2012 IEEE 26th International Parallel and Distributed Processing Symposium

Fast and Efficient Graph Traversal Algorithm for CPUs : Maximizing Single-Node Efficiency

Jatin Chhugani, Nadathur Satish, Changkyu Kim, Jason Sewall, and Pradeep Dubey
Parallel Computing Lab, Intel Corporation

Abstract— **Graph-based structures are being increasingly used to model data and relations among data in a number of fields. Graph-based databases are becoming more popular as a means to better represent such data. Graph traversal is a key component in graph algorithms such as reachability and graph matching. Since the scale of data stored and queried in these databases is increasing, it is important to obtain high performing implementations of graph traversal that can efficiently utilize the processing power of modern processors.**

In this work, we present a scalable Breadth-First Search Traversal algorithm for modern multi-socket, multi-core CPUs. Our algorithm uses lock- and atomic-free operations on a cache-resident structure for arbitrary sized graphs to filter out expensive main memory accesses, and completely and efficiently utilizes all available bandwidth resources. We propose a work distribution approach for multi-socket platforms that ensures load-balancing while keeping cross-socket communication low. We provide a detailed analytical model that accurately projects the performance of our single- and multi-socket traversal algorithms to within 5-10% of obtained performance. Our analytical model serves as a useful tool to *analyze performance bottlenecks* on modern CPUs.

When measured on various synthetic and real-world graphs with a wide range of graph sizes, vertex degrees and graph diameters, our implementation on a dual-socket Intel® Xeon® X5570 (Intel microarchitecture code name Nehalem) system achieves 1.5X–13.2X performance speedup over the best reported numbers. We achieve around 1 Billion traversed edges per second on a scale-free R-MAT graph with 64M vertices and 2 Billion edges on a dual-socket Nehalem system. Our optimized algorithm is useful as a building block for efficient multi-node implementations and future exascale systems, thereby allowing them to ride the trend of increasing per-node compute and bandwidth resources.

I. Introduction

Graph traversal is an important and widely used algorithm that is useful in a variety of fields including data-intensive high performance computing [1], protein interactions, ground transportation [2], social networks [3], [4], [5] etc. Graph traversal has recently been proposed as a benchmark metric for measuring performance of supercomputers [6], [7], [8]. The Graph500 benchmark suite [9], which includes Breadth-First Search (BFS) as a core component, has been proposed for this purpose; and various high-performance computing platforms have already been ranked.

In order to correctly measure the performance of these platforms, it is important to optimize these benchmarks keeping in mind the underlying architecture. Compute and bandwidth resources in modern architectures have been increasing through the integration of more cores and simultaneous improvement in memory technology. Multi-socket architectures, with each socket having their own computation units and memory controllers, are popular in the High-Performance Computing (HPC) community.

Graph traversal poses several interesting challenges when mapped to modern CPU platforms. Graph traversal typically involves long-latency memory accesses followed by little arithmetic computation, leading to ineffective utilization of compute and bandwidth resources. This main memory access latency is difficult to hide due to the spatially incoherent nature of the accesses. Furthermore, the bandwidth requirements of such accesses are also high, leading to significant performance loss. Finally, multi-socket architectures require special handling since they need locality-aware optimizations to ensure that cross-socket traffic is kept low. However, this may conflict with the need to keep all sockets active for load-balancing. In such cases, a careful decision has to be made to balance both these factors for best performance.

In this paper, we concentrate on a representative Breadth-First Search graph traversal problem. There have been many asynchronous and synchronous algorithms proposed to optimize BFS on modern processors. Synchronous methods for BFS traverse the graph in a sequence of lock-steps bounded by synchronization points, with each step performing updates of all vertices at a single depth. In contrast, asynchronous approaches eliminate the use of synchronization points, although this may result in multiple updates for a single vertex and consequent work inefficiency. Synchronous approaches work well for many graphs whose diameters are small compared to the number of vertices—these include many real-world graphs. We hence concentrate on synchronous approaches.

In this paper, we present an efficient synchronous scheme for BFS graph traversal on modern multi-socket CPUs. We make the following contributions:

- Lock-free and atomic-free, parallel, load-balanced BFS traversal to reduce the impact of memory latency.
- Rearrangement of the frontier of vertices and prefetching to hide latency of irregular graph accesses and fully exploit available memory bandwidth on multi-socket CPUs.
- Cache-friendly algorithm with efficient use of memory bandwidth even for very large graphs.
- Multi-socket optimizations to balance dual requirements

1530-2075/12 $26.00 © 2012 IEEE
DOI 10.1109/IPDPS.2012.43

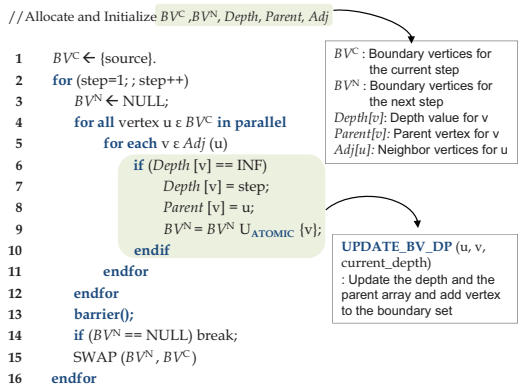

Figure 1. **Code snippet for a baseline unoptimized BFS traversal algorithm.**

of load-balancing while keeping cross-socket communication low.

- Builds an accurate performance model of BFS traversal on multi-socket, multi-core CPUs which works for varied types of graphs—both synthetic and real-world, with a wide range of graph sizes, vertex degrees and graph diameters. This is the first such paper we know of.
- Achieves 1.5–13.2X speedup over previously reported numbers on the same platform; achieves similar performance to a 256-node cluster machine ranked in the Nov. 2010 Graph500 list.

Finally, several multi-node implementations of graph traversal algorithms have been recently proposed that can handle very large graphs that do not fit in a single node's memory [7], [8], [9], [10]. Such implementations have been shown to scale across large numbers of nodes for large scale graphs [8], [10], [11]. However, the costs of powering up these clusters becomes increasingly expensive (up to 50% of the total cost of ownweship) as the clusters grow larger [12], [13], [14]. Thus, for cost reduction purposes, it is increasingly important to have an **efficient single-node** implementation that *minimizes* the number of nodes required (up to memory limits). As an example, our single-node BFS implementation on a common dual-socket Intel® Xeon® X5570 Nehalem system matches that of a 256-node system with a similar single-node configuration ranked in the November 2010 Graph 500 list [9], with significantly lower operating costs. An efficient single-node implementation will also allow multi-node implementations to ride the trend of increasing per-node compute and bandwidth resources in accordance with future technology scaling trends.

II. Motivation

Multi-socket architectures are increasingly being used for high-performance computing. Each socket has its own compute and memory controller, and thus both compute and bandwidth resources increase in proportion to the number of sockets. It is possible for cores in any socket to access data present in DRAM in a different (remote) socket. However, all cross-socket traffic goes through a cross-socket link which has lower bandwidth than accesses to local DRAM and caches. Hence, efficient use of multi-socket architectures requires careful optimizations to ensure that locality of data is preserved. However, there is a trade-off: locality-awareness may conflict with the need to keep all sockets active for load-balancing. In such cases, a careful decision has to be made to balance both these factors for best performance. In this section, we motivate the need for an architecture-aware algorithm choice for BFS.

Latency hiding: Graph traversal algorithms frequently do not fully utilize either compute or bandwidth resources on modern architectures since they are limited by memory latency. There have been many previous attempts to optimize synchronous graph algorithms on CPUs—however, none of them specifically attempt to fully hide latency effects.

Figure 1 shows a code snippet for BFS graph traversal. The code goes through a loop that iterates over the **steps** with a synchronization point between each step. Each step iterates over the set of vertices in the boundary set $\mathcal{BV}^{\mathcal{C}}$, looks up the $\mathcal{Adj}$ data structure (that stores the neighbors) for each vertex, checks each neighbor to see if has been visited and if not, updates the depth and parent arrays. In the figure, accesses to both the $\mathcal{Adj}$ and the *Depth* and *Parent* data structures are **spatially incoherent** and spread across the whole range of vertices—hence each access involves cache and Translation Lookaside Buffer (TLB) misses. Furthermore, previous work has relied on the use of atomic operations [1] to avoid race conditions. This leads to a further increase in latency since they behave as memory fences that lead to serialization [15]. In this work, we counteract the latency impact by adopting a **atomic-free update mechanism**, and **rearrange the** $\mathcal{BV}^{\mathcal{C}}$ **array** to reduce TLB misses when accessing $\mathcal{Adj}$ array. Our resulting algorithm is **no longer limited by memory latency**.

Efficient use of bandwidth: The next step is to reduce the bandwidth consumption of the algorithm as much as possible. In Figure 1, the $\mathcal{Adj}$ access is performed once per vertex, and the cache-line is used for accessing all neighbors of that vertex—this is not a bottleneck for medium-degree to large-degree graphs. However, the *Depth* and *Parent* arrays involves spatially incoherent access for each edge, and can bring in as much as a cache-line (64 bytes) of data each per edge. It is, therefore, critical to avoid per-edge accesses to these arrays. A common approach in previous work has been to keep auxiliary data structures that encode if a vertex has already been visited. This structure contains less information than the entire *Depth* and *Parent*, and can potentially be kept in cache. Previous works [16], [17], [18] have proposed various schemes for maintaining these arrays, but none of

[1] We use "atomic operations" to represent any x86 instructions with the LOCK prefix.

them work for large graphs where even one bit of data per vertex may not fit in cache. In this work, we present a **generalized, cache-resident, atomic-free** representation of this auxiliary bit-array, termed *VIS* for the rest of the paper. Some of the previous schemes [19], [16] perform a static partitioning of vertices between threads to avoid locks and atomic operations. However, this leads to increased load-imbalance.

Multi-socket optimizations: Efficient implementations on multi-socket architectures that are widely used in the HPC community require a trade-off between load-balancing the execution on sockets and keeping cross-socket traffic low. In particular, the cache-lines in the *VIS* data structure can be accessed and updated by all sockets (since the neighbors of vertices can be spatially incoherent), resulting in ping-ponging of the cache-lines and significant cross-socket traffic. Since cross-socket bandwidth can be a major constraint [18], it is important for *VIS* accesses to be locality-aware. We propose a scheme where we partition the vertices among the sockets. This requires a **2-phase** approach, with the first phase writing out neighbors of vertices in the $\mathcal{BV}^{\mathcal{C}}$ lists into intermediate arrays called **Potential Boundary Vertices** ($\mathcal{PBV}$), while the second phase reads the $\mathcal{PBV}$ arrays and updates the $\mathcal{DP}$ data structure. However, a static partitioning scheme can result in load-imbalance—and moreover, this cannot be predicted since it depends on the graph structure. Hence, we need to **dynamically load-balance** the $\mathcal{PBV}$ arrays between sockets while ensuring that the cross-socket traffic is kept low. To the best of our knowledge, this is the first graph traversal algorithm that proposes a balance between load-balancing and locality-aware computation.

Analytical Model: Once we eliminate latency impacts, the performance of our algorithm becomes predictable. We **develop an analytical model** in Section IV **that closely predicts** the performance of our single-socket and multi-socket algorithms. In Section V-C, we show that our implementation matches to within 5-10% of the model predictions, therefore showing that we achieve good efficiency.

III. Algorithm

We first describe our novel algorithm for maintaining and updating the *VIS* array on a multi-core/socket architecture. We then present the justification for various design choices in our algorithm, followed by our load-balanced locality-aware cache-friendly BFS traversal algorithm.

A. Cache-Resident Atomic-Free VIS Structure

In order to reduce the external memory traffic, previous implementations [18] advocate using a small sized auxiliary structure (a bit-vector) to check whether a vertex has already been assigned a depth, and hence reduce this external memory traffic. However, their algorithm suffers from two drawbacks:

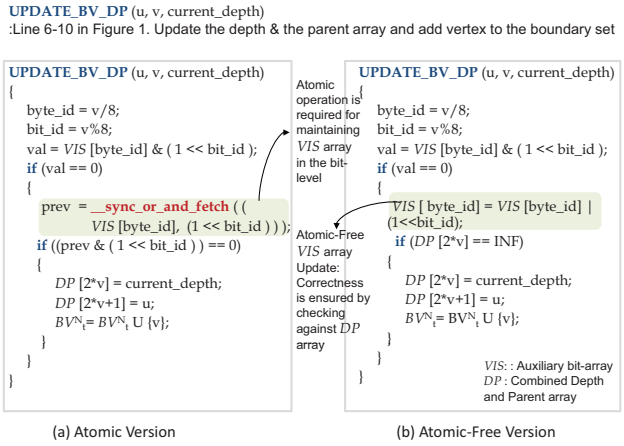

Figure 2. **Code-snippets showing accesses and updates to *VIS* array (a) Using atomic operations. (b) Using our atomic-free scheme.**

(1) The bit-vector is assumed to fit in the LLC caches **during the execution of a step**, which may not hold with increasing number of vertices in the graphs. Once the size of the bit-vector is larger than the LLC size, this scheme degenerates to fetching cache-lines from main memory, and reduced performance. (2) Use of atomic operations to avoid race conditions. This leads to increased latency in computation. In contrast, we present a generalized cache-resident atomic-free way of representing this auxiliary bit-array, termed *VIS*.

Maintaining a cache-resident *VIS* array: For a graph with $|\mathcal{V}|$ vertices, the total memory required to store a bit-structure ($|VIS|$) is $|\mathcal{V}|/8$ bytes. Let $|C|$ represent the LLC size. As an example, say $|\mathcal{V}|$ = 256M, and $|C|$ = 16 MB. Hence $|VIS|$ = 32MB, which is 2X larger than the LLC size [2]. Hence, in order for the *VIS* array to fit in LLC, we need to **partition it** into multiple sub-parts. Since multiple data structures are accessed during the execution of a step, it is advisable to further increase the number of partitions by 2X (hence the subdivided *VIS* array is about half the LLC size) to ensure that *VIS* is not evicted from LLC. Thus we create at least $\lceil\ |\mathcal{V}|\ /\ (4|C|)\ \rceil$ partitions (=256/64 = 4 in this case). This ensures that accesses to *VIS* only need to be re-loaded once at each step of the algorithm, and are cache-resident within each step. We refer the number of partitions required for cache-friendly *VIS* access as $\mathcal{N}_{VIS}$.

Atomic-Free and Lock-Free Updates to *VIS* array: We *do not use any atomic operations* for updating and accessing *VIS*. Since the underlying architecture guarantees atomic reads/writes starting at a *byte (8-bits) granularity* [15], the following two scenarios need further analysis: (1) Multiple threads want to simultaneously update the same bit. (2) Multiple threads want to simultaneously update different bits that fall within the same 8-bit granularity. In the former case, the

[2]In case $|\mathcal{V}|\leq|C|$, one may use a *byte*-structure per vertex for *VIS*.

corresponding bit in *VIS* will eventually be set to 1, and all the threads would update the *depth* of the corresponding vertex. Since all the threads are executing the **same step of the traversal**, they would end up assigning the same *depth* (with potentially different parents) to the vertex, and still get a valid and correct BFS tree. In the latter case, it is possible that the bit corresponding to some of the vertices being accessed may not be set to 1, while the *depth* for those vertices would have been updated. Therefore, to ensure correctness, in case the access of *VIS* for a certain vertex returns a value of 0, we set it to 1, but update the *depth* (and *parent*) and append that vertex to $\mathcal{BV}_t^N$ **only if** the stored *depth* has not been updated so far. Using 8/16/32/64-bits to represent the *depth* and *parent* values ensures that the updates to $\mathcal{DP}$ are always consistent [15]. Although our scheme can lead to more accesses to the $\mathcal{DP}$ array, in practice we noticed an increase of up to 0.2% for small graphs, and this percentage decreases with increase in size of graphs. Hence, a bit value of 0 in our *VIS* array implies that the depth of the corresponding vertex may possibly have been updated, while bit value of 1 implies that the depth of the corresponding vertex has definitely been updated. It is not possible for a bit in *VIS* to be set to 1 when the depth of the corresponding vertex has not been updated at the end of the step. Figure 2 shows the two corresponding code snippets—the one on the left uses atomic operations, while the one on the right is our atomic-free scheme.

We present the first lock-free and cache-friendly algorithm for managing the *VIS* array. We always obtain the **correct depth for all vertices, and a valid BFS tree**.

B. Algorithm Sketch

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ denote the input graph, and v_0 ($\in \mathcal{V}$) denote the starting vertex (at *depth* 0) to the BFS traversal algorithm. Our algorithm stores the *depth* and *parent* of each vertex together in an array, denoted by $\mathcal{DP}$–initialized to INF (∞), which is returned at the end of the algorithm. Refer to Fig. 1 for definitions of $\mathcal{BV}^C$ and $\mathcal{BV}^N$. From the perspective of load-balanced locality-aware parallelization on a multi-socket CPU, it is important to execute the following *three* tasks efficiently, both in terms of *memory accesses*, and *amount of work done* per executing thread: (1) *Maintaining* $\mathcal{BV}^C$ and $\mathcal{BV}^N$. (2) *Accessing neighbors* of vertices in $\mathcal{BV}^C$. (3) *Locality-Aware* allocation and scheduling of work. We now describe each of these in detail.

1) Maintaining $\mathcal{BV}^C$ and $\mathcal{BV}^N$: Consider $\mathcal{BV}^N$. In order to reduce cross-socket communication while writing, it is important for threads executing on a specific socket to write boundary vertices to memory allocated on the same socket. Furthermore, in order to reduce cross-thread communication, we maintain a per-thread array of $\mathcal{BV}^N$, denoted by $\mathcal{BV}_t^N$. Since $\mathcal{BV}^N$ and $\mathcal{BV}^C$ are swapped at the end of each step, a per-thread array of $\mathcal{BV}^C$, denoted by $\mathcal{BV}_t^C$, is also maintained.

2) Accessing neighbors of vertices in $\mathcal{BV}_t^C$: The input graph is represented as a 2D Adjacency Array (denoted by $\mathcal{A}dj$), with $\mathcal{A}dj[i][j]$ storing the j^{th} neighbor of the i^{th} vertex, and $\mathcal{A}dj[i][0]$ storing the total number of neighbors of i^{th} vertex. For a multi-socket CPU, we **evenly divide** the $\mathcal{A}dj$ array amongst the available sockets (to take advantage of the multiple socket bandwidth). Let $\mathcal{N}_S$ denote the number of available sockets, and $|\mathcal{V}_{NS}|$ denote the number of vertices assigned to each socket. Specifically, we store the $\mathcal{A}dj$ array for the *first* $|\mathcal{V}_{NS}|$ vertices (including the addresses $\mathcal{A}dj[0]$ .. $\mathcal{A}dj[|\mathcal{V}_{NS}|\text{-}1]$) on the *first* socket, the next $|\mathcal{V}_{NS}|$ vertices on the *second* socket and so on.

3) Locality-Aware Allocation & Work Scheduling: Consider the access of *VIS* during the execution of a step of the BFS traversal. These accesses are to the neighbors of the vertices in $\mathcal{BV}_t^C$. Since the neighbors of vertices can be spatially incoherent and spread across the whole range of vertices, it is possible for multiple sockets to access the same (or close by) neighbors, thereby resulting in cross-socket cache-line updates and ping-ponging while accessing *VIS*. This can lead to severe loss of performance.

One way to reduce the inter-socket communication of *VIS* is to partition these neighboring vertices into various bins, such that each bin is only accessed by one socket, thereby eliminating any such communication. This requires dividing each step into **2 phases**, where the **first phase (Phase-I)** accesses the neighbors of vertices in $\mathcal{BV}_t^C$, and **partitions (or bins)** them into intermediate arrays based on the vertex id. We denote these intermediate arrays as **Potential Boundary Vertices** ($\mathcal{PBV}$). Let $\mathcal{N}_{PBV}$ denote the total number of bins created. Each thread created $\mathcal{N}_{PBV}$ bins. Similar to the allocation of $\mathcal{BV}$, each thread allocates these arrays in its local memory ($\mathcal{PBV}_t$) [3]. At the end of Phase-I, the bins are divided amongst the sockets, and each socket operates on the assigned bin(s). The work performed in **Phase-II** involves iterating over the vertices in the bin, and using our scheme for atomic-free updates to *VIS* and updating $\mathcal{DP}$ for the unassigned vertices (Figure 2(b)). At the end of Phase-II, each thread produces the list of boundary vertices for the next step ($\mathcal{BV}_t^N$). In order to take advantage of the multiple-socket bandwidth, we evenly divide the allocation of $\mathcal{DP}$ and *VIS* amongst the socket memories.

We now compute the total number of bins required (i.e. $\mathcal{N}_{PBV}$). Recall from Section III-A that we require at least $\mathcal{N}_{VIS}$ bins. Also recall that $\mathcal{A}dj$ is evenly divided amongst sockets. Hence, in order to efficiently utilize the external memory bandwidth, we require at least $\mathcal{N}_S$ bins (equal to number of sockets) so that each socket can access the corresponding $\mathcal{A}dj$ memory locations and extract maximum external memory bandwidth. Hence, we advocate using a total of ($\mathcal{N}_{VIS} \times \mathcal{N}_S$) bins. We now describe our division of work in the two phases between the various sockets and executing threads.

(a) Load-Balanced Locality-Aware Division of Work: Consider the scenario at the beginning of Phase-II, where each thread has partitioned the set of neighboring vertices into

[3]This is achieved through the numa_alloc_onnode API of libnuma.

$\mathcal{N}_{PBV}$ bins. One way to divide the work is to assign $\mathcal{N}_{VIS}$ bins to each socket. Note that this not only eliminates the cross-socket communication of *VIS*, but also ensures that each socket would access and update the regions of $\mathcal{DP}$ and *VIS* that were allocated on their local memories. Hence, this completely eliminates any cross-socket traffic (except the transfer of relevant bin vertices between sockets). Furthermore, the vertices enqueued in $\mathcal{BV}_t^N$ by threads running on this socket would also map to the same socket. The same work distribution scheme can be applied in Phase-I, wherein the threads evenly divide the vertices in $\mathcal{BV}_t^C$ enqueued by threads executing on their socket. However, for each *step*, it is indeed possible for different sockets to be assigned different number of vertices in Phase-II, thereby leading to **load-imbalance and reduced scaling**. Similar **load-imbalance** may exist in Phase-I if the number of vertices assigned to each socket are different.

Hence, we propose a modified scheme that **balances the work between sockets**, while keeping the communication low. At the beginning of Phase-II, each thread computes the total number of vertices in each bin (enqueued by all the threads). We then compute the sum of these vertices, and evenly divide them amongst the sockets, with the *first socket* assigned the first $1/\mathcal{N}_S$ vertices, and so on. This leads to a coherent division of work, wherein **each socket is assigned a few complete bins, and at most two partial bins**. Thus, *each socket would access a few complete bins, and share at most two bins with other socket(s)*. This reduces the amount of cross-socket communication, except for the bins being shared between sockets. Also, due to the orderly division of bins, the vertices enqueued by sockets at the end of Phase-II follow a similar pattern, with all vertices mapping to a specific bin enqueued together (by each of the threads in its local $\mathcal{BV}_t^N$).

Hence, we apply this load-balancing scheme to Phase-I too, where the vertices (of all the $\mathcal{BV}_t^C$) in each bin are computed, and work divided as explained above. Although our scheme increases the amount of cross-traffic between sockets (as compared to a non load-balanced scheme), it now completely utilizes the (much larger) internal cache bandwidth on each socket to achieve improved scaling and parallel performance. Section IV provides an analytical formulation of the improved performance, while the results comparing the two schemes are presented in Section V-A. Note that for both **Phase-I and Phase-II**, each socket evenly divides the vertices in *each assigned bin* between the various threads executing on the socket to completely utilize the internal cache bandwidths.

(b) Reducing TLB Misses: With increasing graph sizes and vertex degrees, the size of $\mathcal{A}dj$ can grow as large as hundreds of GBs. Spatially incoherent accesses to $\mathcal{A}dj$ will incur TLB misses and be heavily latency bound. A common way to reduce such misses is by performing multiple passes over the list of boundary vertices, and processing vertices that belong to a specific range of memory. However, this may reload the $\mathcal{BV}_t^N$ array multiple times, and increase bandwidth

```
// Compute |V_NS| and N_PBV.
// Allocate Adj, DP, VIS, BV_t^C , BV_t^N , PBV_t, and Initialize
for (step=1; ; step++)
        Reset BV_t^N
        Divide_BV_t^C_among_threads( BV_t^C )
        for all vertex u ε assigned BV_t^C
Phase-I     Perform_Prefetches for Adj   /* Prefetch neighbor list */
            Perform_SIMD_Binning_PBV_t( u, Adj, PBV_t ); /* Partition the
                                                    neighbor among
                                                    the N_PBV bins */
        endfor
        barrier();

        Divide_PBV_t_among_threads ( PBV_t )
        for all vertex v ε assigned PBVt
            u = Access_Parent ( v, PBV_t );   /*Parent is stored before the
Phase-II    UPDATE_BV_DP ( u, v, step );      neighbor list*/
                                              /* Same as in Figure 2 (b)*/
        endfor
        Rearrange (BV_t^N)
        barrier();

        new_num_boundary_verticies = Sum (BV_t^N)
        if (new_num_boundary_verticies == 0) break;
        SWAP(BV_t^N , BV_t^C);
endfor
```

Figure 3. **Our load-balanced locality-aware BFS Traversal Algorithm.**

requirements. In contrast, we perform an efficient one-pass low-overhead reordering of the vertices in $\mathcal{BV}_t^N$ (performed by each thread) at the end of second phase of each step.

Our rearrangement algorithm is similar to the partitioning scheme proposed by Kim et al. [20]. To rearrange the vertices, we compute a histogram of values representing the number of accesses to different locations of the memory, and then scatter the vertices based on the obtained histogram into a temporary array, and finally copy back the vertices from the temporary array to $\mathcal{BV}_t^N$. The number of histogram bins is set to the total number of pages occupied by the $\mathcal{A}dj$ array divided by the number of simultaneous pages that can be held in the underlying hardware's TLB.

C. Complete Algorithm/Implementation

Based on the discussion in the previous section, the complete BFS traversal algorithm is delineated in Figure 3. We now describe some of the functions and optimizations mentioned in the algorithm.

(1) Computing $|\mathcal{V}_{NS}|$ and $\mathcal{N}_{PBV}$: To reduce the overhead of computation, $|\mathcal{V}_{NS}|$ (the number of vertices assigned to each socket while allocating $\mathcal{A}dj$, $\mathcal{DP}$ and *VIS*) is rounded to be the nearest power of two $\geq (|\mathcal{V}|/\mathcal{N}_S)$. Hence $|\mathcal{V}_{NS}| = pow(2, \lceil \log(|\mathcal{V}|/\mathcal{N}_S) \rceil)$. Hence, given a vertex id v_2 ($\in \mathcal{V}$), the socket id on which the corresponding data (for the three data structures) resides is : Socket_Id(v_2) $\leftarrow$ $rshift(v_2, \log_2(|\mathcal{V}_{NS}|))$. $\mathcal{N}_{PBV} = (\mathcal{N}_S)(\mathcal{N}_{VIS}) = (\mathcal{N}_S \lceil\ |\mathcal{V}|\ /\ (4|C|)\ \rceil)$ (Sec. III-A and III-B3(a)).

(2) Divide_$\mathcal{BV}_t^C$_among_threads($\mathcal{BV}_t^C$): We use our load-balanced scheme for dividing $\mathcal{BV}_t^C$ described in Sec. III-B3.

(3) Perform_Prefetches_for_$\mathcal{A}dj$: During the execution of Phase-I, each thread is assigned a list of vertices for which $\mathcal{A}dj$ needs to be accessed. However, since successive vertexes may point to spatially incoherent regions of memory, this

access pattern cannot be captured by the hardware pre-fetcher. Hence, while accessing $\mathcal{A}dj$ for the k^{th} vertex in $\mathcal{BV}_t^C$, we issue *_mm_prefetch* instructions to access the address ($\mathcal{A}dj$ + $\mathcal{BV}_t^C[k + PREF\_DIST]$) and the list of neighbors ($\mathcal{A}dj[\mathcal{BV}_t^C[k + PREF\_DIST]]$) into the L1 cache.

(4) Perform_SIMD_Binning_$\mathcal{PBV}_t$: For each assigned vertex in $\mathcal{BV}_t^C$, the thread first enqueues that vertex in each of the $\mathcal{PBV}_t$ bins, and **marks them** (by negating the id), so that it can be used (as the parent) for the next phase if the depth of any of its neighbors is updated [4] . This is followed by computing the bin index for each neighbor, and enqueuing it to the corresponding bin. To reduce the instruction overhead, we compute the bin index of 4 simultaneous neighbors together using SSE instructions. In addition, we use packed store operations (using SSE shuffle instructions) to partition the neighbors amongst the available $\mathcal{N}_{PBV}$ bins. We achieve an overall instruction reduction of 1.3–2X.

(5) Divide_$\mathcal{PBV}_t$_among_threads ($\mathcal{PBV}_t$): We use our load-balanced scheme for dividing $\mathcal{PBV}_t$ (Sec. III-B3).

(6) Access_Parent(v, $\mathcal{PBV}_t$): While traversing $\mathcal{PBV}_t$, if we encounter a negative vertex, we negate it and store it as a parent vertex. For a non-parent vertex, the latest parent vertex is used as its parent vertex. Correctness follows from our protocol for storing parent vertices in $\mathcal{PBV}_t$ as described in (4) above.

(7) Rearrange($\mathcal{BV}_t^N$): We use our reordering scheme (Sec. III-B3(b)) to rearrange boundary vertices in $\mathcal{BV}_t^N$ to reduce TLB misses.

IV. Performance Model

We now present an *analytical model*, aimed at computing the total memory access (in *bytes*) and resultant run-time (in *cycles*) required by our BFS traversal algorithm. The model serves the following purposes: (1) Computing the efficiency of our implementation. (2) Projecting performance for graphs with different topologies. (3) Analyzing performance bottlenecks. In this section, we present the resultant memory traffic and performance for single- and multiple-sockets, while the details of the derivation are presented in Appendix A.

Notation:

$|\mathcal{V}'|$: Number of Vertices assigned a depth during the BFS traversal.
$|\mathcal{E}'|$: Number of Traversed Edges during the BFS traversal.
ρ': Average degree of the vertices assigned a depth (=$\mathcal{E}'/\mathcal{V}'$).
$\mathcal{L}$: Cache-Line Size (in Bytes).
$\mathcal{B}_{QPI}$: Achievable B/W from one socket to another (in GB/sec).
$\mathcal{B}_M$: Achievable Main Memory B/W (in GB/sec).
$\mathcal{B}_{Mmax}$: Maximum B/W from Main Memory to LLC (in GB/sec).
$\mathcal{B}_{L2\rightarrow LLC}$: Achievable Write B/W from L2 to LLC (in GB/sec).
$\mathcal{B}_{LLC\rightarrow L2}$: Achievable Read B/W from LLC to L2 (in GB/sec).
$\mathcal{F}req$: Core Frequency (in GHz).
$\mathcal{D}$: Depth of the Graph.
$|\mathcal{L}2|$: Size of private L2 (per core) (in Bytes).
$|VIS|$: Size of *VIS* Array (in Bytes). Equals $|\mathcal{V}|/8$ Bytes.

Furthermore, X denotes Phase-I, Phase-II or Rearrangement phase in the notation below:

DT_M^X : Data transferred between main memory and LLC (in *bytes per traversed edge*).
DT_{LLC}^X : Data transferred between LLC and caches (in *bytes per traversed edge*).
$ET_{\mathcal{N}_\mathcal{S}}$: Execution time on $\mathcal{N}_\mathcal{S}$ sockets (in *cycles per traversed edge*).
$\mathcal{B}'_Y$: Effective bandwidth for accessing the data structure Y (like *VIS*, $\mathcal{A}dj$, etc.) in a multi-socket setting.
$\alpha_{\mathcal{A}dj}$: Max. fraction of accesses to $\mathcal{A}dj$ from any socket's memory.
$\alpha_{\mathcal{BV}_\mathcal{C}}$: Max. frac. of accesses to $\mathcal{BV}_t^C$, $\mathcal{BV}_t^N$ from any socket's mem.
$\alpha_{\mathcal{PBV}_t}$: Max. frac. of accesses to $\mathcal{PBV}_t$ from any socket's mem.
$\alpha_{\mathcal{DP}}$: Max. frac. of accesses to $\mathcal{DP}$ from any socket's mem.

The total data transferred in each of the phases equals:

$$DT_M^{Phase-I} = 12 + \frac{4+2\mathcal{L}+8\mathcal{N}_{PBV}}{\rho'} \tag{IV.1a}$$

$$DT_M^{Phase-II} = 4 + \frac{8+2\mathcal{L}+4\mathcal{N}_{PBV}+\frac{|\mathcal{V}|}{|\mathcal{V}'|}\frac{\mathcal{D}}{8}}{\rho'} \tag{IV.1b}$$

$$DT_{LLC}^{Phase-II} = \left(1-\frac{L2}{\frac{|VIS|}{\mathcal{N}_{VIS}}}\right)\left(\frac{\mathcal{L}}{\rho'}+\mathcal{L}\right) \tag{IV.1c}$$

$$DT_M^{Rearrange} = \left(\frac{24}{\rho'}\right) \tag{IV.1d}$$

To compute the execution times, we need to compute the effective bandwidth for accessing the data structures, and then divide the corresponding number of bytes transferred by the effective bandwidth to obtain the run-time. For a single-socket execution, the corresponding run-time ET_1 equals:

$$\begin{aligned} ET_1 = &\left(\frac{\mathcal{F}req}{\mathcal{B}_\mathcal{M}}\right)\left(12+\frac{4+2\mathcal{L}+8\mathcal{N}_{PBV}}{\rho'}+\frac{24}{\rho'}\right)+ \\ &\left(\frac{\mathcal{F}req}{\mathcal{B}_\mathcal{M}}\right)\left(4+\frac{8+2\mathcal{L}+4\mathcal{N}_{PBV}+\frac{|\mathcal{V}|}{|\mathcal{V}'|}\frac{\mathcal{D}}{8}}{\rho'}\right)+ \\ &\left(1-\frac{L2}{\frac{|VIS|}{\mathcal{N}_{VIS}}}\right)\left(\left(\frac{\mathcal{F}req}{\mathcal{B}_{L2\rightarrow LLC}}\frac{\mathcal{L}}{\rho'}\right)+\left(\frac{\mathcal{F}req}{\mathcal{B}_{LLC\rightarrow L2}}\mathcal{L}\right)\right) \end{aligned} \tag{IV.2}$$

To compute the run-times for multiple sockets, we present below the effective bandwidth for accessing $\mathcal{A}dj$ (denoted by $\mathcal{B}'_{Adj}$) on $\mathcal{N}_\mathcal{S}$ sockets. A similar expression can be derived for for $\mathcal{BV}_t^C$, $\mathcal{BV}_t^N$, $\mathcal{PBV}_t$ and $\mathcal{DP}$.

$$\mathcal{B}'_{Adj} = \left(\frac{1}{\mathcal{N}_\mathcal{S}\mathcal{B}_{LLC\rightarrow L2}}+\frac{\alpha'_{Adj}}{\min(\mathcal{B}_{QPI},\frac{\alpha'_{Adj}\mathcal{B}_{Mmax}}{\frac{1}{\mathcal{N}_\mathcal{S}}+\alpha'_{Adj}})}\right)^{-1},$$

$$\text{where } \alpha'_{Adj} = \frac{\alpha_{Adj}-\frac{1}{\mathcal{N}_\mathcal{S}}}{\mathcal{N}_\mathcal{S}-1} \tag{IV.3}$$

[4] For $\mathcal{N}_{PBV} \geq \rho$, storing (parent, vertex) pairs is more space efficient. We switch between the two representations based on the actual graph parameters.

Platform Characteristic	Performance
GFlops	2 × 94
Achievable DDR BW	2 × 22 GBps (peak 2 × 32 GBps)
Read BW from LLC → L2	2 × 85 GBps
Write BW from L2 → LLC	2 × 26 GBps
QPI BW per direction	11 GBps

Table I
Platform Characteristics of the dual-socket Intel Xeon X5570 CPU. The factor of 2 in all rows except for QPI bandwidth is a result of our platform having two sockets.

The corresponding bandwidth for accessing *VIS* on $\mathcal{N}_\mathcal{S}$ sockets:

$$\mathcal{B}'_{VIS} = \rho' \mathcal{N}_\mathcal{S} \left(\max\left(\frac{\rho'}{\mathcal{B}_{LLC \rightarrow L2}} + \frac{1}{\mathcal{B}_{L2 \rightarrow LLC}}, \frac{1}{\mathcal{B}_{QPI}} \right) \right)^{-1} \quad \text{(IV.4)}$$

V. Evaluation

We now evaluate the performance of our traversal algorithm on an Intel® Xeon® X5570 Nehalem system. The platform has 2 sockets with a total of 8 cores running at 2.93 GHz. The peak compute and bandwidth are shown in Table I. Each core has a 32 KB L1 cache, and a 256 KB L2 cache. All cores within each socket share a last level L3 cache (LLC) of 8 MB. All caches of both sockets are kept coherent. Benchmarking efforts such as the work by Molka et al. [21] have indicated the read and write bandwidths of various levels of the cache hierarchy and inter-socket communication shown in Table I. Our system has 96 GB RAM and runs Fedora Linux Release 14. We use the ICC ComposerXE compiler and use libnuma for locality-aware memory management.

Benchmarks and Performance Evaluation: We run our graph traversal algorithm on both synthetic and real-world graphs of different sizes and a wide range of graph diameters (the maximum number of edges that need to traversed to reach any graph vertex from any other vertex). For each graph, we run our BFS algorithm five times each with a different starting vertex. We use a performance metric of edges/second traversed, which is the ratio of the total number of edges traversed to the time taken. We report the average performance in our results. We traverse over 98% of all edges in the original graph in each of our runs. For a fair comparison with previous results, we take in the input graphs as given, and do not reorder the vertices in the graph to improve locality. We use two categories of synthetic graphs that have been previously used in evaluating graph algorithms: (1) **Uniformly Random (UR) graphs** where all $|\mathcal{V}|$ vertices have the same degree d and all d neighbors are chosen randomly[5]; and (2) **R-MAT graphs** where the number of neighbors of each vertex follows a power-law distribution [22]. This models scenarios in a number of real-world graphs including social networks. We use the GTGraph [23] graph generator suite to generate these graphs. We use the parameters a=0.57, b=c=0.19 and d=0.05 for generating small world RMAT graphs. These parameters are identical to the ones used for generating synthetic instances in the Graph 500 BFS benchmark [9]. R-MAT graphs make for interesting test instances with non-trivial load-balancing due to skewed degree distributions [11].

[5]We get similar results for random graphs where both source and destination vertices of each edge are chosen randomly; we do not show results for these in this paper.

Category	Graph	# Vertices	# Edges	Depth
Univ. of Florida Sp. Mat. Collection [24]	FreeScale1	3.43 M	17.1 M	128
	Wikipedia	2.40 M	41.9 M	460
	Cage15	5.15 M	99.2 M	50
	Nlpkkt160	8.35 M	225.4 M	163
USA Road Networks [2]	USA-West	6.26 M	15.24 M	2873
	USA-All	23.94 M	58.33 M	6230
Social Networks	Orkut [4]	3.07 M	223.5 M	7
	Twitter [3]	61.57 M	1468.36 M	13
	Facebook [5]	2.94 M	41.92 M	11
Graph500 [9]	Toy++	256 M	4096 M	6

Table II
Graph Characteristics for real-world graphs.

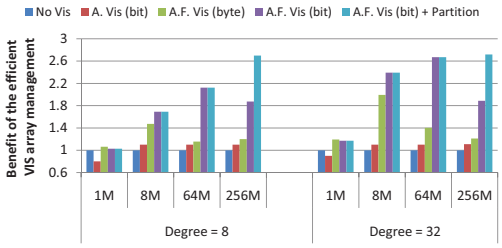

Figure 4. **Relative performance of different representations of the *VIS* array versus a baseline scheme with no *VIS* array on Uniformly Random graphs with varying sizes and degrees. Performance of the atomic-based (A. Vis) bit scheme, and Atomic Free (A.F. Vis) byte, bit and our partitioning scheme are compared.**

We also evaluate the performance of our algorithm on **Real-World graphs** shown in Table II. These graphs exhibit a wide range of *graph sizes* (2.4M–256M vertices), *vertex degrees* (2.4–74.4) and *graph diameters* (6–6230). This includes the Toy++ example (scale=28 and edgefactor=16) from Graph 500 benchmark suite [9] (the largest we could run on our system).

A. Benefit of Schemes

We show the benefit of using different techniques for checking whether a vertex has been assigned, as well as benefit of various schemes to optimize for multi-socket performance.

Maintaining the *VIS* array: Figure 4 shows the benefit of using auxiliary bit-array structures for the *VIS* array for Uniformly Random graphs with varying number of vertices. The first scheme involves a direct check of the $\mathcal{DP}$ of each neighboring vertex **without maintaining a *VIS* array**. Since the neighbors are randomly distributed, this results in a random access to the depth array. For a small number of vertices of 2M or less, the $\mathcal{DP}$ array fits in cache, and the figure shows that this random access does not degrade performance significantly. However, for $|\mathcal{V}| > 2$M, the $\mathcal{DP}$ array is too large (16MB) to fit into the LLC, and the resulting random access to memory can (depending on locality) require

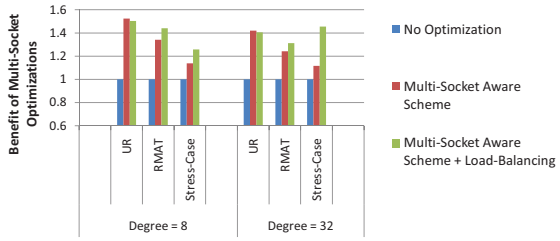

Figure 5. **Relative performance of our Multi-Socket aware and Multi-Socket plus load-balancing schemes compared to a scheme with no multi-socket optimizations. Results are shown on Uniformly Random (UR), RMAT and Stress-Case graphs with $|\mathcal{V}|$ = 16 million and varying degrees.**

a bandwidth of as much as an entire cache-line (64 bytes) per depth access. In practice, we see a **1.7–2.7X performance drop from the best scheme**, the gap increasing with $|\mathcal{V}|$.

All other schemes involve using a *VIS* array. The second scheme involves **atomic instructions** to resolve concurrent updates to a bit-vector *VIS* array (as in Agarwal et al. [18]). This incurs high latency, and the resulting performance is only 10% faster at best (and sometimes even slower) than not maintaining any *VIS* array. In contrast, we use atomic-free updates of the *Depth* array to eliminate latency impacts. We show results for three schemes—using 1 **byte** per vertex, 1 **bit** per vertex or a **partitioned bit array** to further reduce the size of the structure. For graphs with fewer than 16M vertices, the byte structure fits in LLC. The figure shows that the byte scheme performs about 1.4–2X better than that no-*VIS* scheme for graphs with 8M vertices. However, our results show that the bit array scheme outperforms the byte scheme even when the byte structure fits in LLC. Our bit-scheme puts lower pressure on the LLC and hence results in better performance. **For larger graphs (64M in the figure)**, the byte-structure stops fitting in LLC; and the **bit-scheme outperforms other schemes by 1.4–1.9X**. Finally, for very large graphs of 256M or beyond, even the bit-structure does not fit in LLC; in such cases we partition the bit-structure as described in Section III-A. This results in a further 1.3X improvement for $|\mathcal{V}|$ = 256M (we use $\mathcal{N}_{VIS}$ = 4 partitions). This partitioning scheme degenerates to the bit-scheme for smaller sized graphs and does not improve performance for those cases. In all cases, **our best scheme closely matches the prediction of the analytical model**.

Effect of multi-socket optimizations: Fig. 5 shows the effect of our multi-socket optimizations described in Section III-B on the performance of graph traversal on various graphs (UR, R-MAT, and a stress case we describe shortly). We use graphs with 16M vertices and degrees 8 and 32 for this experiment. The first scheme has no multi-socket optimizations and performs spatially incoherent accesses causing cross-socket cache-line updates and ping-ponging. This results in poor cross-socket scaling; from the figure, this scheme **consistently performs the worst** among the various schemes.

The other schemes involve a tradeoff between data locality and load balanced computation (using the two phase execution described in Section III). The multi-socket aware scheme reduces inter-socket communication at the expense of load-imbalance, while the load-balanced scheme evenly divides each $\mathcal{PBV}_t$ bin between the sockets while keeping cross-socket communication low (Section III-B3). The **UR graphs do not have any load-imbalance** since each vertex has neighbors that are equally likely to fall into any of the $\mathcal{N}_{PBV}$ bins. Hence both the multi-socket aware and load-balancing **schemes perform similarly**. We also saw similar trends with random graphs, where each vertex does not necessarily have the same degree, but the choice of neighbor is still random. On **R-MAT graphs**, we find some bias in the neighbors. In practice, for a=0.57, b=c=0.19, we find that an average of 60% of the enqueued vertices are assigned to one socket (α_{Adj} = 0.6). We see there is a **benefit of about 5–10% for our load-balancing scheme**.

We also run our experiments on a **stress-case graph**, which is a bipartite graph where all vertices in the $\mathcal{BV}_t^C$ array are either small or large (at alternate depths)—and hence always belong to one of the two sockets. While this has been designed to exercise the worst case load-balancing, we see similar characteristics in some of our real-world graphs including the Nlpkkt160 graph shown later. For the stress-case graphs, our load-balancing scheme shows **as much as 30% performance improvement** over the multi-socket scheme.

We note that with increasing degree, the first scheme with no multi-socket optimizations improves in performance over the multi-socket scheme (but is always slower). The reason is that for large degrees, most of the cross-socket *VIS* traffic is read-only rather than read-write (since each vertex is only updated once but read as many times as the degree). Read-only traffic has lower latency and bandwidth requirements. Another interesting trend is that the load-balancing scheme performs especially well for large degree stress cases. This is because most of the benefit from load-balancing comes from the fact that the internal LLC bandwidth of both sockets is now utilized while reading the *VIS* array. The bandwidth required for reading the *VIS* array is a bigger bottleneck for large degree graphs, since the *VIS* array is accessed on a per-edge basis and large degree graphs have more edges per vertex than small degree graphs. Hence the benefit of load-balancing is higher for larger degree graphs.

Effect of latency hiding: Performing the rearrangement of $\mathcal{BV}_t^N$ arrays gains us an average of 1.15X in performance. Furthermore, the use of SIMD while binning into $\mathcal{PBV}_t$ yields a 1.3–2X instruction reduction, and results in utilization of all bandwidth resources.

B. Comparison to Other Approaches

Figure 6 compares the performance of our approach (maintaining a cache-resident *VIS* array with load-balancing for multi-socket) versus previous best performance reported by

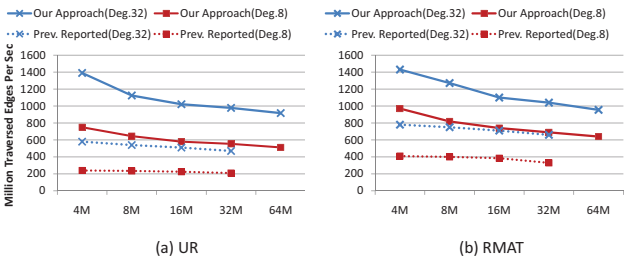

Figure 6. **Performance of our cache-friendly load-balanced approach in Million Traversed Edges Per Second versus previous best reported numbers (from Agarwal et al. [18]) for (a) Uniformly Random and (b) RMAT graphs with varying sizes and degrees. Our scheme performs 1.5–3X better than previously reported results on the same platform.**

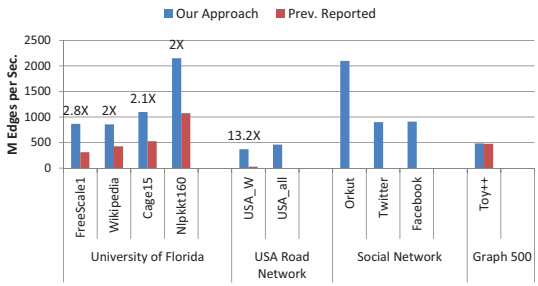

Figure 7. **Performance of our approach in Millions of Traversed Edges Per Second versus previous best performance on real-world graphs. All competing performance numbers (except Toy++) have been measured on our system. Our approach is 2–13.2X faster than previous approaches. Previous results for USA_all, Orkut, Twitter and Facebook graphs were not available.**

Agarwal et al. [18] on **(a) UR and (b) RMAT graphs** for different number of vertices and degrees. The figures show that **our scheme obtains 1.5–3X better performance using an identical system**. We obtain near-linear socket scaling (around 1.98X for UR, and 1.93X for RMAT graphs on 2 sockets). Furthermore, **our 2-socket numbers are 1.1–2.1X better than the best reported 4-socket Nehalem-EX numbers** [18]. This is due to our latency hiding techniques (including atomic-free depth updates, prefetch, use of rearrangement and SIMD) shown in the previous graphs, coupled with a (small) improvement due to use of load balancing for R-MAT graphs. Further, **our results on UR graphs are about 10.5X better** than those from Xia and Prasanna [19].Note that our implementation utilizes bandwidth efficiently, and that our performance results match well with our analytical model. Our model further predicts that we will scale by another 1.8X on a 4-socket Nehalem-EX system. We also ran our algorithm on random graphs, where each edge has a random start and end vertex. As predicted by our model, our performance results do not change, since there is no load-imbalance in the average case.

Figure 7 shows our results on a variety of **real-world graphs** from different fields with different graph characteristics. We compare our performance to the best reported performance whenever available. Since previous results have been reported on different platforms, we re-implemented their algorithms (or ran their code if publicly available) on our system for a fair comparison.

The first four graphs are taken from the **University of Florida sparse matrix suite**, and the previous best results for these graphs are from Leiserson et al. [25]. Our scheme gives **2–2.8X better** performance. These graphs tend to have low diameter and large degrees. For the **USA road graphs** [26], our scheme is **up to 13.2X better in performance**. In contrast to the earlier set of graphs, USA road graphs have very low average degree. Note that these input graphs have vertices ordered in a way that there is significant spatial coherence between the vertex and its neighbors, which our model does not take into account. Our model thus overestimates the bandwidth required both from DDR to LLC and from LLC in this case. The third set of graphs comes from social networks. These graphs have low diameter and large degrees. There were no published results with these graphs; hence we only show our results. For social network graphs, our performance closely matches the analytical model to within 10%. The final graph is the **Toy++ example from the Graph500 benchmarks** [1]. As for R-MAT graphs, we find that our performance is again within 5% of the analytical model. Further, our absolute performance on an 8-core 2-socket Nehalem-EP system **matches the reported performance on a Red-Sky supercomputer system with 512 processors** of the same processor type [9] on the Nov. 2010 list. Note that we halved our performance results for consistent reporting with the Graph500 website results.

C. Validating Performance Model

Figure 8 compares the cycles per element spent in Phase-I and Phase-II for our best scheme versus that predicted by the analytical model. All results are shown on 2 sockets with partitioning of the *VIS* array, and with the load-balancing scheme. For both UR and R-MAT graphs, our results for match the model to within 5–10% on the average over a variety of graph sizes and degrees (details in Appendix D).

We now show our model parameters and final predictions for an example RMAT graph with $|\mathcal{V}|$ = 8M and degree 8. For our example RMAT graph, $|\mathcal{V}'|$ = 4M, $|\mathcal{E}'|$ = 61.2M, hence ρ' is 15.3. Here $\mathcal{N}_{PBV}$ = 2, $\mathcal{L}$=64 and $\mathcal{D}$=6. Further, $|\mathcal{L}2|$=256 KB, $|VIS|$ = 8M and $\mathcal{N}_{VIS}$ = 1. Plugging these into equations IV.1a–IV.1d and IV.2, and using the bandwidth numbers from Table I, we obtain single-socket execution time of 6.48 cycles/edge. For our **dual-socket** platform, we obtain α'_{Adj}=0.6 for depths 3 and 4 that dominate the execution time. Computing the effective bandwidths and using these in Eqn. IV.3 and IV.4 gives **overall** cycles of 3.47 cycles/edge (**844 M edges/second**). We achieve 820M edges/second, which is only 3% off our predictions.

VI. Related Work

Breadth-first search (BFS) is a key example of graph traversal that exhibits many performance problems commonly present in traversal algorithms. There have been many attempts

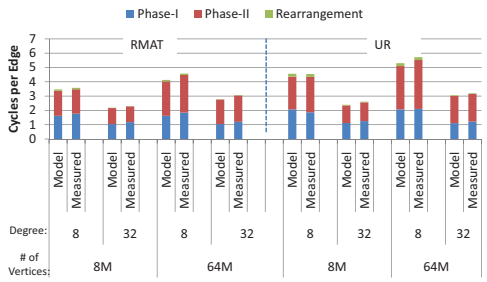

Figure 8. **Comparison of cycles per traversed edge spent in Phase-I, Phase-II and Rearrangement steps of our algorithm versus those predicted by the analytical model presented in Section IV for RMAT and Uniformly Random graphs with varying sizes and degrees. Our model matches achieved performance to within 5–10% on the average.**

to improve the speed of BFS. Asynchronous BFS traversal which eliminate barrier synchronization have been previously studied in the context of graphs with large diameters [27], [28], [29]. Synchronous BFS algorithms (i.e., Bulk Synchronous Parallel or wavefront algorithm) [18], [7], [30], [25], [26], [10], [16] are however inherently more work-efficient in that they guarantee that the depth of all vertices is updated exactly once. Synchronous BFS traversal has been implemented on several special-purpose high-performance platforms [17], [30], [31], [32], [26], [16], as well as multi/many-core CPUs with increasing number of on-die cores [18], [25], [19]; however, none of these have fully utilized bandwidth resources due to the impact of memory latency. On special purpose platforms, previous work tends to focus on statically partitioning graph vertices to individual processing units at the expense of load-balancing because dynamic load-balancing schemes are difficult to implement in the absence of cache coherence. On multi-core CPUs, parallel BFS algorithms that load-balance computation on different threads have been described by Leiserson et al. [25] and Xia and Prasanna [19]. For example, Leiserson et al. use the Cilk++ framework with a work-stealing based scheduler to ensure load-balancing between threads. However, these works have not explored the use of bitmaps and their scheme has a 2–10X performance gap as compared to our scheme even for relatively small sized graphs. This gap will become larger with increasing graph sizes due to the inefficient use of memory bandwidth.

The closest study to this paper is the paper by Agarwal et al. [18]. They performed optimizations for multi-socket CPUs and showed that their implementation was 2.4X and 5X higher in performance than previous results on the Cray XMT and BlueGene/L processors respectively. They replaced the locks in previous implementations with lower cost atomic operations and exploit bitmaps to filter out memory accesses. Our approach is different from their paper in that 1) our approach is not only lock-free, but also atomic-free. While atomic operations are cheaper than locks, they still have higher latency than regular instructions. 2) Unlike their scheme, our multi-socket optimization ensures load-balancing between sockets by dynamically balancing the number of vertices processed between sockets; our scheme offers good socket scaling even for adversarial, heavily skewed graphs. Also, coupled with TLB optimization and the use of SIMD, we achieve 1.5–3X speedup over their approach on the same multi-socket platform. Finally, we propose a generalized cache-resident representation that can handle arbitrary sizes of graphs. This is critical to performance with current trends of increasing graph sizes.

VII. Conclusion

In this paper, we presented an efficient graph traversal algorithm that utilizes all available bandwidth resources. We presented an analytical model that accurately projects performance, and provides suggestions for improving graph traversal performance on future architectures. Our algorithm is useful as a building block for efficient multi-node implementations, and allows these implementations to ride the trend of increasing per-node compute and bandwidth resources.

References

[1] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray User's Group*, May 2010.

[2] C. Demetrescui, A. Goldberg, and D. Johnson, "The Ninth DIMACS Implementation Challenge: The Shortest Path Problem," http://dimacs.rutgers.edu /Workshops/Challenge9/.

[3] H. Kwak, C. Lee, H. Park, and S. B. Moon, "What is twitter, a social network or a news media?" in *WWW*, 2010, pp. 591–600.

[4] A. Mislove, M. Marcon, P. K. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Internet Measurement Conf.*, '07, pp. 29–42.

[5] C. Wilson, B. Boe, A. Sala, K. P. N. Puttaswamy, and B. Y. Zhao, "User interactions in social networks and their implications," in *EuroSys*, 2009, pp. 205–218.

[6] M. Anderson, "Better benchmarking for supercomputers," *IEEE Spectrum*, vol. 48, pp. 12–14, January 2011.

[7] D. A. Bader and K. Madduri, "Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2," in *ICPP*, 2006, pp. 523–530.

[8] A. Yoo, E. Chow, K. W. Henderson, W. M. III, B. Hendrickson, and Ü. V. Çatalyürek, "A scalable distributed parallel breadth-first search algorithm on bluegene/l," in *SC*, 2005, p. 25.

[9] "The Graph 500 List (Nov 2010)," http://www.graph500.org.

[10] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *SIGMOD*, 2010, pp. 135–146.

[11] A. Buluç and K. Madduri, "Parallel breadth-first search on distributed memory systems," *CoRR*, vol. abs/1104.4518, 2011.

[12] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele, "Many-Core Key-Value Store," in *To Appear at Second Intl. Green Computing Conference (IGCC 2011)*.

[13] W. Lang, J. M. Patel, and S. Shankar, "Wimpy node clusters: what about non-wimpy workloads?" in *DaMon*, '10, pp. 47–55.

[14] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan, "FAWN: a fast array of wimpy nodes," in *SIGOPS*, 2009, pp. 1–14.

[15] "Intel 64 and IA-32 Architectures Software Developer's Manual, System Programming Guide, Part 1," *Intel Press*, vol. 3A, '11.

[16] D. P. Scarpazza, O. Villa, and F. Petrini, "Efficient breadth-first search on the cell/be processor," *IEEE Trans. Parallel Distrib. Syst.*, vol. 19, no. 10, pp. 1381–1395, 2008.
[17] S. Edelkamp, D. Sulewski, and C. Yücel, "Perfect hashing for state space exploration on the gpu," in *ICAPS*, 2010, pp. 57–64.
[18] V. Agarwal, F. Petrini, D. Pasetto, and D. A. Bader, "Scalable graph exploration on multicore processors," in *SC*, '10, pp. 1–11.
[19] Y. Xia and V. K. Prasanna, "Topologically Adaptive Parallel Breadth-first Search on Multicore-Processors," in *PDCS*, 2009.
[20] C. Kim, E. Sedlar *et al.*, "Sort vs. Hash Revisited: Fast Join Implementation," *PVLDB*, vol. 2, no. 2, pp. 1378–1389, '09.
[21] D. Molka, D. Hackenberg, R. Schöne, and M. S. Müller, "Memory performance and cache coherency effects on an intel nehalem multiprocessor system," in *PACT*, 2009, pp. 261–270.
[22] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *SDM*, 2004.
[23] D. A. Bader and K. Madduri, "GTgraph: A Synthetic Graph Generator Suite," http://sdm.lbl.gov/ kamesh/software.
[24] T. Davis, "The University of Florida Sparse Matrix Collection," http://www.cise.ufl.edu/research/sparse/matrices.
[25] C. E. Leiserson and T. B. Schardl, "A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers)," in *SPAA*, 2010, pp. 303–314.
[26] L. Luo, M. D. F. Wong, and W. mei Hwu, "An effective gpu implementation of breadth-first search," in *DAC*, '10, pp. 52–55.
[27] F. Guerriero and R. Musmanno, "Parallel Asynchronous Algorithms for the K Shortest Paths Problem," *Journal of Optimization Theory and Applications*, vol. 104, no. 1, pp. 91–108, 2000.
[28] M. A. Hassaan, M. Burtscher, and K. Pingali, "Ordered vs. unordered: a comparison of parallelism and work-efficiency in irregular algorithms," in *PPOPP*, 2011, pp. 3–12.
[29] R. A. Pearce, M. Gokhale, and N. M. Amato, "Multithreaded asynchronous graph traversal for in-memory and semi-external memory," in *SC*, 2010.
[30] S. Hong, S. K. Kim, T. Oguntebi, and K. Olukotun, "Acc. CUDA graph algorithms at max. warp," in *PPOPP*, '11, pp. 267–276.
[31] S. Hong, T. Oguntebi, and K. Olukotun, "Efficient Parallel Graph Exploration on Multi-Core CPU and GPU," in *PACT*, '11, pp. 78–88.
[32] D. Merrill, M. Garland, and A. Grimshaw, "High Performance and Scalable GPU Graph Traversal," Univ. of Virginia, Tech. Rep. UVA CS-2011-05, 2011.

APPENDIX

To compute the memory traffic, we focus on the amount of data transferred between main memory and LLC, and between LLC and internal caches (for LLC data structures like partitioned *VIS*). Refer to Sec. IV for the notation.

A. Computation of Data Transfer

Figure 3 describes the complete algorithm. We first compute the total data transferred for Phase-I:

1.1 Traversing through $\mathcal{BV}_t^C$: Since each assigned vertex is enqueued once in the $\mathcal{BV}_t^C$ array, the total amount of data read equals $4|\mathcal{V}'|$ bytes.

1.2 Reading address of the location storing neighbors of the vertex: For each vertex in $\mathcal{BV}_t^C$, we first need to access the address where the neighbors are stored. Since successive vertexes may point to spatially incoherent regions of memory, each access would bring in a complete cache-line, for a total of $\mathcal{L}|\mathcal{V}''|$ bytes.

1.3 Traversing through the neighbors of a vertex: For a vertex with ρ'neighbors, we need to read $4(1+\rho')$ bytes (reading the number of neighbors followed by the neighbors). Since data transfers occur at the granularity of cache-lines, it can be shown (in the average case) that the number of cache-lines read would be $(1+4\rho'/\mathcal{L})$. Hence, total amount of data read equals $\mathcal{L}|\mathcal{V}''|(1+4\rho'/\mathcal{L})$ bytes.

1.4 Writing $\mathcal{PBV}_t$: While binning the neighbors for a given vertex, we first write the vertex id to each bin (and mark it as a parent for the next phase), and then partition the neighbors amongst the $\mathcal{N}_{PBV}$ bins. Hence, for a vertex with ρ'neighbors, the total amount of data written equals $4|\mathcal{V}''|(\mathcal{N}_{PBV} + \rho')$ bytes. Since writing also brings in data for reading, the total amount of memory traffic equals $8|\mathcal{V}''|(\mathcal{N}_{PBV} + \rho')$ bytes.

Adding up, and dividing by the number of traversed edges leads to Eqn. IV.1a.

We now compute the total accesses required for Phase-II:

2.1 Traversing through $\mathcal{PBV}_t$: The data that was written during the last step (#1.4) of Phase-I is now read by Phase-II. This data equals $4|\mathcal{V}''|(\mathcal{N}_{PBV} + \rho')$ bytes.

2.2 Reading *VIS* in to LLC: For each *step* of the traversal, potentially all the $\mathcal{N}_{VIS}$ partitions of *VIS* need to be read in completely from the main memory to LLC. Since each of the $\mathcal{N}_{VIS}$ partitions of *VIS* can completely reside in LLC, the various accesses to *VIS* will fetch data either from LLC or L2 of the cores. Hence, total main memory traffic generated equals $(\mathcal{D})(|VIS|)$ bytes.

2.3 Updating $\mathcal{DP}$: The *depth* and *parent* of a vertex is updated *once*, when the corresponding bit in *VIS* is equal to 0. Since there exists potentially no spatial coherence in the order of vertices for this update, a cache-line is brought into the LLC, the corresponding data updated, and the cache-line written back. Hence each update generates a traffic of $2\mathcal{L}$ bytes—a total of $2\mathcal{L}|\mathcal{V}''|$ bytes.

2.4 Writing $\mathcal{BV}_t^N$: Each thread appends the list of vertices whose *depth* was updated into the $\mathcal{BV}_t^N$ array, which is then read at the beginning of the Phase-I of the *next* step. The total amount of data written equals $4|\mathcal{V}''|$ bytes—a net memory traffic of $8|\mathcal{V}''|$ bytes.

Adding up, and dividing by the number of traversed edges leads to Eqn. IV.1b. In addition to the main memory traffic, we need to consider the memory traffic generated inside the core while accessing the *VIS* array. We need to consider following *two* scenarios separately:

2.5 Updating a bit in *VIS* from 0 to 1: As explained in Sec. III-A, once an access to *VIS* returns a value of 0, the corresponding bit is set to 1, followed by potential updates to the $\mathcal{DP}$ array. When updating *VIS*, the cache-line may be brought into L2 caches. Then, when the cache-line is flushed out of L2 to LLC, it adds to the traffic to LLC. Hence *for each assigned vertex*, the LLC read traffic equals $\mathcal{L}$ bytes and LLC write traffic equals $\mathcal{L}$ bytes.

2.6 Accessing a bit (already set to 1) in *VIS*: When the relevant cache-line already resides in the core's L2, then there

is no extra traffic. Otherwise, it either needs to be fetched from LLC, or some other core's L2 (since it was accessed for updating it by that core). In case the cache-line resides in LLC, it needs to be transferred to the core's L2. In case the cache-line resides in a remote L2, it needs to be transferred to LLC, and then read from LLC. Since we have already covered the write traffic to LLC, this also generates an additional read request from LLC. Since each vertex is accessed ρ' times, the total amount of traffic generated *from LLC for each vertex that is not resident in L2* is $(\rho'\text{-}1)(\mathcal{L})$ bytes.

The exact number of cases where a cache-line needs to be fetched from LLC depends on the micro-architectural execution of the caches, timing of data movement, etc. Hence, we use an *average case analysis*, and compute the expected number of bytes transferred by assigning the probability of a L2 hit equal to the size of L2 divided by the total partitioned size of *VIS* (assuming $|VIS| \geq |\mathcal{L}2|$). This leads to Eqn. IV.1c.

3. Rearrangement Phase: As described in Section III-B3, we compute a histogram, scatter the vertices in $\mathcal{BV}_t^N$ into a temporary array, and then copy from the temporary array back to $\mathcal{BV}_t^N$. Hence the total amount of traffic generated w.r.t. main memory equals $(4 + 8 + 4 + 8)|\mathcal{V}''|$ bytes (Eqn. IV.1d).

B. Execution Time on Single Socket

For a single-socket implementation, we assume all the memory allocations are done on the local memory. Recent benchmarking efforts by Molka et al. [21] indicate that the LLC has the 256-bit interface used for both reading and writing, and shared by all on-die cores. Thus we need to *add up* the times (Eqns. IV.1a, IV.1b, IV.1c and IV.1d) to obtain Eqn. IV.2.

C. Execution Time on Multiple Sockets

For remote transfers, the data is transferred through the QPI, at a slower bandwidth ($\mathcal{B}_{QPI}$) than the main memory bandwidth ($\mathcal{B}_M$). Our benchmarks indicate that for any socket, the *achievable bandwidth* to remote and local LLC are *respectively proportional* to the amount of data being transferred to remote and local LLC (up to the max. achievable bandwidth on each). Recall from Section III that $\mathcal{A}dj$, *VIS*, $\mathcal{DP}$ are all evenly divided between the $\mathcal{N}_\mathcal{S}$ sockets, while each thread locally allocates $\mathcal{BV}_t^C$, $\mathcal{BV}_t^N$ and $\mathcal{N}_{PBV}$ bins of $\mathcal{PBV}_t$. In order to accurately account for the disproportionate amount of memory accesses to different data structures, we use the following *four* terms—$\alpha_{\mathcal{A}dj}$, $\alpha_{\mathcal{BV}_C}$, $\alpha_{\mathcal{PBV}_t}$and $\alpha_{\mathcal{DP}}$(defined in Section IV).

For example, if 63% of all the accesses to $\mathcal{A}dj$ are being fetched from socket #1's memory, then $\alpha_{\mathcal{A}dj}$= 0.63 [6]. We take the maximum value, since the time taken to process these accesses from socket #1 would be *greater* than the time taken from any other socket, and hence become the overall run-times for accessing $\mathcal{A}dj$. Let us now develop an analytical expression for the resultant bandwidth w.r.t. accesses to $\mathcal{A}dj$. A similar formulation can be applied to $\mathcal{BV}_t^C$, $\mathcal{BV}_t^N$, $\mathcal{PBV}_t$ and $\mathcal{DP}$.

[6]This ratio is a property of the boundary states for a given *step* of the traversal, and may change for every *step*.

In case $\alpha_{\mathcal{A}dj}$ *equals* $(1/\mathcal{N}_\mathcal{S})$, the *effective bandwidth would be* $\mathcal{B}_M\mathcal{N}_\mathcal{S}$. Assume $\alpha_{\mathcal{A}dj} > 1/\mathcal{N}_\mathcal{S}$ for the rest of the discussion. In case of no load-balancing, all accesses to $\mathcal{A}dj$ would be local, and the effective bandwidth for the $\mathcal{N}_\mathcal{S}$ sockets would be $\mathcal{B}_M/\alpha_{\mathcal{A}dj}$. Using *our load-balancing scheme*, we transfer $(1/\mathcal{N}_\mathcal{S})$ fraction of $\mathcal{A}dj$ to the local memory, while the remaining accesses are divided between the remaining sockets. Assuming equal distribution of accesses leads to Eqn. IV.3.

Eqn. IV.3 adds up the time taken to transfer the data over QPI (up to a maximum rate of $\mathcal{B}_{QPI}$), and the time taken to transfer data from LLC to the internal caches. As an example, for $\mathcal{N}_\mathcal{S}$ = 4 and $\alpha_{\mathcal{A}dj}$ = 0.7, the effective bandwidth (using parameters in Table I) evaluates to $2.7\mathcal{B}_M$(as opposed to $1.42\mathcal{B}_M$for non load-balanced), which is a **speedup of 1.9X** due to load-balancing. Similar exp. can be derived for $\mathcal{BV}_t^C$, $\mathcal{BV}_t^N$, $\mathcal{PBV}_t$ and $\mathcal{DP}$.

We now focus on the access of *VIS* array in a multi-socket environment. For a non-load-balanced implementation, the *effective bandwidths* for writing from L2 to LLC ($\mathcal{B}_{L2\rightarrow LLC}$) and reading from LLC to L2 ($\mathcal{B}_{L2\rightarrow LLC}$) will both scale by $(1/\alpha_{\mathcal{PBV}_t})$ (same as the disparity in the $\mathcal{PBV}_t$ array). As a result of our load-balancing scheme, there may be accesses to *VIS* by two or more sockets. However, cross-socket communication will occur when a bit has been updated from 0 to 1, and hence the updated cache-line needs to be transferred from remote to local LLC. Since the QPI transfers can happen in parallel, the total time taken is equal to the greater of the time taken on each socket's LLC as compared to the time taken to transfer the cache-line over QPI. This leads to Eqn. IV.4.

D. Detailed Validation of Performance Model

We expand on the discussion in Sec. V-C. RMAT graphs contain a number of isolated vertices which are never visited, and consequently the average degree of vertices assigned a depth (ρ') is 15.3. Plugging in Phase-I results in 21.7 bytes/edge of DDR traffic (Eqn. IV.1a), while the Phase-II DDR traffic is 13.54 bytes/edge (Eqn. IV.1b). The LLC traffic for Phase-II is 51.1 bytes/edge (Eqn. IV.1c). Finally, rearrangement only takes 1.6 bytes/edge (Eqn. IV.1d). For a single socket, Eqn. IV.2 predicts that the single-socket time for Phase-I is 2.88 cycles/edge, while Phase-II takes a total of 1.8 + (1- 1/4)*2.67 = 3.80 cycles/edge, where the (1- 1/4) factor comes from the prob. of the *VIS* structure fitting in L2.When we move to **dual-socket**, we found from our runs that the value of α_{Adj} = 0.6. Eqn. IV.3 shows that the **effective BW increase of 1.7X** over 1 socket. Hence **Phase-I** cycles reduce by 1.7X to **1.62 cycles/edge** on 2 sockets. The DDR cycles for Phase-II also reduces by 1.7X to 1.06. However, according to Eqn (A.7), the internal LLC bandwidths nearly doubles. Further the (1 - 1/4) factor becomes (1 - 1/2), since the **effective cache size doubles on 2 sockets**. Hence, we obtain **1.75 cycles/edge for Phase-II**. The rearrangement step scales linearly to 0.10 cycles/edge on 2 sockets. This yields **overall** cycles of 3.47 cycles/edge (**844 M edges/second**). We achieve 820M edges/second, which is only 3% off our predictions.