Fine-Grained Task Migration for Graph Algorithms using Processing in Memory

Paula Aguilera[1], Dong Ping Zhang[2], Nam Sung Kim[3] and Nuwan Jayasena[2]

[1]Dept. of Electrical and Computer Engineering
University of Wisconsin-Madison
paguilera@wisc.edu

[2]AMD Research
{Dongping.Zhang, Nuwan.Jayasena}@amd.com

[3]Dept. of Electrical and Computer Engineering
University of Illinois Urbana-Champaign
nskim@illinois.edu

Abstract— **Graphs are used in a wide variety of application domains, from social science to machine learning. Graph algorithms present large numbers of irregular accesses with little data reuse to amortize the high cost of memory accesses, requiring high memory bandwidth. Processing in memory (PIM) implemented through 3D die-stacking can deliver this high memory bandwidth. In a system with multiple memory modules with PIM, the in-memory compute logic has low latency and high bandwidth access to its local memory, while accesses to remote memory introduce high latency and energy consumption. Ideally, in such a system, computation and data are partitioned among the PIM devices to maximize data locality. But the irregular memory access patterns present in graph applications make it difficult to guarantee that the computation in each PIM device will only access its local data. A large number of remote memory accesses can negate the benefits of using PIM.**

In this paper, we examine the feasibility and potential of fine-grained work migration to reduce remote data accesses in systems with multiple PIM devices. First, we propose a data-driven implementation of our study algorithms: breadth-first search (BFS), single source shortest path (SSSP) and betweenness centrality (BC) where each PIM has a queue where the vertices that it needs to process are held. New vertices that need to be processed are enqueued at the PIM device co-located with the memory that stores those vertices. Second, we propose hardware support that takes advantage of PIM to implement highly efficient queues that improve the performance of the queuing framework by up to 16.7%. Third, we develop a timing model for the queueing framework to explore the benefits of work migration vs. remote memory accesses. And, finally, our analysis using the above framework shows that naïve task migration can lead to performance degradations and identifies trade-offs among data locality, redundant computation, and load balance among PIM devices that must be taken into account to realize the potential benefits of fine-grain task migration.

Keywords—Graph Algorithms; Processing In Memory.

I. Introduction

While processors have gradually increased their computing capabilities, the main memory has not experienced the same degree of improvement, particularly falling behind in latency and energy consumption [1]. At the same time moving data from the location where it is stored to the processor where it is used for computation is highly inefficient for applications with irregular memory access patterns and low data reuse. As a result, the memory system is becoming responsible for an increasing percentage of the total system energy consumption. A solution to this problem is to place the computation closer to its data, thereby reducing the movement of data through the memory hierarchy resulting in energy savings and performance improvement [2].

3D-stacking is an enabling technology that allows multiple dies to be stacked on top of each other in the same package. A memory module with processing in memory (PIM), or a *PIM stack*, implemented using 3D stacking places one or more memory dies atop (or under) a logic die implementing compute capabilities. We study a system (Figure 2) that consists of a host processor and multiple such PIM stacks. Each in-memory processor has low-latency, high-bandwidth, and low-energy access to data in the *local* memory stacked on top of it. Access to *remote* data in other memory modules incur higher latency, lower bandwidth, and higher energy consumption. Ideally, computation and data are partitioned among the PIM stacks in such a way that data locality is maximized. However, some computations present irregular, data-dependent memory access patterns and partitioning the data and computation statically cannot completely eliminate remote memory accesses.

A large number of remote memory accesses can negate the benefits of PIM. In these situations migrating work to execute on the PIM device co-located with the data that it accesses can be more efficient than fetching the data from a remote PIM module, when the overhead of migrating work is lower than that of performing remote memory accesses.

Graph processing algorithms are fundamental to many application domains today. Algorithms such as breadth-first search, connected components, and shortest path are frequent in graph analysis. Some of these algorithms are performed over very large graphs, often consisting of millions of vertices. Graph algorithms exhibit memory-intensive behavior with irregular and graph-topology-dependent memory access patterns and limited data reuse, resulting in high memory bandwidth demand. These characteristics make them a good match for PIM systems. However, the irregular memory access patterns make it challenging to partition the computation and data across the multiple PIM stacks so that computation is co-located with its data and remote memory accesses are avoided. Therefore, we explore work migration techniques targeted towards graph processing algorithms to improve their performance in systems with multiple PIM stacks.

A large body of research exists on work and task migration in the context of distributed systems and large-scale machines. However, PIM introduces a number of new considerations that warrant revisiting the topic. The PIM stacks within a single system, in some ways, resemble non-uniform memory access (NUMA) architectures. However, the close coupling of PIM to main memory makes local memory access extremely inexpensive from both performance and power perspectives relative to traditional NUMA machines and other forms of multi-processor systems. Further, the PIM stacks can be interconnected via memory interfaces enabling efficient, fine-grain communication (e.g., a single cache line) among them. In addition, PIM's proximity to memory enables highly efficient, hardware-assisted implementations of queue primitives. These factors justify and enable a finer granularity of tasks than many of the prior studies. In our study, we consider work migration at the granularity of the processing performed on a vertex.

In cases where a large graph is distributed across multiple memory modules, we anticipate the common case is for all PIM devices to execute the same code on different subsets of data. Therefore, task migration in these cases does not require moving code between devices as the code base is identical and is already available at each PIM stack. Task migration simply involves communicating vertices, edges, or other data elements to be processed at a remote stack. Therefore, migrating a task can be as simple and as low overhead as communicating a single word (e.g., a vertex ID) to a remote PIM device. In many ways, this is similar to an *active message* [3] and does not require migration of a register set or context state.

First, we propose a queuing framework to implement fine-grained migration; using this framework, vertex IDs are sent to the PIM stacks that will process them. Second, we propose hardware mechanisms that take advantage of PIM's proximity to memory to implement efficient queues and reduce the overhead of work migration. Third, we develop a high level timing model for our queuing framework to evaluate the proposed hardware support for efficient queues and to study the performance of work migration vs. a *baseline* where vertices are enqueued in a round robin fashion to the PIM stacks and remote memory accesses are performed as necessary. We propose and evaluate a variety of task migration strategies that range from strict migration on any remote vertex access to allowing some amount of remote accesses to reduce redundant computation. We find that load imbalance negatively impacts performance in all of the algorithms, and study how to overcome it. We also account and compare the number of remote vs. local memory accesses performed by the different approaches and estimate the total energy of the memory accesses based on the cost of a remote memory access relative to the cost of a local memory access. This paper makes the following main contributions:

- Propose a queueing framework to perform light-weight fine-grained task migration among PIM devices and introduce hardware mechanisms that take advantage of PIM to implement efficient queuing (Section III).
- Propose a parameterized timing model to estimate the performance of graph algorithms implemented atop this framework (Section IV).
- Use the above timing model to study the performance trade-offs of work migration under a variety of system configurations as well as application and graph characteristics (Section V).

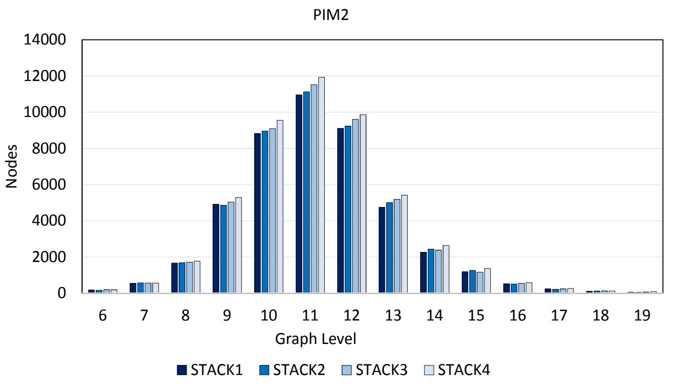


Figure 1. Neighbor distribution of a BFS traversal for a graph that has been partitioned among four PIM stacks. Graph level is the distance from the root vertex. This figure shows the spatial distribution of the neighbors for the vertices at each level of the graph that are located on PIM2.

II. Background and Motivation

A. 3D-Stacked Processing In Memory

This work studies PIM architecture implemented using 3D die stacking technology. 3D stacking allows a compute (PIM) die implemented in a logic process to be tightly coupled with memory dies implemented in a DRAM process. This helps address the problems of high manufacturing costs and low performance of traditional PIM proposals that integrated compute units and DRAM on a single die using the same process technology. 3D stacking provides high-bandwidth and low-energy memory access from PIM to local DRAM due to their proximity and the high density of through silicon via (TSV) interconnections [4]. Applications that require high memory bandwidth and low data reuse can benefit from PIM.

B. Large Scale Graph Processing

Graphs represent relationships between different objects; the vertices are the objects and the edges are the relationships between them. Graphs are common in many different applications and can span very large data sets. Further, graph algorithms have input-dependent memory access patterns.

To illustrate the number of remote vertices that are visited in an example graph algorithm, we look at the behavior of BFS. BFS traverses a graph starting at the root and processing all its direct neighbors first before proceeding to process the next level of neighbors. BFS processes all the vertices that are at the same distance from the root before to processing any vertex located one level further. Although this data is taken from BFS, other graph algorithms show similar characteristics.

We study a Google web graph taken from the Stanford Network Analysis Project (SNAP) [5], where the vertices are webpages and the edges represent the hyperlinks between them. This graph consists of 875,713 vertices and 5,105,039 edges. We evenly partition the graph data structure across a system with four PIM device[1], PIM1 through PIM4. We

[1] While the specific graph partitioning affects the data locality characteristics, our interest is in understanding the effectiveness of task migration independent of the graph partitioning scheme (i.e., not all graphs can be effectively partitioned). Therefore, we use a simple partitioning here. Further, we believe effective task migration can avoid the high overheads typically required in graph partitioning schemes that seek to minimize communication.

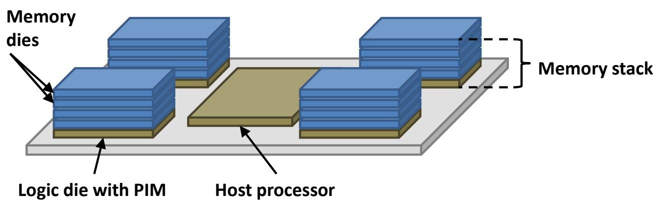


Figure 2. Example system with a host processor and multiple memory devices with PIM capabilities.

perform a vertex-based partitioning, where we place equal numbers of vertices in each of the memory modules. All the outgoing edges of a vertex are placed on the same PIM as the vertex. Figure 1 shows the neighbor distribution for the vertices located in a single PIM device (we arbitrarily select PIM2) at each level of the graph. The vertices that fall in PIM2 are the only neighbors that are local to PIM2. Neighboring vertices located in PIM1, PIM3, and PIM4 are remote to PIM2. As we can see, there is an approximately uniform distribution of neighbor vertices among all the PIM stacks for this graph, resulting in the majority of the neighboring vertices being located in remote PIM devices. Processing such a large number of remote vertices implies many remote memory accesses, which can hurt performance and negate the benefits PIM.

There are two commonly used approaches to parallelize graph algorithms: topology-driven and data-driven [6]. In the first approach all the vertices are visited regardless of whether there is work to perform or not. In the second approach vertices are only visited if they are active and there is work to be done on them. This second implementation uses a worklist to track vertices that need processing; threads read active vertices from the worklist and if, during the processing of a vertex, more vertices become active, they are written to the worklist. Updates to the worklist need to be atomic to prevent conflicts among concurrent updates. While data-driven implementations are more work-efficient, they can suffer from contention when multiple threads are updating the shared worklist [6]. For our work we use the data-driven approach.

III. System Architecture

In this section we first describe the system that we model for our experiments, and later we propose hardware support for efficient implementation of shared data structures in our system of study.

A. *System Organization*

Figure 2 shows our system of study, which consists of a host processor connected to multiple memory modules. Each memory module contains one or more memory dies and computation capabilities on a separate logic die connected through 3D stacking with the memory dies. In such a system, memory intensive computations can be offloaded to the PIM logic. We assume inter-PIM links that allow each PIM to access any memory in the system, although it is always lower performance and higher energy to access remote memory. Our study system architecture supports a shared, unified virtual memory address space among the host and PIM devices. Further, caches are kept coherent between the host and each PIM device as well as among the PIM devices.

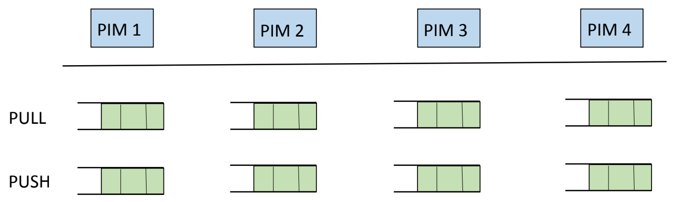


Figure 3. Queuing mechanisms used in our system to implement the graph algorithms and the work migration mechanisms.

Our study system implements a CPU as our in-memory processor in each of the memory stacks, which enables the execution of general purpose programs. However, our evaluation methodology supports exploring the impact of PIM processors with greater degrees of parallelism. For our experiments, we also vary the number of PIM devices from 4 to 16. For our studies we don't use the host for the computation, all the computation are performed and evaluated in the in-memory processors. The host would be in charge of performing the graph partioning and launching of the PIM computation, but we do not account for that overhead.

B. *Queuing Framework*

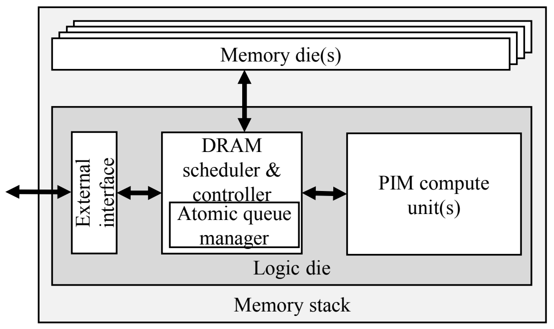
Our queuing framework consists of two queues per PIM device. One queue (PULL) is used to read the active vertices (vertices that require processing in the current iteration of the algorithm) and the other queue (PUSH) is used to keep track of the vertices that become active while processing the current vertices; those neighboring vertices will be processed in the next iteration of the algorithm. Figure 3 shows the queuing framework. In work migration, when a vertex is processed, its neighbors are enqueued to the PIM device where their data is located, based on the partitioning of the graph data structure. In work migration vertices will be processed locally. While for the *baseline*, when a vertex is processed, its neighbors are enqueued in a round robin fashion to the PIM devices in the system. Some vertices will be processed locally and some remotely. For both approaches some queue operations will be performed to local memory (local queue) and others to remote memory (remote queue). This framework is based on an implementation of BFS for distributed systems [7].

Our queues are an array-based implementation. As there might be multiple threads simultaneously operating on the same queue, we use atomics to guarantee isolation and correctness. Currently we do not impose any restrictions on the size of the queues.

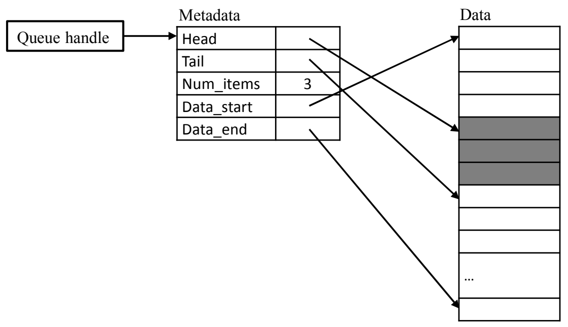
C. *Hardware Support for Efficient Shared Data Structures*

As the queues in our framework are shared and can be accessed by all the threads in the system, atomic instructions are used to update the index that threads need to use to access the array-based queues. The use of atomics results in performance overheads and increased contention.

We propose leveraging the proximity of PIM to memory to implement hardware support that can serialize the queue operations and guarantee atomicity without the need for explicit software atomics, thereby enhancing the performance of shared queues. Our proposed work migration scheme benefits directly from these hardware mechanisms while other applications and execution models may also benefit from having more efficient shared data structures.



(a)



(b)

Figure 4. (a) the hardware support for the atomic queues implemented in the memory controller and (b) the metadata for a queue with three data elements in it.

Hardware support for such *atomic queues* is implemented in the PIM dies, at the DRAM controllers as shown in Figure 4(a). Each atomic queue does not span multiple memory modules or multiple memory channels (i.e., all accesses to an atomic queue go through the same DRAM controller). An atomic queue is allocated via a special system call that allocates memory for it and sets up the necessary metadata in memory, returning a pointer to the metadata structure.

Figure 4(b) shows the metadata structure which contains information regarding the head and tail of the queue, the number of elements in it and the memory space reserved for the data. Atomic queues are word-aligned and each enqueue or dequeue atomically inserts or removes a single-word entry to or from the queue. This capability is sufficient to enqueue vertex IDs or identifiers used in task queues[2]. An enqueue operation is issued as a special store operation to the address of the queue (i.e., the address of the metadata block returned at queue allocation) and the vertex ID as the data to enqueue. A dequeue operation is issued as a load instruction to the queue address. Specialized hardware at the DRAM controllers is responsible for updating the queue metadata as necessary as well as atomically performing the requested queue operation. Atomic queue metadata is allocated in a part of the addresses space that cannot be cached by the PIM or host processors to simplify the implementation. Note that this does not lead to performance overheads as the queue metadata are not manipulated by the processor after allocation. Further, to optimize the performance, the queue management hardware may contain a cache exclusively for storing recently used queue metadata (as all accesses to the queue must go through this hardware, its cache has no coherence requirements).

Once a queue operation starts, all other operations to that queue are stalled at the serialization point (i.e., DRAM controller) until the currently executing operation completes. Queue accesses are only serialized with respect to other accesses to the same queue. Normal load/store operations as well as operations to other queues can proceed in parallel.

IV. Evaluation Methodology

In this section we present the methodology for evaluating work migration against the *baseline*, where data is accessed remotely when needed and no work migration is done. First, we implement a queuing framework to support a distributed data-driven implementation of breadth-first search (BFS), single source shortest path (SSSP) and betweenness centrality (BC) in our system of study. Second, we look at possible ways to improve the queue efficiency by taking advantage of PIM. Third, we develop a timing model to compare work migration against the *baseline*. Using both the queuing framework and the timing model, we compare the performance of work migration vs. the *baseline*.

The main purpose of using this framework instead of a cycle-level simulator is to easily and quickly perform a rapid exploration of a large parameter space consisting of various hardware systems, algorithm characteristics, and data sets.

A. Timing Model

To compare the behavior of work migration vs. the *baseline*, we develop the high level timing model shown in Figure 5. Figure 5(a) shows the *baseline*, where both the graph and the computation are partitioned among all the memory modules but the neighbors of a vertex that is being processed are enqueued in a round robin fashion to the queues of the various PIM devices; some vertices are processed locally and others are processed remotely (performing remote memory accesses to the vertex data that is located on a remote PIM stack). Figure 5(b) represents the work migration scenario, where both the graph and the computation are also partitioned among all the PIM stacks, but vertices are processed locally; some queue operations are performed to local queues and others to remote queues. When a vertex is processed their neighbors are enqueued to the PIM queue where their data is located, which may be the local queue or a queue located on a remote memory module.

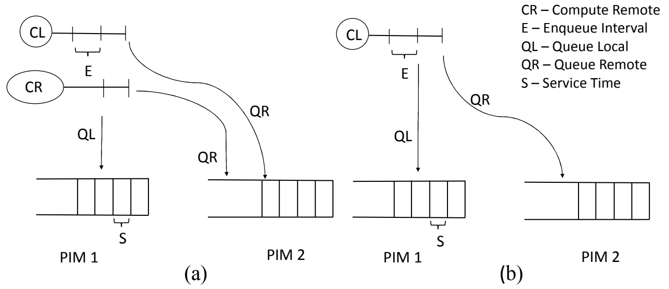


Figure 5. Timing model: (a) represents the *baseline* and (b) represents the work migration scenario.

[2] This functionality is sufficient to enqueue any general data structure by allocating and populating it outside the queue and atomically enqueuing a pointer to it.

The parameters shown in Figure 5 represent the number of cycles for each of the key operations in our system. CL and CR model computation time while the remaining parameters model various aspects of queueing and memory accesses. We use this parameterized model to study different application characteristics and a variety of hardware systems.

CL denotes the number of cycles spent doing computation on a vertex that is local to the PIM. Varying this parameter allows us to model applications with different amounts of computation per vertex. BFS is an example of an algorithm that performs no computation per vertex and has a low CL value. CR is the number of cycles spent doing computation on a vertex that is remote to the PIM in the *baseline* model. CR is not an independent parameter in our model and is computed as $CR = CL + X \cdot C$, which models the remote computation as consisting of the same computation as a local vertex plus some number of remote memory accesses. The total remote memory access overhead during the vertex's computation is expressed as the product of the remote memory access communication overhead C and the number of serialized remote memory accesses X (sets of independent memory accesses are counted as a single access in X as those can be issued in parallel to overlap their latencies). Increasing the value of C allows us to model systems in which it takes longer to perform a remote memory access[3], and varying X models applications with a different ratio of computation to remote memory accesses.

Parameter QL is the number of cycles it takes for a vertex to arrive at a queue that is in the local memory of the PIM device performing the enqueue operation. This parameter is based on the latency of accessing local main memory. QR is the number of cycles that it takes a vertex to reach a remote queue; we define $QR = QL + C$. The service time of a queue is the time it takes to enqueue or dequeue a vertex and we represent it as S. This parameter models potential overheads of using the queues. A naïve queue implementation, such as one that requires acquiring a global lock will have a high value of S while more efficient queue implementations will have lower values. E is the interval between back-to-back enqueues of vertices issued by a single PIM device. A low value of E corresponds to more frequent issue of enqueue operations and models PIM processors with higher operating frequencies or higher degrees of parallel execution. Note that lowering the value of E also increases the degree of contention seen by the queues. Even though multiple processors can issue an enqueue operation to the same queue, these operations are serialized at the queue and only one element can be accepted by the queue at each time interval (as defined by the parameter S); this way we correctly account for contention at the queue.

We initially assume that atomicity in queue manipulations is achieved via an atomic modification of the queue pointers (i.e., atomic increment or decrement of queue pointers). We model the cost of these atomic operations as equivalent to the cost of a load to main memory that is serialized with the queue access. Some atomic operations will be performed to the local memory and some others to a remote memory, depending on what PIM the queue that is being accessed is located. We assume that reaching to any remote PIM constitutes the same amount of communication overhead (C).

[3] Although our framework can be extended to model different communication delays (i.e., different values of C) for each pair of communicating PIM devices, our results here use a constant remote access latency for all PIM devices. We expect this to yield reasonable results on average for applications with irregular neighbor remote memory accesses, such as the graph algorithms considered here.

B. Experiment Setup

With the queuing framework and the timing model we first study the benefits of using PIM to serialize the access to the queues and later we compare the performance of work migration vs. the *baseline* for the selected graph traversal algorithms. We study the effect of load imbalance and compare the energy of the memory accesses for the different approaches. We study a set of algorithms that are core primitives for graph processing and are fundamental for many other more complex applications.

For our experiments we choose parameters within a range that is reasonable for our study system. We choose the latency to access local memory (QL) to be 64 cycles. We vary the latency of accessing remote memory (QR) from 64 to 640 cycles, sufficient to cover a broad range of design options from multiple memory modules within a single package to various interconnection network options among memory modules in separate packages. We choose parameter $E = 1$, as we model processors that can enqueue one element per cycle. Parameter $S = 1$, as the queues can accept one element per cycle.

We measure the number of local and remote memory accesses for each approach and compare the total relative memory energy considering the relative energy of a remote memory access with respect to a local memory access. Similar to our performance modeling, we consider a range of remote memory access energies that span a broad range of possible implementations. We compare the total memory energy for the cases where the energy of a remote memory access is 1x, 5x and 10x that of a local memory access.

C. Graph Algorithms

Breadth First Search (BFS) is a fundamental graph traversal algorithm. It traverses all the vertices in a graph that are reachable from the source vertex. Initially all vertices are set as not visited and the traversal starts at the source. First, we visit the neighbors of the source, which are at a distance 1 from the source. Then we visit all the neighbors of the vertices at distance 1 that have not been visited yet. We keep repeating the same operation until all vertices reachable from the source vertex have been visited, visiting all vertices at distance k from the source before visiting any vertex at distance k+1.

Single Source Shortest Path (SSSP) is a graph analytics application that computes the shortest path of each node from a designated source node in a graph with non-negative edge weights by using a demand-driven modification of the Bellman-Ford algorithm [8]. Each node maintains an estimate of its shortest distance from the source called *dist*. Initially, this value is infinity for all nodes except for the source, whose distance is 0. The algorithm proceeds by iteratively updating distance estimates starting from the source and maintaining a worklist of nodes whose distances have changed and thus may cause other distances to be updated.

TABLE I. MAIN CHARACTERISTICS OF THE GRAPHS

Name	Description	Type	# Vertices	# Edges	Degree	Distribution	Diameter
Road	Roads of Texas	undirected	1,379,917	1,921,660	1.4	normal	1054
Amazon	Co-purchased products	undirected	403,394	3,387,388	8.4	normal	35
Pokec	Social network	directed	1,632,803	30,622,564	18.7	power	14

Betweenness centrality (BC) is a social analysis application, which is a special case of graph analytics. It is used to measure the influence a vertex has on a graph. A vertex's BC score is related to the fraction of shortest paths between all vertices that pass through the vertex. In a graph with *n* vertices, *n* breadth-first search graph traversals are performed one from each vertex in the graph, and augment each traversal to compute the number of shortest paths passing through each vertex. The algorithm computes BC in two stages. First, the distance and shortest path counts from the source vertex *s* to every other vertex are determined. Second, the vertices are revisited starting with the farthest vertex from *s* first, and dependencies are accumulated.

Our implementation uses the algorithm proposed by Madduri *et al.* [9]. They propose a lock-free parallel algorithm for BC that achieves better spatial locality by tracking the successors of each vertex instead of the predecessors as traditionally proposed by Brandes [10].

D. Workloads

We use three real-world graphs from SNAP [5], described below, that present different characteristics. TABLE I summarizes their main characteristics. The diameter of a graph is the longest shortest path in the graph. There are two types of distributions: *normal,* in which most vertices have approximately the same number of edges, and *power-law,* in which some vertices have a very large number of edges while most vertices have just a few. We use the compressed sparse row (CSR) representation to store the structure of the graphs. In this representation vertices and edges are stored in different arrays and the vertex array stores offsets into the edge array, providing the offset of the first outgoing edge of each vertex.

Texas road network graph: This graph shows intersections and end points as vertices and the roads connecting these intersections or endpoints as edges.

Amazon co-purchasing graph: This graph represents products that are often bought together on Amazon. Vertices are products and edges show which ones are bought together.

Pokec social network graph: Pokec is the most popular on-line social network in Slovakia. It connects more than 1.6 million people. The vertices are the members and the edges represent their friendship relationships.

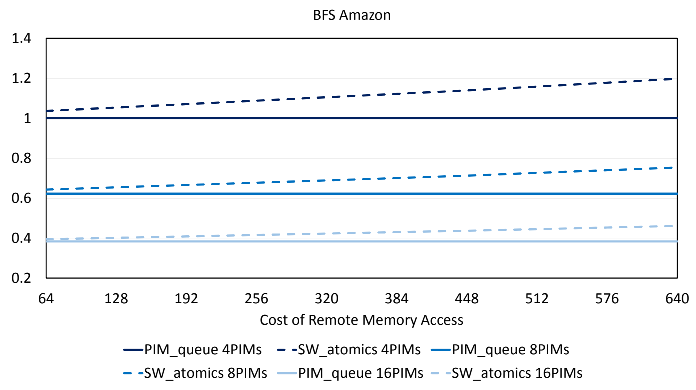
V. EVALUATION RESULTS

In this section we compare the performance of work migration vs. the *baseline* case by applying the timing model to the graph traversal algorithms. We first study the performance impact of using PIM to serialize the queue operations without the need for application-level atomics. Later we study the performance of work migration vs. the *baseline* and we propose modifications to improve the performance of work migration. Finally we study load imbalance in our graph traversal algorithms and mechanisms to improve load balancing while maintaining locality.

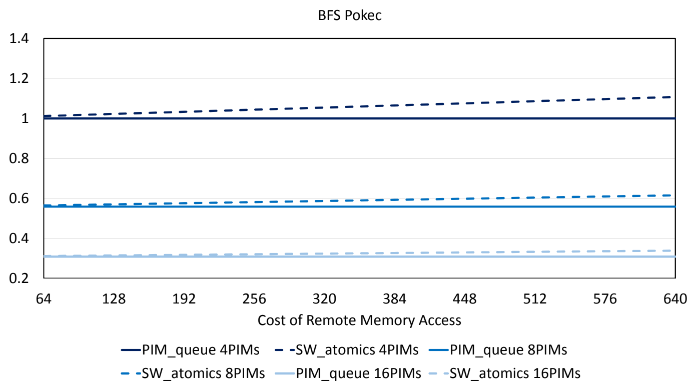
A. Hardware Support for Efficient Shared Queues

In this section we study the effect of our proposed mechanisms to take advantage of the proximity of PIM to the memory to serialize the queue operations guaranteeing atomicity without the use of explicit software atomics.

Figure 6 shows the performance difference of using explicit software atomics (sw_atomics) to serialize the queue operations vs. using PIM (pim_queues). We present results for BFS for the Amazon and Pokec graphs, when doing migration for 4, 8 and 16 PIMs. For these experiments we vary the cost of accessing remote memory *QR* from 64 to 640 cycles, which is 1x to 10x the cost of accessing local memory *QL*. The results are normalized to the performance of pim_queue for 4 PIMs. We observe that for all cases our proposed mechanism performs better. The Amazon graph demonstrates a larger relative benefit, up to 16.7% improvement for 4PIMs when *QR*=640, due to the graph having a larger diameter. The



(a)



(b)

Figure 6. Performance of using explicit software atomics to update the queues vs. using our proposed mechanisms that uses PIM to serialize the queue operations. We vary the latency to a remote PIM from 64 to 640 cycles (*QR*), which is 1x to 10x the latency to a local PIM (*QL*). Results are normalized.

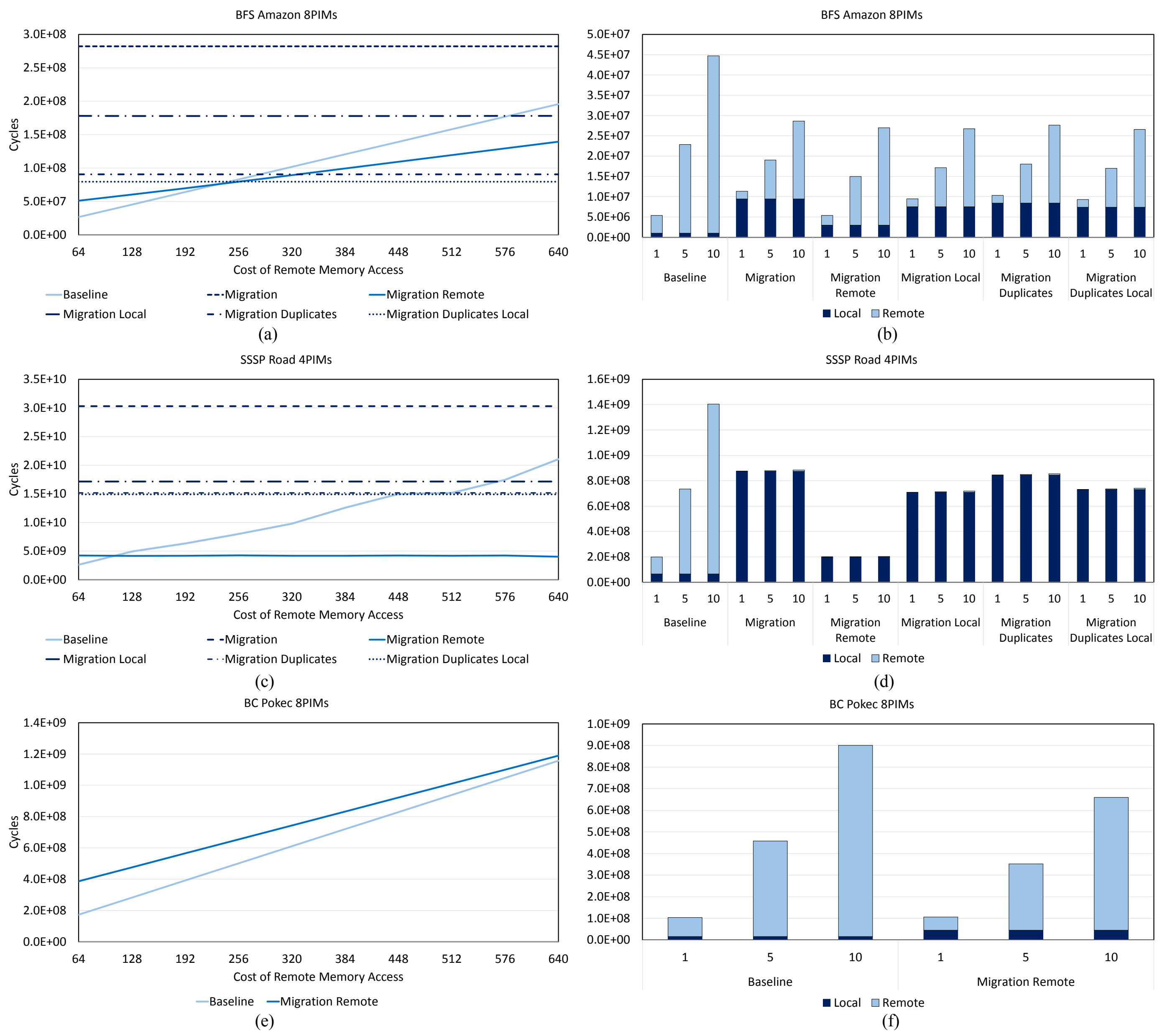


Figure 7. The first column shows the performance of BFS for the Amazon graph (8PIMs), SSSP with Texas road network (4PIMs) and BC for the Pokec social network (8PIMs). We vary the latency to a remote PIM from 64 to 640 cycles (*QR*), which is 1x to 10x the latency to a local PIM (*QL*). The second column shows the energy of the memory accesses when the energy of a remote memory access is 1x, 5x and 10x that of a local memory access.

maximum improvement that we see for Pokec is 9.7% for 4PIMs when *QR*=640. A clear advantage of our hardware support is that when avoiding the use of explicit software atomics, the performance of BFS is not dependent on the latency to access remote memory in a remote PIM. This is because we can eliminate the remote memory accesses from the critical path, as will be explained in the next section. For the rest of our results we use this hardware support.

B. *Study of Work Migration*

In this section we study the performance of work migration compared to the *baseline*. We also compare the number of local and remote memory accesses performed by each of the policies.

We consider two different alternatives to implement BFS and SSSP. When processing a vertex these algorithms only require data related to that vertex or data from the immediate predecessor vertex (SSSP requires the distance of the predecessor vertex to the root in order to compute the distance of the current vertex to the root). First, we avoid the need to perform remote memory accesses in the critical path by enqueuing all the neighbors of the vertex that is being processed to the corresponding PIM. Checking whether those vertices need to be processed or not is done in the next iteration of the algorithm requiring only local memory accesses. However, this may result in redundant enque and deque operations for vertices that need not be processed. For SSSP the distance to the predecessor can also be enqueued with the vertex, we do this by merging both words together into a longer word that can be enqueued and dequeued atomically. The second alternative first checks whether the neighboring vertices of the vertex that is currently being processed need to be processed or not before enqueuing them. This ensures that only the vertices that need to be processed are enqueued, but requires performing remote memory accesses in the cases where the neighbors are located in different PIM stacks. We call these two implementations *migration* and *migration remote*:

- **Migration:** This policy eliminates remote memory accesses from the critical path by enqueuing all the neighbors of the vertex that is being processed to their corresponding PIM. The check is later done locally. This policy results in vertices being enqueued and dequeued that might not need to be processed, but avoids remote memory accesses in the critical path.
- **Migration remote:** This policy performs remote memory accesses to check if the neighboring vertices need to be processed. If a vertex needs to be processed it is enqueued to its PIM. This policy avoids enqueuing and dequeuing extra vertices at the cost of having remote memory accesses in the critical path.

In the case of BC, when a vertex is being processed data from its successors is needed. So memory accesses to the successors are unavoidable and these memory accesses can be local or remote. For BC we use the second implementation alternative.

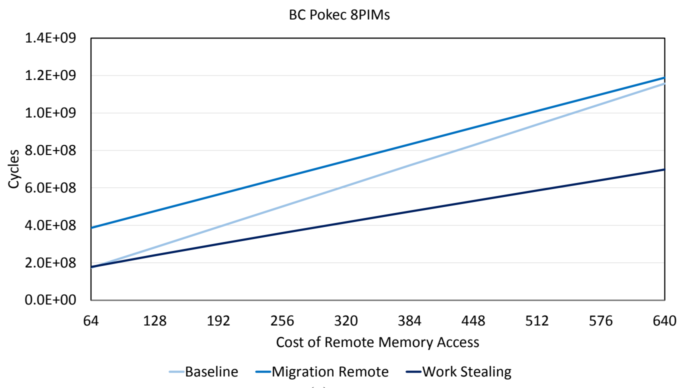
Enqueueing and dequeueing all the neighboring vertices to later check whether they need to be processed can be expensive. The higher degree a graph has, the more expensive *migration* becomes. We propose the following modifications to *migration* in order to reduce the number of vertices that are enqueued and still avoid remote memory accesses in the critical path:

- **Migration local:** Similar to *migration*, but this policy checks the vertices that are local to the PIM to ensure they need to be processed in the next iteration before enqueuing them. Since the vertex is local, the memory accesses to the vertex's data are local. Vertices that are not local are still enqueued without validation.
- **Migration duplicates:** This policy is similar to *migration*, but the queues are implemented as a hash table to eliminate duplicate vertices. Every time an element is enqueued, the hash table is looked up to see if the element is already present, and if it is not the element would be written to the table. This results in extra memory accesses when enqueuing, but they are overlapped with the computation. This table could be implemented in software. Accesses to the hash table would still be serialized by our previously proposed hardware mechanisms introduced in Section III.C.
- **Migration duplicates local:** Combination of *migration local* and *migration duplicates*.

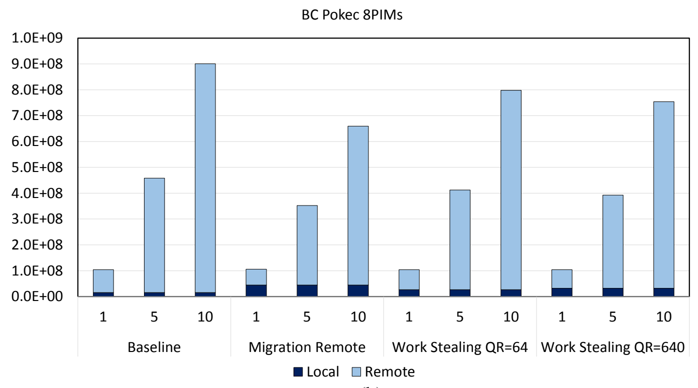
Figure 7 shows the performance and memory accesses of the different migration policies vs. the *baseline* for BFS, SSSP and BC, as we increase the cost of accessing remote memory. We vary the cost of accessing remote memory (*QR*) from 64 to 640 cycles. We compute the total memory access energy as the energy of a remote memory access is 1x, 5x and 10x that of a local memory access. We study the behavior of the Amazon, road and Pokec graphs, when using 4 and 8 PIM devices. We only show a subset of the results due to space constraints. The presented data is sufficiently representative of the rest and can help understand the trade-offs of the different policies. For BFS and SSSP, Figure 7(a) and (c), we can use the *migration* policy where all the neighboring vertices of the vertex being processed are enqueued, and checking whether the vertex has been visited (or whether the distance to the vertex is lower, for SSSP) is done locally. *Migration* results in a large number of local memory accesses and only some enqueue operations are remote. As the remote queue operations are overlapped with the computation, the performance of *migration* is not affected by the increasing cost of accessing remote memory. But the performance of *migration* is worse than that of the *baseline* and *migration remote*. Although the number of remote memory accesses is low for *migration*, the large number of local memory accesses results in high total memory energy, as Figure 7(b) and (d) show.

In Figure 7(a) and (c) we observe that the performance of *migration* improves with the proposed modifications (*migration local, migration duplicates* and *migration duplicates local*) and they are also not affected by the cost of accessing remote memory. On the contrary, the performance of *baseline* worsens as the cost of accessing remote memory increases. At lower *QR* the *baseline* performs better, as it achieves better load balance among PIMs by enqueuing vertices in a round robin way, and the effect of *QR* is lower and less important than the impact of load imbalance. The *baseline* performs multiple remote memory accesses when processing a vertex remotely, to read the graph data structure that is stored in remote PIMs. These accesses are often interdependent resulting in multiple serialized round trip accesses to the remote PIM. For example, the vertex array needs to be accessed first to find the corresponding index to the edge array, which contains the neighbors of a vertex. On the contrary, *migration* and its variants are not sensitive to higher latency to remote PIM stacks as accesses to a remote PIM are performed when a vertex is enqueued to a remote queue, which is off the critical path. These new policies based on *migration* result in fewer local memory accesses than *migration* as shown in Figure 7(b) and (d), lowering the total memory energy.

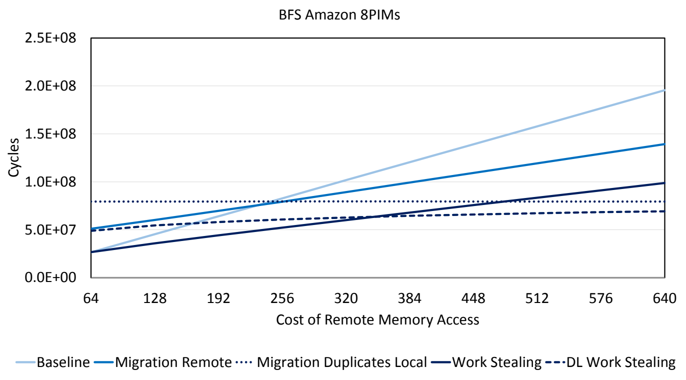
In Figure 7(a) we observe that the performance of *migration remote* worsens with *QR*. This is due to the remote memory accesses that this policy performs to check what vertices to enqueue. For low *QR*, *baseline* performs better than *migration remote* as it is more balanced, but as *QR* increases a low number of remote memory accesses becomes more important. The number of remote memory accesses for *migration remote* is however larger than for the other migration policies; as can be seen in Figure 7(b). This is not the case for SSSP with the Texas road graph; as Figure 7(c) shows. The Texas road network graph presents few edges per vertex, and high locality in its neighbors, meaning that the neighbors of a vertex are likely to be on the same PIM stack as the parent. Therefore, *migration remote* is not affected by *QR* as most of the memory accesses are local. This is why *migration remote*



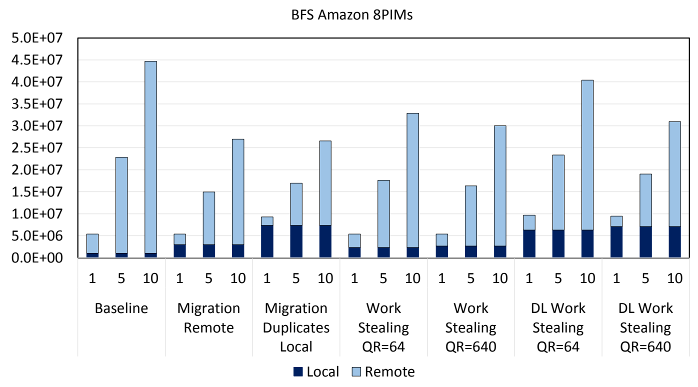
(a)



(b)



(c)



(d)

Figure 8. The first column shows the performance of *work stealing* for BC with the Pokec graph (8PIMs) and *work stealing* and *DL work stealing* for BFS with Amazon graph (8PIMs). We vary the latency to a remote PIM from 64 to 640 cycles (*QR*), which is 1x to 10x the latency to a local PIM (*QL*). The second column shows the energy of the memory accesses when the energy of a remote memory access is 1x, 5x and 10x the energy of a local memory access.

results in better performance than any of the other migration policies. As *migration remote* does check and, if necessary, updates the distance to the neighboring vertices before enqueuing the vertex, only the vertices that need to be processed are enqueued, considerably reducing the number of memory accesses.

In Figure 7(e) for BC, we use *migration remote*; we need to perform memory access and update a vertex's data before it is enqueued. This is the reason why *migration remote*'s performance increases with the cost of accessing remote memory. In this case *migration remote*'s performance is worse than the performance of the *baseline* due to the load imbalance. The degree of imbalance is input-dependent, but we observe it in all the studied graphs. As the number of PIMs increases, load imbalance goes up. Figure 7(f) shows the memory accesses performed by both policies. We observe that the number of remote memory accesses is larger for the *baseline* than for *migration remote*, the *baseline* results in more energy spent in memory accesses. In the next section we explore the effect of work stealing to improve the performance of *migration remote* while still trying to maintain locality.

C. Load Balancing and Locality

As shown before for the BC case, the *baseline* performs better than *migration remote* although the *baseline* results in more remote memory accesses. For BFS and SSSP, the *baseline* performs better than the rest of the policies for low *QR*. For all cases this is due to the load imbalance in the graph algorithms and the graphs.

In this section we study *work stealing* implemented on top of *migration remote* and *DL work stealing* implemented on top of *migration duplicates local* to address the load imbalance problem. We implement a traditional work stealing mechanism, where threads first execute the vertices that are in their local queue and once their local queue is empty, if there is still work to be completed in the system, they steal from the queue with the most elements. When the cost of accessing remote memory and local memory is close, work stealing reduces load imbalance and the number of vertices processed by each PIM is similar. As the cost of accessing remote memory increases, the cost of processing vertices remotely goes up, and fewer vertices will be processed remotely, naturally reducing the degree of work stealing and decreasing the number of remote memory accesses. Work stealing is more efficient as the cost of accessing remote memory goes up.

Figure 8 shows the performance of *work stealing* for BC using the Pokec graph and both *work stealing* and *DL work stealing* for BFS using the Amazon graph. Figure 8(a) shows the results for BC, and *work stealing* always performs better than the *baseline*. This is because *work stealing* first executes as many vertices locally as possible and then it executes vertices remotely (stealing from other queues) to balance the load of the different threads. Figure 8(c) shows the results for BFS, *work stealing* is the best performing policy for lower *QR*. For higher *QR, DL work stealing* performs better as it is not affected by *QR*. Figure 8(b) and (d) show the memory accesses performed by the different policies and the total memory access energy as the energy of a remote memory access is 1x, 5x and 10x that of a local memory access. We see that *work stealing* performs more remote memory accesses and less local memory accesses than *migration remote*, but less remote memory accesses than the *baseline*. In Figure 8(b) we see that *DL work stealing* presents more local memory accesses than *work stealing*, but less than *migration duplicates local*. Figure 8(b) and (d) show that for both work migration policies the number of remote memory accesses decreases as the cost of accessing remote memory goes up resulting in lower total memory energy with *QR*.

We observe that there is a trade-off between performance and energy; both *work stealing* and *DL work stealing* provide better performance for higher energy than other policies such as *migration remote or migration duplicates local*, since in order to balance the load it needs to steal vertices and execute them remotely, but this difference decreases as *QR* increases.

VI. Related Work

The common approach taken in distributed systems to process graphs is to accumulate all the edges corresponding to non-local vertices and send them to the owner processor at the end of each iteration. There is thus an all-to-all communication step at the end of each frontier expansion. Inter-processor communication is considered a significant communication overhead, the cost of this step will depend on the particular network topology and the partitioning of the graph [7]. Our proposed fine-grained work migration overlaps the communication with the computation, improving performance.

Hardware approaches have also been proposed to improve the performance of data-driven graph traversal implementations by reducing the contention on the shared data structures [11], but these often require dedicated hardware to hold the queue's data or metadata or both. As a result, these schemes incur high overheads to provision these dedicated structures and scalability is still limited by the hardware availability. Dedicated hardware also complicates context switching as the dedicated hardware must either be included in the context state or continue to be occupied by inactive contexts. Dedicated accelerators for parallel graph processing have also been proposed in the context of PIM [12]. We use general purpose processors, which are able to execute a variety of applications not only graph algorithms.

Lock-free structures have also been proposed based on the computation of prefix-sums of multiple, parallel agents accessing the queue (so that each knows which location within the queue to access in parallel) [13]. However, these are difficult to orchestrate over loosely-coupled execution engines, as is the case with multiple processors that may not reside on the same chip. Our proposal does not require coordination between the different PIMs to compute the prefix-sums.

VII. Conclusion

This work shows that the irregular memory access patterns present in graph algorithms make it challenging to implement these algorithms obtaining high data locality on systems with multiple PIM devices. We propose fine-grained work migration to maximize data locality in PIM-based systems.

This work proposes a framework to explore a large space of graph application behaviors and system architectures. We provide insight regarding what characteristics of the graph applications and system parameters result in efficient work migration to take advantage of data locality.

Hardware support for efficient queue implementations can considerably increase the performance of our work migration mechanisms. We present a technique for such hardware support that leverages the DRAM controller in PIM to enforce serialization of operations to the same queue, eliminating the overheads of atomics. Our method also does not require dedicated hardware storage, allowing it to scale to arbitrary numbers of queues and arbitrarily large queues, limited only by the DRAM capacity of the memory modules.

To conclude we can say that fine-grain task migration results in lower number of remote memory accesses in graph algorithms compared to the baseline, but naïve migration schemes increase the number of local memory accesses which can hurt both performance and energy. We proposed optimizations to fine-grain task migration to reduce the number of vertices that are redundantly enqueued/dequeued, mitigating the increase in local memory accesses and still keeping remote memory accesses off the application's critical path. We also evaluated work stealing mechanisms to further improve the performance of fine-grain task migration by reducing the load imbalance at the cost of some loss of data locality in the memory accesses.

References

[1] D. A. Patterson, "Latency Lags Bandwith," *Commun. ACM,* 2004.

[2] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu and M. Ignatowski, "TOP-PIM: Throughput-oriented Programmable Processing in Memory," in *HPDC*, 2014.

[3] T. von Eicken, D. E. Culler, S. Copen Goldstein and K. Erik Schauser, "Active Messages: A Mechanism for Integrated Communication and Computation," in *ISCA*, 1992.

[4] D. Chang, B. Gyungsu, K. Hoyoung, A. Minwook, R. Soojung, N. Kim and M. Schulte, "Reevaluating the latency claims of 3D stacked memories," in *ASP-DAC*, 2013.

[5] J. Leskovec and A. Krevl, "Stanford Network Analysis Project (SNAP)," 2014. [Online]. Available: http://snap.stanford.edu/data.

[6] R. Nasre, M. Burtscher and K. Pingali, "Data-Driven Versus Topology-driven Irregular Computations on GPUs," in *IPDPS*, 2013.

[7] A. Buluc and K. Madduri, "Parallel Breadth-first Search on Distributed Memory Systems," in *SC*, 2011.

[8] "https://en.wikipedia.org/wiki/Bellman%E2%80%93Ford_algorithm," [Online].

[9] K. Madduri, D. Ediger, K. Jiang, D. A. Bader and D. Chavarria-Miranda, "A faster parallel algorithm and efficient multithreaded implementations for evaluating betweenness centrality on massive datasets," in *IPDPS*, 2009.

[10] U. Brandes, "A Faster Algorithm for Betweenness Centrality," in *Journal of Mathematical Sociology*, 2001.

[11] J. Y. Kim and C. Batten, "Accelerating Irregular Algorithms on GPGPUs Using Fine-Grain Hardware Worklists," in *MICRO*, 2014.

[12] J. Ahn, S. Hong, S. Yoo, O. Mutlu and K. Choi, "A Scalable Processing-in-memory Accelerator for Parallel Graph Processing," in *ISCA*, 2015.

[13] R. Nasre, M. Burtscher and K. Pingali, "Atomic-free Irregular Computations on GPUs," in *GPGPU-6 at ASPLOS*, 2013.