the memory pipelining properties is beyond the scope of this paper, we have noticed that the degree of memory pipelining is not a bottleneck for the most optimized version of our BFS algorithm, allowing linear scaling in both Nehalem architectures.

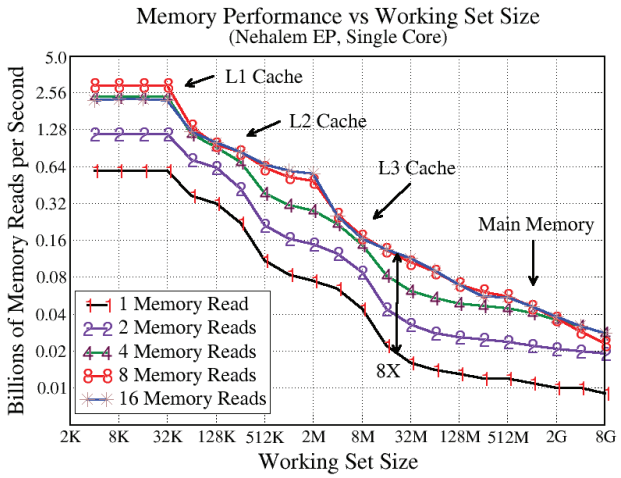


Fig. 2: Impact of memory pipelining, Nehalem EP.

algorithmic design. In this case we used a buffer of fixed size, 4MB, which is shared by an increasing number of threads mapped on two distinct Nehalem EP sockets.

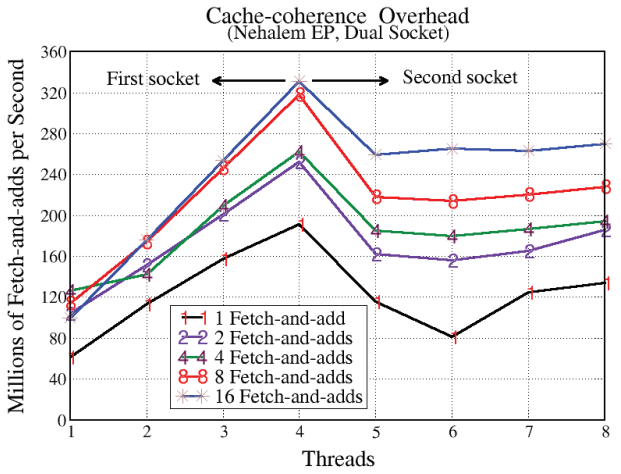


Fig. 3: Processing rates with Fetch-and-add and a dual socket configuration.

We observe that atomic ops cannot be pipelined as effectively as memory reads, mostly because the implementation of the primitives relies on the `lockb`[2] assembly instruction that locks the access to part of the memory hierarchy. More interesting is the performance gap when we transition from 4 to 5 threads, crossing the socket boundary. In this case the coherency traffic and the locking instructions limit the scalability of the access pattern: using 8 cores on two sockets, we achieve the same processing rate of only 3 cores on a single socket.

The insight provided by these two simple experiments is that, while a read-only access pattern can be easily scaled across multiple sockets relying on the native memory pipelining units, more sophisticated patterns that put pressure on the cache-coherency protocol require an innovative algorithmic solution.

[2] `lockb` is the basic instruction to implement atomic operations on x86 processors, so the results can be directly generalized to other primitives such as read-and-or, that we use in the graph exploration.

III. BFS Algorithms

In this section we present the methodology that we used to parallelize the BFS algorithm. We first introduce the notation employed throughout the rest of this paper and a simplified parallel version of BFS. Then, we refine the algorithm and introduce several optimizations that explicitly manage the hierarchy of working sets and enable scaling across cores and sockets.

A graph $G(V,E)$ is composed of a set of vertices V and a set of edges E. 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. Vertices are visited in *levels*: when a vertex is visited at level l, it is also said to be at a distance l from the root. When we visit the adjacency list of vertex u, for each vertex v not already visited, we set the parent of v to be u or $P[v] \longleftarrow u$.

Algorithm 1 presents an initial, simplified parallel BFS algorithm. At any time, CQ (current queue) is the set of vertices that must be visited at the current level. Initially CQ is initialized with the root r (see line 4). At level 1, CQ will contain the neighbours of r, at level 2, it will contain these neighbour's neighbours (the ones that have not been visited in levels 0 and 1), and so on. The algorithm maintains a next queue NQ, containing the vertices that should be visited in the next level. After reaching all nodes in a BFS level, the queues CQ and NQ are swapped. The high-level description Algorithm 1 exposes the nature of the parallelism, but abstracts several important details. For example in the assignment in line 10-12 and 8 must be executed atomically in order to avoid race conditions.

Algorithm 1 Parallel BFS algorithm: high-level overview.

Input: $G(V,E)$, source vertex r
Output: Array $P[1..n]$ with $P[v]$ holding the parent of v
Data: CQ : queue of vertices to be explored in the current level
NQ : queue of vertices to be explored in the next level

```
1  for all v ∈ V in parallel do
2      P[v] ⟵ ∞;
3  P[r] ⟵ 0;
4  CQ ⟵ Enqueue r;
5  while CQ ≠ φ do
6      NQ ⟵ φ;
7      for all u ∈ CQ in parallel do
8          u ⟵ Dequeue CQ;
9          for each v adjacent to u in parallel do
10             if P[v] = ∞ then then
11                 P[v] ⟵ u;
12                 NQ ⟵ Enqueue v;
13     Swap(CQ, NQ);
```

Algorithm 2 refines the original design by adding atomic operations (lines 11, 15 and 18) and two important optimizations.

Algorithm 2 Parallel BFS algorithm for a single socket configuration.

Input: $G(V,E)$, source vertex r
Output: Array $P[1..n]$ with $P[v]$ holding the parent of v
Data: $Bitmap[v]$: bit set to 1 if vertex v is visited, otherwise 0
CQ : queue of vertices to be explored in the current level
NQ : queue of vertices to be explored in the next level
Functions: *LockedDequeue*(Q) : Returns the front element of the queue Q and updates the front pointer atomically
LockedReadSet(a,val) : Returns the current value of a and sets it to val atomically
LockedEnqueue(Q,val) : Inserts val to the end of the queue and updates the pointer atomically

```
1  for all v ∈ V in parallel do
2      P[v] ⟵ ∞;
3  for i ⟵ 1..n in parallel do
4      Bitmap[i] ⟵ 0;
5  P[r] ⟵ 0;
6  CQ ⟵ Enqueue r;
7  fork;
8  while CQ ≠ φ do
9      NQ ⟵ φ;
10     while CQ ≠ φ in parallel do
11         u ⟵LockedDequeue (CQ);
12         for each v adjacent to u do
13             a ⟵ Bitmap[v];
14             if a = 0 then
15                 prev ⟵ LockedReadSet (Bitmap[v],1);
16                 if prev = 0 then
17                     P[v] ⟵ u;
18                     LockedEnqueue (NQ, v);
19     Synchronize;
20     Swap(CQ, NQ);
21 join;
```

The first optimization is the use of a bitmap (line 4 in Algorithm 2) to mark the vertices during the visit. While the access pattern is still random across all vertices, this greatly reduces the working set size. For example, in 4MB we can store all the visit information for a graph with 32 million vertices. In Figure 2, we can see that this can improve the processing rate (number of reads per unit of time) by at least a factor of four.

A more subtle optimization is performed in lines 13 and 14: in this case we avoid the potentially expensive atomic in line 15, i.e., *LockedReadSet* (a `__sync_or_and_fetch()` in the actual implementation), by first checking whether the vertex has already been visited. It is worth noting that the bit assigned in line 13 may be overwritten by another thread, so to avoid race-conditions we still need to perform an atomic operation. But as shown in Figure 4, the number of atomic operations is much lower than the number of queries in the later stages of the BFS exploration.

Algorithm 3 includes two important final optimizations. As shown in Figure 3, a random access pattern that requires atomic memory updates cannot scale efficiently across multiple sockets due to heavy traffic for line invalidation

Algorithm 3 Multicore Multi Socket Parallel BFS algorithm

Input: $G(V,E)$, source vertex r, n number of vertices (multiple of number of sockets)
Output: Array $P[1..n]$ with $P[v]$ holding the parent of v
Data: $Bitmap[v]$: bit set to 1 if vertex v is visited, otherwise 0
$CQ[s]$: queue of vertices to be explored in the current level for socket s
$NQ[s]$: queue of vertices to be explored in the next level for socket s
$SQ[s]$: queue for inter-socket communication channel for socket s, resides in memory of s, stores tuple (v,u), where u is the father of v
Functions: *LockedDequeue*(Q) : Returns the front element of the queue Q and updates the front pointer atomically
LockedReadSet(a,val) : Returns the current value of a and sets it to val atomically
LockedEnqueue(Q,val) : Inserts val to the end of the queue and updates the pointer atomically
GetTotalSockets() : returns the total number of sockets used in the algorithm
DetermineSocket(v) : returns the socket ID on which graph node v resides

```
1  sockets =GetTotalSockets();
2  Partition graph, allocate ns = n/sockets nodes to each socket
   Partition P and Bitmap corresponding to the graph partition, such that
   if graph node v ∈ socket s then both P[v] and Bitmap[v] ∈ socket s;
3  for v ⟵ 1 to n do
4      P[v] ⟵ ∞;
5      Bitmap[v] ⟵ 0;
6  P[r] ⟵ 0;
7  for s ⟵ 1 to sockets do
8      CQ[s] ⟵ φ;
9      NQ[s] ⟵ φ;
10 CQ[DetermineSocket(r)] ⟵ Enqueue r;
11 fork;
12 this = GetMySocket();
13 while CQ[this] ≠ φ do
14     while CQ[this] ≠ φ do
15         u ⟵LockedDequeue (CQ[this]);
16         for each v adjacent to u do
17             s ⟵ DetermineSocket(v);
18             if s = this then
19                 a ⟵ Bitmap[v];
20                 if a = 0 then
21                     prev ⟵ LockedReadSet (Bitmap[v],1);
22                     if prev = 0 then
23                         P[v] ⟵ u;
24                         LockedEnqueue (NQ[this], v);
25             else
26                 LockedEnqueue (SQ(s),(v, u));
27     Synchronize;
28     while SQ(this) ≠ φ do
29         (v, u) ⟵ LockedDequeue (SQ(this));
30         a ⟵ Bitmap[v];
31         if a = 0 then
32             prev ⟵ LockedReadSet (Bitmap[v],1);
33             if prev = 0 then
34                 P[v] ⟵ u;
35                 LockedEnqueue (NQ[this], v);
36     Synchronize;
37     Swap(CQ[this], NQ[this]);
38     NQ[this] ⟵ φ;
39 join;
```

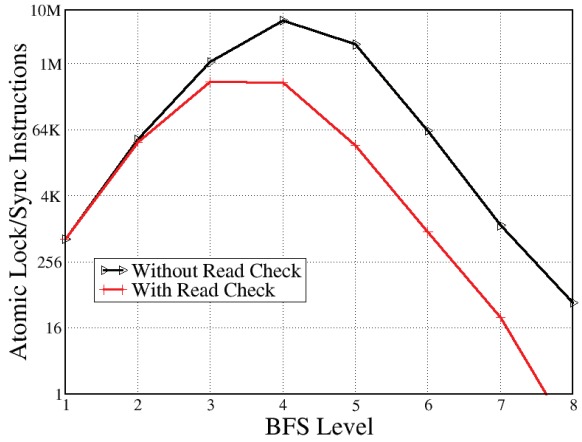


Fig. 4: Number of bitmap accesses and atomic operations in a BFS search, random uniform graph with 16 millions of edges, and average arity 8. By using a simple check we can dramatically reduce the number of atomic operations in the later stages of the exploration.

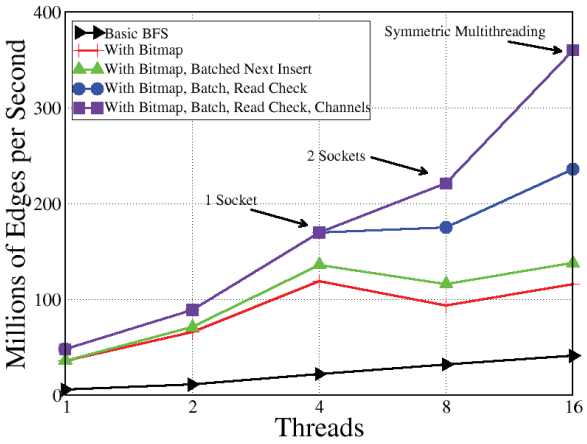


Fig. 5: Impact of various optimizations

and cache locking that will limit the amount of memory pipelining. In order to mitigate this problem we implemented a race-free, lightweight communication mechanism between groups of cores residing on different sockets. Our remote communication channels, those used for inter-socket communication, rely on two important algorithmic building blocks: an efficient locking mechanism, based on the *Ticket Lock* [22] that protects a single-producer/single consumer lock-free queue based on the *FastForward* algorithm [23]. In a nutshell, the remote channel is implemented as a FastForward queue where both producers and consumers are protected on their respective side by a Ticket Lock. FastForward, is a cache-optimized single-producer/single-consumer concurrent lock-free queue for pipeline parallelism on multicore architectures, with weak to strongly ordered consistency models. Enqueue and dequeue times are as low as 20 nanoseconds on the Nehalem architectures considered in this paper. One important property of the FastForward queues is that both sender and receiver can make independent progress without generating any unneeded coherence traffic. The activity and the coherency traffic of the FastForward queues can be almost entirely overlapped with the computation when the application is operating in throughput mode, which is the case of our BFS graph exploration. Algorithm 3 shows how these channels are implemented using socket queue SQ (lines 26, 28-35). While scanning the adjacency list of a node, we insert local vertices into the same socket next queue NQ by checking the *Bitmap* (lines 18-24), otherwise we use the socket queue SQ of the corresponding remote node (line 26). At the end of every stage, each socket reads its SQ and processes the nodes by checking their *Bitmap* and inserting the vertices into the socket next queue NQ (lines 28-35).

The final optimization is the use of batching when inserting and removing from the inter-socket communication channel, as shown in Algorithm 3 in lines 26 and 29. Rather than inserting at a granularity of a single vertex, each thread batches a set of vertices to amortize the locking overhead. Overall, including all the synchronization, locking and unlocking and buffer copies, the normalized cost per vertex insertion is only 30 nanoseconds. The inter-socket channels proved to be the key optimization, that allowed us to achieve very good scaling across multiple Nehalem EX sockets.

The overall impact of all optimizations is summarized in Figure 5 for the Nehalem EP. It is worth noting that the change of slope of the most optimized version of the algorithm between 4 and 8 threads is mostly due to the change of algorithm –we rely on a two-phase algorithm, the first one processing the local vertices and the second one processing the remote ones sent through the inter-socket communication channels, and not to the communication overhead, given that most operations are overlapped with carefully placed `__mm_prefecth()` instrinsics [24].

The proposed algorithm can be easily generalized to distributed memory machines that use fast and lightweight communication mechanisms, such as PGAS language and libraries [25].

IV. Experimental Results

We tested our algorithm on two different classes of graphs.

- Uniformly Random Graphs : Graphs with n vertices each with degree d, where the d neighbours of a vertex are chosen randomly.
- Scale-free graphs (R-MAT): Graphs generated using the GTgraph [26] suite based on the R-MAT graph model to represent real-world large-scale networks. Using a small number of parameters, R-MAT samples from a Kronecker product to produce scale-free graphs with community structure. These graphs have a few high degree vertices and many low-degree ones.

Table II provides the configuration details of the two Intel systems under consideration, a dual-socket Nehalem EP and a four-socket Nehalem EX, and several other parallel systems for BFS discussed in the literature.

	CPU Speed (GHz)	Sockets	Cores /Socket	Threads /Core	Threads	Cache Size /Socket	Cache Size	Memory
INTEL Xeon 7500 (Nehalem EX)	2.26	4	8	2	64	24M	96M	256G
INTEL Xeon X5570 (Nehalem EP)	2.93	2	4	2	16	8M	16M	48G
INTEL Xeon X5580 (Nehalem EP)	3.2	2	4	2	16	8M	16M	16G
CRAY XMT	500MHz	128	-	-	16K			1TB
CRAY MTA-2	220MHz	40	-	-	5120			160G
AMD Opteron 2350 (Barcelona)	2.0	2	4	1	8	2M	4M	16G

TABLE II

We measure the performance of the proposed BFS algorithm in edges processed per second. This is computed using $\frac{m_a}{running\ time}$, where m_a is the actual number of edges traversed during the BFS computation ($m_a < m$ for graphs that are not fully connected, where m is the total number of edges in the graph). In our measurements we noticed a maximum difference of 2% between m and m_a. The source vertex was chosen randomly in all the experiments given in this section.

Figures 6a & 6b plot the processing rate and speedup obtained on a uniformly random graph with 32 million vertices when the number of edges varies from 256 million to 1 billion, and number or threads varies from 1 to 16 on a Nehalem EP. The speedup is defined as the ratio of processing rate on t threads over 1 thread. We used the best performing algorithm for each thread configuration. When the threads run on the same socket, we disable inter-socket channels to get the highest performance. We use one thread per core up to 8 threads and use SMT to scale to 16 threads. We see that the performance of our algorithm not only scales well to multiple cores on the same socket but also across the two sockets of an EP system. Figure 6c plots the maximum processing rate obtained on a 2-socket Nehalem EP for uniformly random graphs when the number of edges varies from 256 million to 1 billion and the number of vertices varies from 1 million to 32 million. We observe in this plot that the processing rate only drops by a small factor when increasing the number of vertices. This is due to the higher random access latency with larger working set sizes. We obtain a processing rate between 200 to 800 million edges per second on a Nehalem EP with 2 sockets for both uniformly random and R-MAT graphs with 32 million vertices when the number of edges is varied from 256 million to 1 billion.

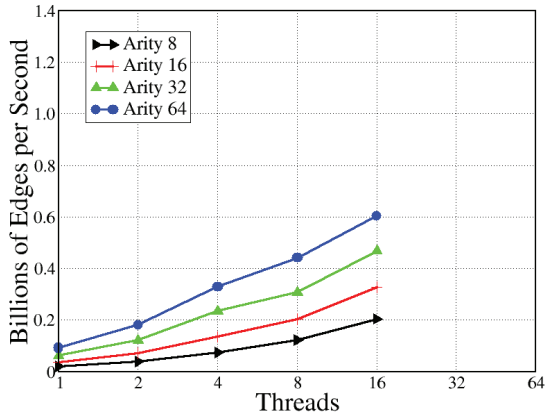
Similarly, Figure 7 plots the processing rate and speedup of R-MAT graphs on a Nehalem EP. We notice that the R-MAT graphs have higher processing rates than uniformly random graphs. This is because R-MAT graphs have a few high degree vertices that lead to a performance advantage more than the performance degradation caused by the low degree vertices in the graph. The slight reduction in the slope of the speedup curve from 4 to 8 threads is because of the change in the algorithm used when running on 1 socket vs 2 sockets (each socket of Nehalem EP contains 4 cores). The multi-socket algorithm performs extra insert/delete operations on inter-socket communication channels. We obtain a constant slope speedup curve, when the same channel-based algorithm is used on a single socket.

Figures 8 & 9 show the processing rate and speedup of uniformly random and R-MAT graphs on a 4 socket Nehalem EX. Our algorithm scales well to all 64 threads (32 cores) available on this blade, giving a speedup between 14-24 over the performance of a single thread. Again, as we noticed before, the slope of the speedup curve tails off from 8 to 16 threads, when the algorithm starts using inter-socket channels for task division and communication (each socket of Nehalem EX contains 8 cores). We also note in Figures 8c & 9c that the processing rate is not influenced by the number of vertices in the graph. This is due to a larger cache size on the Nehalem EX. We obtain a processing rate from 0.55 to 1.3 billion edges per second on 4 sockets of this processor for both uniformly random and R-MAT graphs with 32 million vertices when the number of edges varies from 256 million to 1 billion.

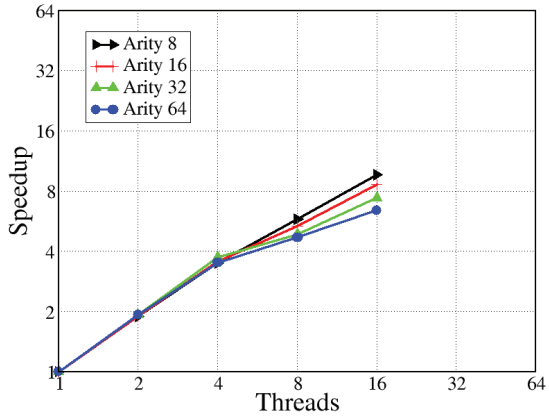
We provide a reference table (Table III) that summarizes several published results on parallel breadth first search. These results are categorized based on the graph size/types used, scalability with processors and with graph size, and choose some selected performance results that can be compared with the performance of our algorithm. The performance column in this table gives the performance in million edges per second (ME/s) for a graph with N vertices and M edges. When compared with these results, our BFS algorithm running on 4-socket Nehalem EX is:

- 2.4 times faster than a Cray XMT [15] with 128 processors running at 500 MHz with 1 TB shared memory, when exploring a uniformly random graph with 64 million vertices and 512 million edges;
- 5 times faster than 256 BlueGene/L [20] processors, on a graph with average degree 50. The system consists of 512 MB memory per compute node (each node with 2 PowerPC 440 processors running at 700 MHz);
- capable of processing 550 million edges per second with an R-MAT graph with 200 million vertices and 1 billion edges, comparable to the performance of a similar graph on a Cray MTA-2 [16] with 40 processors running at 220 MHz with 160 GB shared memory.

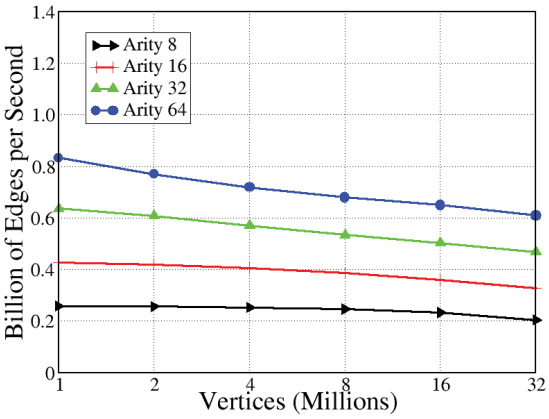
Please note that the Cray XMT and MTA-2 systems currently can handle graphs of much larger sizes due to the large



(a) Processing rates

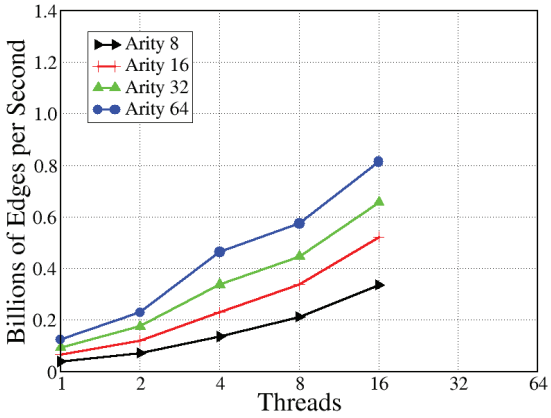


(b) Scalability

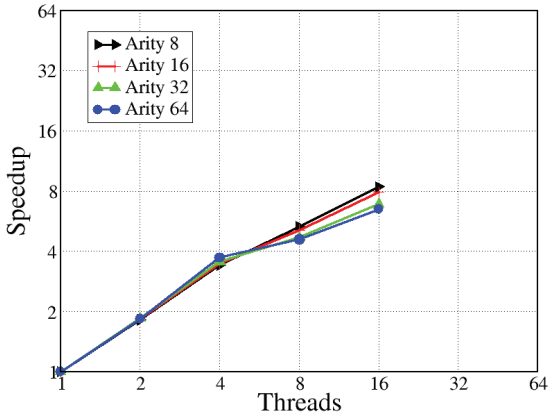


(c) Sensitivity to graph size

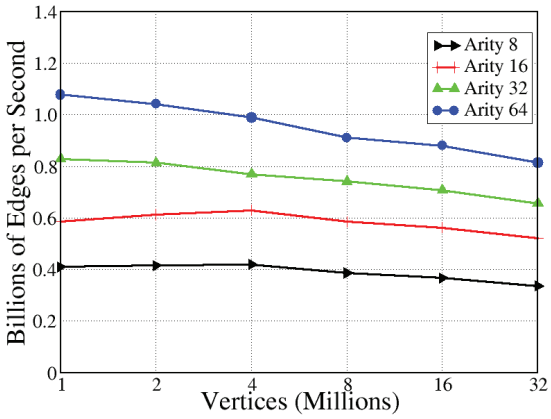
Fig. 6: Uniformly random graphs, Nehalem EP



(a) Processing rates



(b) Scalability



(c) Sensitivity to graph size

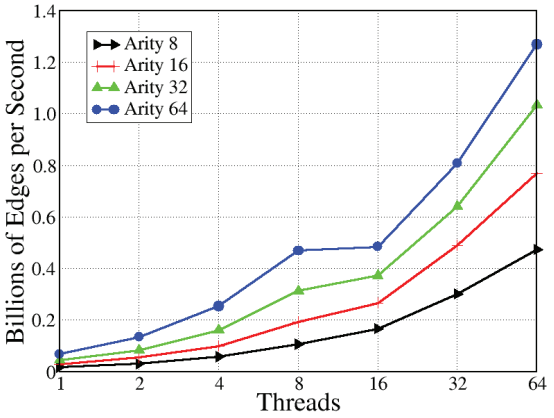
Fig. 7: RMAT graphs, Nehalem EP

system memory, whereas our algorithm can only handle the graphs that can fit inside the main memory of the Intel systems (256GB on 4-socket Nehalem EX that is expected to increase to 1TB and more on future systems).

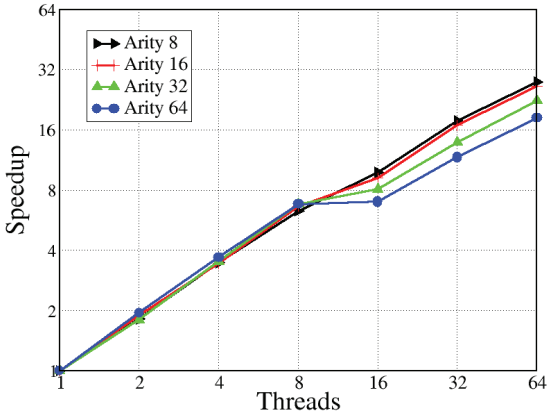
Figure 10 plots the throughput of our algorithm, where we run a single BFS per socket and run multiple instances of the algorithm on different graphs on different sockets. This is representative of the SSCA#2 benchmarks, and gives an estimate on the performance of our algorithm for such workloads.

V. Conclusion

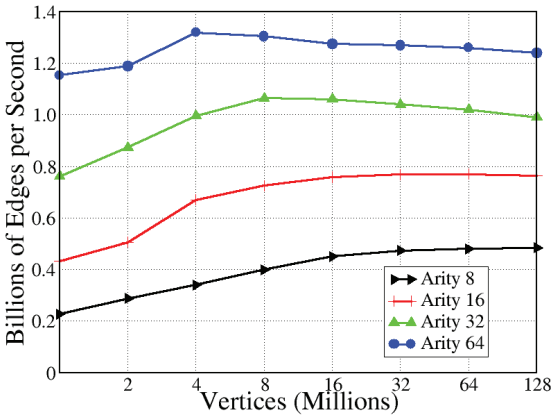
In this paper we have presented a scalable breadth-first search (BFS) algorithm for commodity multicore processors. In spite of the highly irregular access pattern of the BFS, our algorithm was able enforce various degrees of memory and processor locality, minimizing the negative effects of the cache-coherency protocol between processor sockets. The experimental results, conducted on two Nehalem platforms, a



(a) Processing rates

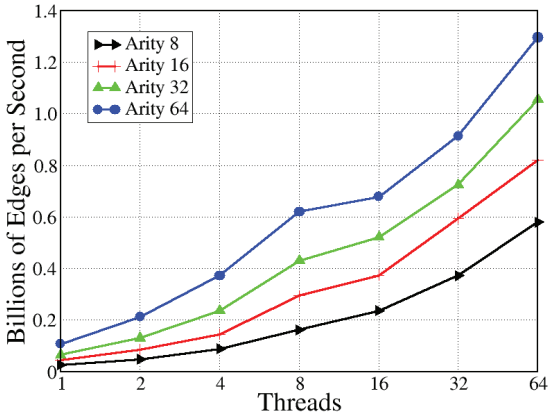


(b) Scalability

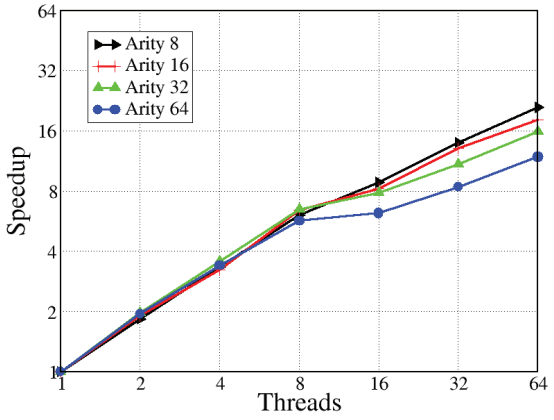


(c) Sensitivity to graph size

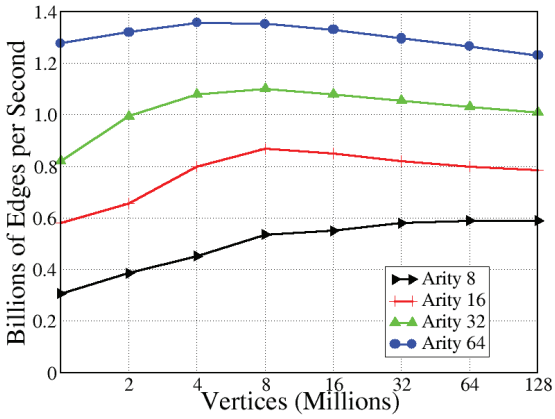
Fig. 8: Uniformly random graphs, Nehalem EX



(a) Processing rates



(b) Scalability



(c) Sensitivity to graph size

Fig. 9: RMAT graphs, Nehalem EX

dual-socket Nehalem EP and a four-socket Nehalem EX, have demonstrated an impressive processing rate in parsing graphs that have up to a billion edges. Using several graph configurations the Nehalem EX system has reached, and in many cases exceeded, the performance of special-purpose supercomputers designed to handle irregular applications. These are significant results in parallel computing as prior results in graph traversal report very limited or no speedup on irregular graphs when compared to their best sequential implementation. We believe that the results presented in this paper forms a valuable foundation to develop the architectural and algorithmic building blocks of upcoming exascale machines. We plan to extend the algorithmic design in the near future to map the graph exploration on distributed-memory machines that integrate powerful processing nodes, such as those considered in this paper, with high-performance, low-latency communication networks and lightweight PGAS programming languages.

Reference	Graph (size) type	Type	Performance			Base architecture	Scalability (cores)	Scalability (graph size)
			N (vertices)	M (edges)	ME/s	p processors		
Bader, Madduri [16]	(Large) Random, R-MAT and SSCA	R-MAT	200M	1B	500	40	Cray MTA-2	Good	Good
		SSCA2v1	32M	310M	250	10			
		SSCA2v1	4M	512M	250	10			
Mizell, Maschhoff [15]	(Large) Uniformly Random	Uniformly Random	64M	512M	210	128	Cray XMT	Good	Excellent
Scarpazza, Villa, Petrini [14]	(Medium) Uniformly Random	Uniformly Random	25	256M	101	1 chip	IBM Cell/B.E.	Good	Poor
		Uniformly Random	5M	256M	305	1 chip			
		Uniformly Random	2.5M	256M	420	1 chip			
		Uniformly Random	1M	256M	540	1 chip			
Yoo, Chow, Henderson, McLendon, Hendrickson, Catalyurek [20]	Large	-	Peak	$d10$	80	256	IBM BlueGene/L	Good	Good
		-	Peak	$d50$	232	256			
		-	Peak	$d100$	492	256			
		-	Peak	$d200$	731	256			
Xia, Prasanna [19]	(Small) Uniformly Random, Grids	8-Grid	1M	16M	220	Peak	dual Intel X5580	Moderate	Moderate
		16-Grid	1M	32M	311	Peak			

TABLE III

Fig. 10: SCCA#2 benchmark, throughput with uniform graphs, Nehalem EX

REFERENCES

[1] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "The potential of the Cell processor for scientific computing," in *Proc. 3rd Conf. on Computing Frontiers (CF'06)*, Ischia, Italy, 2006, pp. 9–20.

[2] L. Carrington, D. Komatitsch, M. Laurenzano, M. Tikir, D. Micheab, N. L. Goff, A. Snavely, and J. Tromp, "High Frequency Simulations of Global Seismic Wave Propogation Using SPECFEM3D GLOBE on 62K Processors," in *Proc. Intl. Conf. for High Performance Computing, Networking, Storage and Analysis (SC'08)*, Austin, TX, November 2008.

[3] D. A. Bader, G. Cong, and J. Feo, "On the Architectural Requirements for Efficient Execution of Graph Algorithms," in *Proc. Intl. Conf. on Parallel Processing (ICPP'05)*, Georg Sverdrups House, University of Oslo, Norway, June 2005.

[4] A. Clauset, M. E. J. Newman, and C. Moore, "Finding Community Structure in Very Large Networks," *Physical Review E*, vol. 6, no. 70, p. 066111, December 2004.

[5] J. Duch and A. Arenas, "Community Detection in Complex Networks Using extremal Optimization," *Physical Review E*, vol. 72, January 2005.

[6] M. E. J. Newman, "Detecting Community Structure in Networks," *European Physical Journal B*, vol. 38, pp. 321–330, May 2004.

[7] ——, "Fast Algorithm for Detecting Community Structure in Networks," *Physical Review E*, vol. 69, no. 6, p. 066133, June 2004.

[8] M. E. J. Newman and M. Girvan, "Finding and evaluating community structure in networks," *Physical Review E*, vol. 69, no. 2, p. 026113, February 2004.

[9] L. Zhang, Y. J. Kim, and D. Manocha, "A Simple Path Non-Existence Algorithm using C-Obstacle Query," in *Proc. Intl. Workshop on the Algorithmic Foundations of Robotics (WAFR'06)*, New York City, July 2006.

[10] A. Sud, E. Andersen, S. Curtis, M. C. Lin, and D. Manocha, "Real-time Path Planning for Virtual Agents in Dynamic Environments," in *IEEE Virtual Reality*, Charlotte, NC, March 2007.

[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 (SC'05)*, Seattle, WA, November 2005.

[12] 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.

[13] 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: IEEE Computer Society, 2006.

[14] D. P. Scarpazza, O. Villa, and F. Petrini, "Efficient Breadth-First Search on the Cell/BE Processor," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 10, pp. 1381–1395, 2008.

[15] D. Mizell and K. Maschhoff, "Early experiences with large-scale Cray XMT systems," in *Proc. 24th Intl. Symposium on Parallel & Distributed Processing (IPDPS'09)*, Rome, Italy, May 2009.

[16] D. A. Bader and K. Madduri, "Designing Multithreaded Algorithms for Breadth-First Search and st-connectivity on the Cray MTA-2," in *Proc. 35th Intl. Conf. on Parallel Processing (ICPP'06)*. Columbus, OH: IEEE Computer Society, August 2006, pp. 523–530.

[17] D. Bader, V. Agarwal, and K. Madduri, "On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study on List Ranking," in *Proc. Intl. Parallel and Distributed Processing Symposium (IPDPS'07)*, Long Beach, CA, March 2007.

[18] F. Blagojevic, A. Stamatakis, C. Antonopoulos, and D. Nikolopoulos, "RAxML-Cell: Parallel Phylogenetic Tree Inference on the Cell Broadband Engine," in *Intl. Parallel and Distributed Processing Symposium (IPDPS'07)*, Long Beach, CA, March 2007.

[19] Y. Xia and V. K. Prasanna, "Topologically Adaptive Parallel Breadth-first Search on Multicore Processors," in *Proc. 21st Intl. Conf. on Parallel and Distributed Computing and Systems (PDCS'09)*, Cambridge, MA, November 2009.

[20] 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 (SC'05)*. Seattle, WA: IEEE Computer Society, 2005.

[21] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller, "Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System," in *Proc. 18th Intl. Conf. on Parallel Architectures and Compilation Techniques (PACT'09)*, Raleigh, NC, September 2009, pp. 261–270.

[22] S. Sridharan, A. Rodrigues, and P. Kogge, "Evaluating Synchronization Techniques for Light-weight Multithreaded/Multicore Architectures," in *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*. New York, NY, USA: ACM, 2007, pp. 57–58.

[23] J. Giacomoni, T. Moseley, and M. Vachharajani, "FastForward for Efficient Pipeline Parallelism: A Cache-Optimized Concurrent Lock-Free Queue," in *Proc. 13th Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, Salt Lake City, UT, February 2008.

[24] A. Klimovitski, "Using SSE and SSE2: Misconceptions and Reality," *Developer UPDATE Magazine, Intel*, vol. 1, pp. 1–8, March 2001.

[25] W. W. Carlson, J. M. Draper, D. E. Culler, K. Y. E. Brooks, and K. Warren, "Introduction to UPC and Language Specification," IDA Center for Computing Sciences, Tech. Rep. CCS-TR-99-157, May 1999.

[26] D. Bader and K. Madduri, "GTgraph: A Synthetic Graph Generator Suite," 2006. [Online]. Available: http://sdm.lbl.gov/~kamesh/software/GTgraph