Designing Algorithms for the EMU Migrating-threads-based Architecture

Mehmet E Belviranli, Seyong Lee, Jeffrey S Vetter
Oak Ridge National Laboratory
belviranlime@ornl.gov, lees2@ornl.gov, vetter@ornl.gov

***Abstract*—The decades-old memory bottleneck problem for data-intensive applications is getting worse as the processor core counts continue to increase. Workloads with sparse memory access characteristics only achieve a fraction of a system's total memory bandwidth. EMU architecture provides a radical approach to the issue by migrating the computational threads to the location where the data resides. The system enables access to a large PGAS-type memory for hundreds of nodes via a Cilk-based multi-threaded execution scheme.**

EMU architecture brings brand new challenges in application design and development. Data distribution and thread creation strategies play a crucial role in achieving optimal performance in the EMU platform. In this work, we identify several design considerations that need to be taken care of while developing applications for the new architecture and we evaluate their performance effects on the EMU-chick hardware. We also present a modified BFS algorithm for the EMU system and give experimental results for its execution on the platform.

I. Introduction

The demise of Moore's law has led the evolution of computing towards multi-core CPUs, many-core chips, general purpose GPUs, FPGAs, and other domain-specific accelerators. As the number of processing units sharing the same memory bus increased, these architectures employed deep-pipelines, multi-level caches, and wide memory channels to get around the so-called 'Von Neuman bottle-neck'. Such systems favored compute-intensive workloads with predictable access patterns and high locality to exploit their advertised throughputs.

On the other hand, applications that fall into 'data-intensive' category, such as sparse matrix-based BLAS operations and graph analytics, only managed to achieve a tiny fraction of the *Rmax* (i.e., maximal achieved performance). These types of workloads usually exhibit 'weak-locality' where frequent irregular accesses are scattered across a large memory range; therefore, their compute efficiencies are majorly limited by data movement capabilities of the underlying architectures. As the computation scales over a slower fabric, the dramatic increase in latency forces the need to further focus on 'data-movement' and reduce memory-imposed transfer and communication bottlenecks.

The decades-old memory-bottleneck problem has been handled by the industry and research communities in many dimensions. Among the most notable of these, in SMP-based systems, a temporary remedy is being sought by investing into higher-throughput 3D-stacked memory systems such as high-bandwidth memories (HBM) [16]. The idea of processing in memory (i.e., PIM) coupled with HBM [18][11][3] has recently shown promising solutions for shared memory systems, but they are limited in scalability due to being single-node oriented. On the other hand, partitioned global address space (PGAS) based languages [9], [5] provided an alternative and more-scalable solution by tightly coupling threads to memory locations. However, the limited scope of application classes that can utilize such languages and the difficultly in designing algorithms around the concept of 'stationary-threads' have not provided a long-term solution to the problem in question.

EMU[7] is a new architecture that attempts to address the weak-locality problem in data-intensive applications by migrating the threads to location where the data resides. The memory is accessed via a global address space, and since the data does not move, the need for cache coherency is eliminated. The programming model is based on Cilk [2], and thread migration is automatically performed by the device runtime as the memory accesses occur.

In this study, we discuss the architectural considerations that need to be taken into account while designing applications for EMU systems. We also present a modification of level-synchronous breadth-first-search (BFS) for EMU. Our paper makes the following contributions:

- We enumerate, discuss, and compare several programming considerations for the EMU architecture.
- We present an exploratory performance analysis on the effects of these considerations on the EMU-chick hardware.
- We design a level-synchronous BFS optimized for the EMU platform.
- We present a preliminary evaluation of the proposed BFS implementation on the EMU-chick hardware.

II. EMU Architecture

EMU is a novel architecture that provides scalable access to a common partitioned global address space (PGAS) through a simple programming interface. The hardware is hierarchically organized as nodes, nodelets, and gossamer cores from top to bottom, and each nodelet owns a partition of the globally addressable memory. Different from existing PGAS and active messaging based solutions, a thread in EMU migrates to the memory location of the operand that the current instruction operates on. The rest of this section summarizes the hardware and programming model for the EMU system. Further details on the EMU platform can be found in [7].

A. Hardware

The specific hardware discussed in this study is the 8-node EMU-chick system [17], and it is implemented using one FPGA per node in a single chassis. The architecture, on the other hand, is designed to scale up to hundreds of nodes that can

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

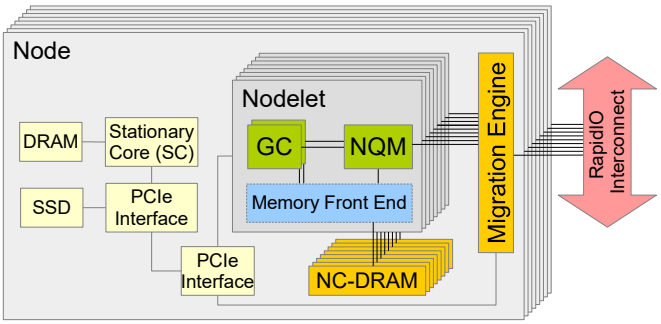

Fig. 1: Emu-chick hardware overview.

fit inside a single rack. Figure 1 shows an overview of the components that the current hardware is composed of, and the Table I gives the specifications of the EMU-chick product.

Each node incorporates eight nodelets, an array of DRAMs, a migration engine, PCI-Express interfaces, and a stationary processor (SP), accompanied with an SSD. A nodelet contains two Gosamer cores (GC), each of which supports 64 concurrent in-order, single-issue hardware threads. Nodelet Queue Manager (NQM) moves packets between GCs, migration engine (ME), and Memory front-end (MFE). Incoming thread packets (i.e., execution requests) are controlled by NQM, and GCs request a thread from NQM when they become ready-to-execute. ME acts as a crossbar between NQM and the six serial RapidIO[6] interfaces that connect the node to other nodes.

Memory organization is the centerpiece of the EMU design. Each node has a 64-byte channel DRAM, divided into eight 8-byte narrow-channel-DRAMs (NC-DRAM). Because EMU does not employ shared caches or coherency, using higher number of memory units with narrower channels allows higher access rates to be achieved. MFE serves the memory transactions initiated by local GCs, and it also handles the atomic operations requested by remote GCs. SPs are used to run node-level service tasks, such as running an O/S and injecting the threads migrating to the node into the ME.

B. Programming Model

EMU supports a PGAS model, and the memory is accessible via conventional pointer addressing. The parallelism is expressed via Cilk programming model [2], and the current LLVM-based EMU compiler supports three Cilk keywords: `cilk_spawn`, `cilk_sync`, and `cilk_for`.

`cilk_spawn` is used to create a new thread via a non-blocking call. Thread contexts in EMU are typically around 10-20 64-bit words, and they are packed in the newly spawned child threads. The context is duplicated in full so that the children can further spawn threads independently, however, with lower priorities than their parent. The thread migration occurs without programmer intervention. The node boundaries are invisible to the application and the entire system is seen as a collection of nodelets.

A spawn is considered either a local or remote spawn, depending on the following two conditions:

- **Remote spawn:** If a memory address is given in the parameters of the spawned function, the compiler inserts proper instructions so that the parent thread first migrates into the node/nodelet that hosts the memory at the given address, and then, the child thread is spawned at the remote location.
- **Local spawn:** If no address can be extracted within the spawn parameters, the child thread is spawned locally, and migration is not performed until the child issues its first memory access.

Remote spawns are performance-effective if the parent thread needs to launch many threads that will access consecutive locations on the target nodelet. Local spawns are preferred if the number of child threads that are expected to operate on nearby remote memory locations is not high.

`cilk_sync` causes the calling (i.e., parent) thread to wait for all spawned children. There are implicit syncs at the end of each code block. `cilk_for` spawns a thread for each index of the loop, and currently it is implemented in a naive way that causes performance issues. The details of parallelizing `for` loops efficiently on the EMU architecture is explained and experimented in the upcoming sections.

EMU API also provides intrinsic functions to perform atomic and remote operations. These operations allow threads to issue remote writes without migrating. These writes are performed by the memory front-end of the corresponding nodelet/node that hosts the address in question. Some atomic operations return the result of the operation whereas a remote update only returns an ACK to the issuing thread to indicate that the operation is complete. EMU does not provide an interface to perform remote reads, and a thread should always migrate to the nodelet of the memory that is being read.

TABLE I: EMU-Chick hardware specifications

EMU-Chick Specifications	
Nodes	8 total
Nodelets	8 per node
Memory	64GB per nodelet
Compute cores	2 Gossamer cores (GC) per nodelet
Service cores	1 Stationary core (SC) per node
System Interconnect	4-lane Serial RapidIO 2.3 @ 6.25 Gbit

III. Developing algorithms for EMU architecture

In this section, we first raise several design considerations unique to the EMU architecture and discuss their implications on applications. Then, we demonstrate the resulting performance of these designs using scalar vector-add on the EMU-chick hardware.

In general, reducing memory access times is a primary concern for applications running on traditional SMP based systems. However, on EMU, minimizing total number of thread migrations is the most important design goal. Memory allocation patterns, thread spawning strategies, and parallelism granularity are the three factors that affect the total number of migrations, and we will discuss them below.

A. Memory allocation patterns

The way that the buffers are allocated in the memory has a direct implication on the number of thread migrations, hence the performance, for EMU applications. Because the computation is always done on the nodelet where the operands are located, memory should be distributed across the global address space while also keeping the indicies belonging to different data ranges close enough so that a thread can execute an instruction with a minimal number of hops.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

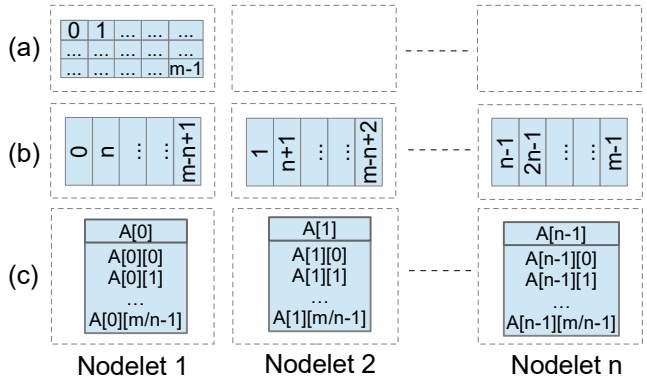

Fig. 2: Different memory allocation patterns for EMU: (a) `mw_localmalloc`, (b) `mw_malloc1dlong` and (c) `mw_malloc2d`

EMU provides three types of memory allocation interfaces: `mw_localmalloc`, `mw_malloc1dlong` and `mw_malloc2d`, as visualized in Figure 2. `mw_localmalloc` allocates a single chunk of memory on the nodelet that is local to the pointer given in the parameters. `mw_malloc1dlong` allocates an array, where each element (i.e., long) is striped across all nodelets. `mw_malloc2d` lets the programmer specify the total number of blocks and the size of each block , and returns a double pointer whose dimensions correspond to the block and element indicies, respectively. Each block is a consecutive chunk created on a single nodelet, and if the number of blocks is bigger than the number of nodelets, then the blocks are distributed across nodelets in a cyclic round-robin fashion.

Considering a 2D allocation pattern, the size of each block and the total number of blocks strongly co-relate to the number of thread migrations. The total migration overhead, M_T, in a simple memory-scan operation can be written as a function of intra-node (M_m) and inter-node (M_n) migration costs, block size ($|B|$) total number of blocks (n_b), nodelets per node (n_m), and total number of nodes(n_n), as follows:

$$M_T = M_m(n_b - \frac{n_b}{n_m}) + M_n\frac{n_b}{n_m} \quad (1)$$

From Equation 1, it is clear that the overhead will get larger as the number of total blocks increases. On the other hand, having larger blocks will cause more threads to be created initially on a less number of nodelets. This will limit both the memory bandwidth and the total computation power and also will increase the idleness of the remaining nodelets during the initial spawn operations.

Figure 3 depicts the outcome of this trade-off relation for a scalar vector addition. This experiment was run on all 8 Nodes and 64 nodelets with total number of blocks varying between 1 and 2^{18} on total vector sizes of 2^{24}, 2^{25}, 2^{26}, and 2^{27} elements. The chart on the top shows the execution times in y-axis for the entire x-axis range and the bottom chart gives more detail for the block-size range where the performance is optimal. The results clearly show that lower number of blocks heavily impact memory-bandwidth utilization and, as block count gets larger, the execution is bottlenecked by inter-nodelet and inter-node migration overheads, regardless of the total vector size.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

B. Thread spawning strategies

Deciding where to spawn threads is also an important consideration in programming the EMU system. As described in the previous section, a local spawn is preferable if the child thread will access a data range that will not be accessed by the consecutively spawned children. On the other hand, in most cases the child threads will expose a weak-local behavior where they will operate on the data that are located close to each other. In such applications, like graph algorithms, a remote spawn, where the parent thread will migrate to the nodelet that a sub-set of children will be most likely to execute on, will result in less migration overhead.

Another factor affecting performance is determining when the child threads should be spawned. For example, if a parent needs to spawn a parallel thread for every iteration of a `for` loop, there are two options:

1) *Flat spawn*: The parent thread sequentially iterates over the `for` loop and spawns children. If remote spawns are utilized, the parent will first migrate to the nodelet where the child thread will be spawned. The total spawning and parent migration overhead can reach up to $O(n)$ in this scenario.
2) *Tree spawn*: The process of creating children in EMU can be parallelized by using a recursive, hierarchal spawn. When compared to the previous approach, the overhead of tree spawn can be reduced to $O(logn)$ at the cost of additional threads spawned for the non-leaf nodes of the recursion tree.

Figure 4 depicts the results of an experiment that shows the effects of using different depths of recursive spawns for scalar vector-add on varying input sizes. The experiment was run on a single node, 8 nodelets with a fixed number of blocks (n_b=64). Spawn tree depth level one, $D = 1$, refers to the *flat spawn* case explained above, and other series correspond to

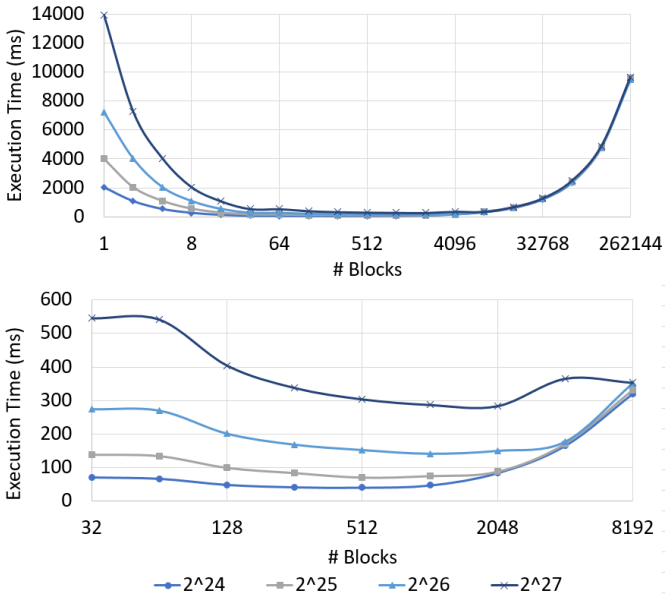

Fig. 3: Execution time for scalar vector add for different data sizes and block sizes:(a) Entire x-axis range shown on the top and (b) a sub-set of x-axis shown on the bottom for a more detailed view.

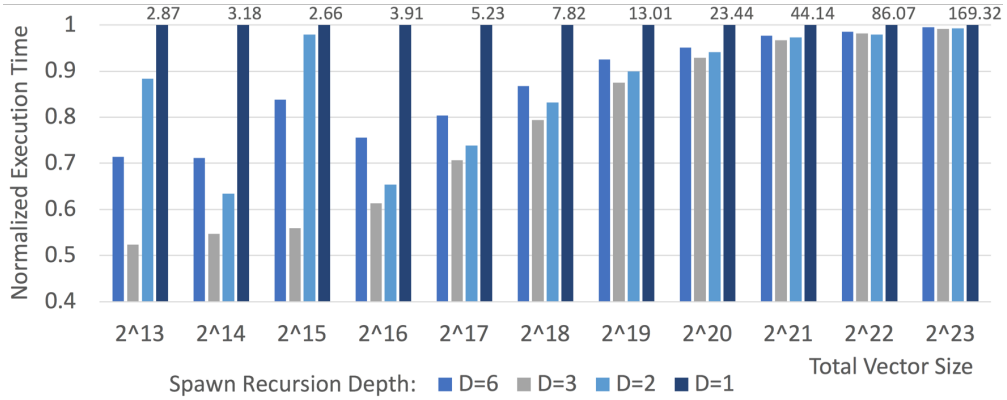

Fig. 4: Effects of spawn recursion depth on performance for varying vector lengths and fixed number of blocks (n_b=64) . The execution times are normalized to D=1 values, and the absolute times in miliseconds for D=1 values are shown in the above corresponding columns.

the *tree spawn* cases. The execution times are normalized to $D = 1$ results. In all cases, a non-flat spawn tree achieves better performance, since the overhead of serial spawn takes considerably longer, especially when the computation (i.e., vector length) is short. Best performance is obtained when the tree depth is three, ($D = 3$), where every thread spawns $2^2 = 4$ children at every recursion level, until a thread for each of the 64 blocks is launched.

C. Parallelism granularity

Each nodelet in EMU-chick can support up to 256 live threads and also can hold context information of up to 500 threads, depending on their total resource usage. In current HW implementation, threads that exceed this level will fail to spawn, and the issuing parent thread will be forced to execute them as a regular function call.

This type of thread scheduling behavior requires the programmer to implement active measures so that total number of threads per nodelet is kept under control. This can be done in two ways:

1) *Workers and queues:* In producer/consumer type of applications, a fixed number of threads can be spawned to process a large chunk of data allocated on a nodelet. This approach would require employing queues for each worker and inter-nodelet load balancing policies.
2) *Thread groups and barriers:* An easy way to ensure that `for` loops do not spawn more than the maximum that the nodelet can execute concurrently is to group the spawn operations into an additional loop and deploy `cilk_sync` after each group. While this method would require less programming effort, global barriers may prevent full utilization of the GCs and memory bandwidth in a nodelet.

The experiments presented in this paper follow the latter approach. In order to control the number of threads spawned on each nodelet, we divide each block into further *chunk*s. We spawn a thread for each chunk and within the chunk, we serially iterate over the indicies so that the total number of threads are limited. Optimal number of threads per nodelet, hence the chunk size per workload, is currently determined by trial and error due to missing resource usage and profiling tools.

Figure 5 shows the effects of using different number of threads per nodelet, while the total number of blocks and block sizes are fixed for a given input size. For this experiment we set the scalar component of the vector add to be calculated in a computation-heavy way so that the migration overhead does not dominate. The results show that any thread count between 32 and 512 behaves optimal. As the number of threads per nodelet gets higher, the execution times increase dramatically, since thread contexts cannot be stored on the nodelet and every newly spawned thread is serially executed. Smaller input sizes are more sensitive to this behavior due to decreased number of memory migrations.

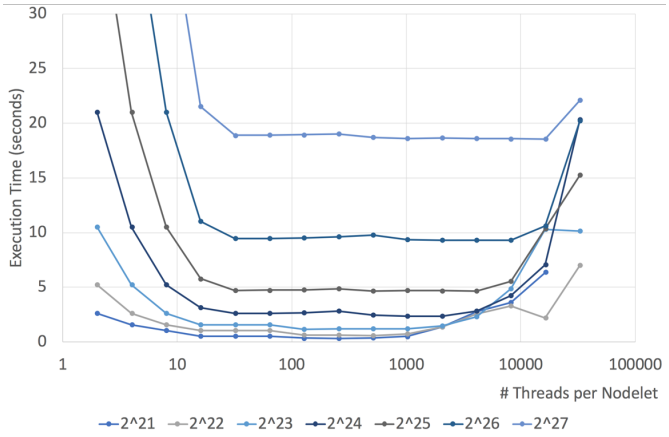

Fig. 5: Effects of using different number of threads per nodelet, with fixed block size and counts, for different input sizes.

IV. Case Study: Level-Synchronous BFS

In this section, we present our design and evaluation for a level-synchronous BFS implementation optimized for the EMU-chick system (BFS-EMU).

A. Graph representation and data staging

In our implementation, we used compressed sparse row representation (CSR)[4] to represent graphs in memory. CSR involves two data structures: node and edge arrays. *Nodes* array holds the number of out-going edges (i.e., neighbors) and the starting index of neighbors on the *edges* array, which stores the target node index for each edge. We found CSR to be an appropriate representation since a node and its neighbors can be processed with fewer number of migrations and local spawns in most cases.

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

I/O operations on EMU are currently binded to the first nodelet of the node that the execution is initiated. For this reason, we perform data staging (i.e., reading the graph from the input files) in two separate phases. To prevent repetitive thread migrations between the first nodelet and others during data staging phase, the graph is first read into the a temporary local buffer on the first nodelet. Then, in-memory *nodes* and *edges* arrays stored in the first nodelet are copied into the distributed memory.

B. Data partitioning

Proper distribution of data across multiple nodelets and nodes is the key to obtaining high performance in the EMU architecture. Level-synchronous BFS employs multiple data structures with varying data sizes to ensure the synchronization between multiple breadths and also to store the resulting `costs` array. Since the `nodes` array is accessed along with these structures, it is essential that all other data corresponding to a node is also placed on the same nodelet.

BFS-EMU relies on five data structures (`nodes`, `masks`, `updatingMasks`, `costs`, and `visited`) with `nNodes` elements each and an `edges` array with `nEdges` elements. Data is partitioned into a total of `nBlocks` blocks using `mw_alloc2d`, and they are distributed across nodelets in a cyclic round-robin fashion.

C. The algorithm

Our modified level-synchronous BFS algorithm for EMU exploits the weak-locality exposed by the CSR format. In level synchronous BFS, each breadth has two phases with synchronization in between: *visit* (line 6) and *update* (line 10). In the first phase, `graph_mask` needs to be checked for every node before deciding for further propagation into the neighbors of that node. BFS-EMU follows a hierarchical launch strategy and spawns a thread for every node block (`visit_node_block`, line 6) so that further node related operations become local to the spawned children.

When a thread is created for a node block (i.e., *block-level-thread*), the starting address for the sequential memory that corresponds to the block is passed as parameter so that a 'remote spawn' can be performed. This address is obtained via `mw_arrayIndex()` function (not shown), which calculates the remote location without requiring the parent thread to migrate to the first nodelet where the double pointer array is defined. *Block-level-thread* further iterates over the nodes in the block (line 15) and spawns leaf level children (i.e., *node-level-thread*) (line 16) that will operate on each node. To prevent creation of excessive number of threads, `cilk_spawn` is invoked for every `chunkSize`.

*Node-level-thread*s are responsible for iterating the edges of a given node, and they further spawn a thread (line 23) for each neighbor (i.e., *leaf-level-thread*s). Since the neighbors of a node are stored in a consecutive portion of `edges` array, a node-level-thread migrates to the nodelet, where the neighbor indices are located, only once. Each *leaf-level-thread* first migrates to the nodelet where the neighbor's corresponding data is located and updates the `costs`, `visited`, and `masks` arrays of the neighbor (lines 26-28).

After the neighbors of the nodes in the current breadth are visited, all threads migrate back to the first nodelet and sync

```
1  void EMU_BFS(){
2   bool done=false;
3   while(done == false){
4    done=true;
5    for( long i = 0; i <nBlocks; i++)
6     cilk_spawn visit_node_block(nodes[i], ...);
7    cilk_sync;
8
9    for( long i = 0; i <nBlocks; i++)
10    cilk_spawn update_node_block(masks[i], &done,
       ...);
11   cilk_sync;
12  }
13 }
14 void visit_node_block(...){
15  for (j = 0; j < elementsInBlock; j+=chunkSize)
16   cilk_spawn visit_node(..);
17  cilk_sync;
18 }
19 void visit_node(nodeIndex, ...){
20  for (long k = j; k < j+chunkSize; k++)
21   if (masks[k]==true)
22    for(each neighbor of nodes[k])
23     cilk_spawn visit_neighbor(...);
24 }
25 void visit_neighbor(neighborIndex, cost, ...){
26  if (!visited[neighborIndex]){
27    costs[neighborIndex]=cost+1;
28    updatingMasks[neighborIndex]=true;
29  }
30 }
31 void update_node_block(...){
32  for (long j = 0; j < elementsInBlock; j+=
      chunkSize)
33   cilk_spawn update_node(...);
34  cilk_sync;
35 }
36 void update_node(...){
37  for (long k = j; k < j+chunkSize; k++)
38   if(updatingMasks[k]==true){
39    masks[k]=true;
40    visited[k]=true;
41    updatingMasks[k]=false;
42    *done=false;
43   }
44 }
```

Listing 1: Pseudo code for BFS-EMU

there. In the second phase of the breadth, `visited` flags of each node are updated based on the `updatingMasks` that are set in the first phase, so that they will be visited in the next breadth, if marked.

D. Evaluation

We evaluate BFS-EMU using different input graph sizes with varying number of average degrees per node (i.e., $deg(v)$). Due to limitations of the current multi-node execution mechanism, we are only able to execute BFS-EMU stably on a single node with eight nodelets. We also run the original Cilk based implementation on a dual-socket 14-core Xeon Haswell CPU for comparison purposes. Figure 6 gives absolute execution times on EMU using all eight nodelets on the top, and CPU execution times on the bottom. We run the algorithm using graph sizes changing from 128K to 16M nodes with average per-node degrees varying from four to seven.

Since FPGA-synthesized Gossamer cores on EMU are inferior to Xeon CPUs in terms of both frequency and ILP characteristics, EMU execution times are considerably slower than CPU times. On the other hand, EMU shows a scaling curve very close to linear (as shown by the dashed lines), whereas

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

CPU performance deviates from the linear curve as the number of nodes increases. Also, EMU handles the increase in average degrees better than the CPU execution. Increasing neighbor counts enables better distribution across nodelets hence ends up in larger performance benefits. In CPU execution, on the other hand, since the major bottleneck is memory bandwidth, higher degree graphs do not perform differently.

We further investigate strong scaling properties of the EMU platform by limiting the total number of threads while keeping the data size same. In EMU, parallelism granularity is implicitly controlled via memory allocations so that the threads will run only on the nodelets/nodes where the data is allocated. Figure 7 shows the results of this experiment where we allocate the data on varying number of nodelets to restrict total computation power. The relative speedup versus 1-nodelet execution increases rapidly for the graphs with lower degrees. Overall, the results show a sub-linear scaling pattern due to the way level-synchronous BFS works. During initial and final breadths, GCs are mostly under utilized; therefore having higher number of nodelets do not directly translate into performance.

V. Related Work

EMU architecturally inherits conceptual characteristics of several earlier systems. Cray-XMT [13] is a highly multi-threaded shared-memory architecture where all memory instructions go through the entire memory system, therefore data-locality is no longer a concern. Active Messages [8] pack the instruction address to integrate communication and computation.The Execution Migration Machine [15] also relies on migrating threads similar to EMU but targets SMP systems.

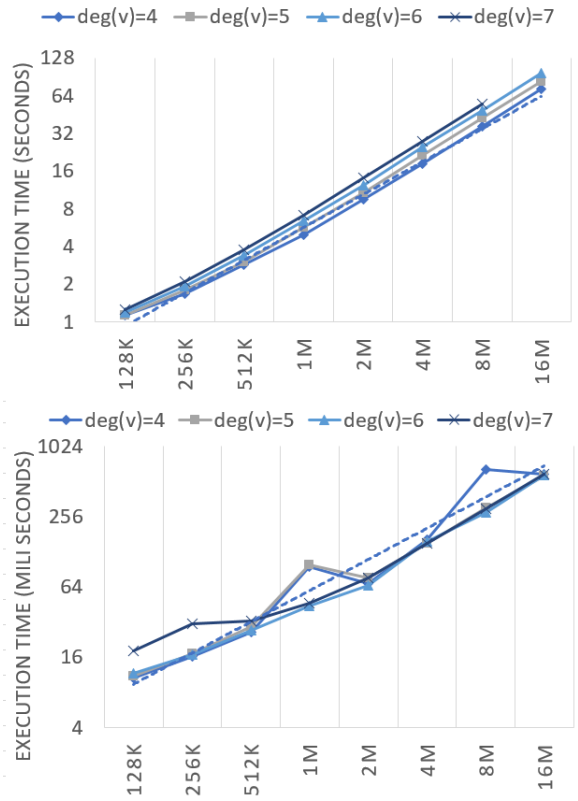

Fig. 6: BFS execution times on EMU on the top and CPU times on the bottom.

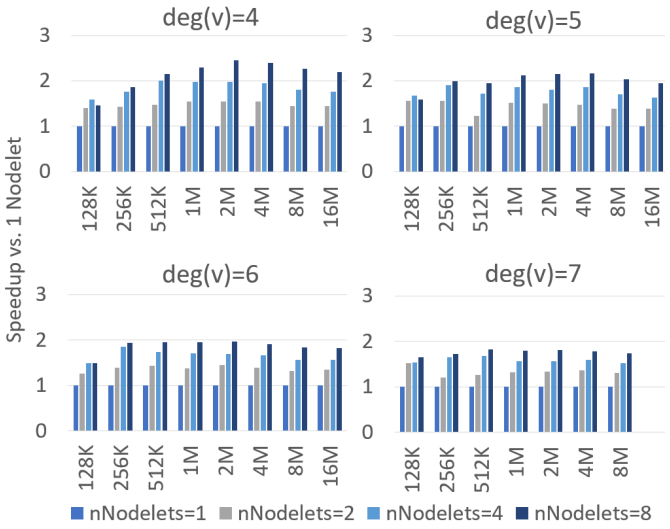

Fig. 7: Strong scaling of different graph sizes.

Processing in Memory (PIM) based architectures [14][12] allow in/near memory computation with higher energy efficiency. However, only a few of these architectures were implemented in actual hardware, with limited amount of computational power [10]. Software based approaches like Charm++ [1] helps load balance on distributed memory systems by migrating computational objects; whereas PGAS based systems like UPC [9] usually rely on stationary threads operating near the partitioned memory location.

VI. Conclusion

EMU presents a different approach to solve increasingly worsening memory bottleneck problem in high performance computing. The proposed solution relies on migrating threads to the location where the memory resides, and EMU achieves this without programmer intervention.

Our work shows that there are several unique architectural considerations that need to be addressed while developing for the EMU platform. While the programming interface mainly relies on Cilk, proper usage of intrinsics are required for performance effective algorithm design.

Acknowledgments

This material is based upon work supported by the U.S. Department of Energy, Office of Science, Office of Advanced Scientific Computing Research, under contract number DE-AC05-00OR22725. This manuscript has been authored by UT-Battelle, LLC, under contract DE-AC05-00OR22725 with the US Department of Energy (DOE). The US government retains and the publisher, by accepting the article for publication, acknowledges that the US government retains a nonexclusive, paid-up, irrevocable, worldwide license to publish or reproduce the published form of this manuscript, or allow others to do so, for US government purposes. DOE will provide public access to these results of federally sponsored research in accordance with the DOE Public Access Plan (http://energy.gov/downloads/doe-public-access-plan).

978-1-5386-5989-2/18/$31.00 ©2018 IEEE

REFERENCES

[1] Acun, B., Gupta, A., Jain, N., Langer, A., Menon, H., Mikida, E., Ni, X., Robson, M., Sun, Y., Totoni, E., Wesolowski, L., Kale, L.: Parallel programming with migratable objects: Charm++ in practice. In: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp. 647–658. SC '14, IEEE Press, Piscataway, NJ, USA (2014), https://doi.org/10.1109/SC.2014.58

[2] Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system, vol. 30. ACM (1995)

[3] Boroumand, A., Ghose, S., Patel, M., Hassan, H., Lucia, B., Hsieh, K., Malladi, K.T., Zheng, H., Mutlu, O.: Lazypim: An efficient cache coherence mechanism for processing-in-memory. IEEE Computer Architecture Letters 16(1), 46–50 (Jan 2017)

[4] Borštnik, U., VandeVondele, J., Weber, V., Hutter, J.: Sparse matrix multiplication: The distributed block-compressed sparse row library. Parallel Computing 40(5-6), 47–58 (2014)

[5] Chamberlain, B., Callahan, D., Zima, H.: Parallel programmability and the chapel language. The International Journal of High Performance Computing Applications 21(3), 291–312 (2007)

[6] Changrui, W., Fan, C., Huizhi, C.: A high-performance heterogeneous embedded signal processing system based on serial rapidio interconnection. In: 2010 3rd IEEE International Conference on Computer Science and Information Technology (ICCSIT). vol. 2, pp. 611–614. IEEE (2010)

[7] Dysart, T., Kogge, P., Deneroff, M., Bovell, E., Briggs, P., Brockman, J., Jacobsen, K., Juan, Y., Kuntz, S., Lethin, R., McMahon, J., Pawar, C., Perrigo, M., Rucker, S., Ruttenberg, J., Ruttenberg, M., Stein, S.: Highly scalable near memory processing with migrating threads on the emu system architecture. In: Proceedings of the Sixth Workshop on Irregular Applications: Architectures and Algorithms. pp. 2–9. IA3 '16, IEEE Press, Piscataway, NJ, USA (2016), https://doi.org/10.1109/IA3.2016.7

[8] von Eicken, T., Culler, D.E., Goldstein, S.C., Schauser, K.E.: Active messages: A mechanism for integrated communication and computation. In: Proceedings of the 19th Annual International Symposium on Computer Architecture. pp. 256–266. ISCA '92, ACM, New York, NY, USA (1992), http://doi.acm.org/10.1145/139669.140382

[9] El-Ghazawi, T., Smith, L.: Upc: unified parallel c. In: Proceedings of the 2006 ACM/IEEE conference on Supercomputing. p. 27. ACM (2006)

[10] Gaeke, B.R., Husbands, P., Li, X.S., Oliker, L., Yelick, K.A., Biswas, R.: Memory-intensive benchmarks: Iram vs. cache-based machines. In: Proceedings 16th International Parallel and Distributed Processing Symposium. pp. 7 pp– (April 2002)

[11] Gao, M., Ayers, G., Kozyrakis, C.: Practical near-data processing for in-memory analytics frameworks. In: 2015 International Conference on Parallel Architecture and Compilation (PACT). pp. 113–124 (Oct 2015)

[12] Hall, M., Kogge, P., Koller, J., Diniz, P., Chame, J., Draper, J., LaCoss, J., Granacki, J., Brockman, J., Srivastava, A., Athas, W., Freeh, V., Shin, J., Park, J.: Mapping irregular applications to diva, a pim-based data-intensive architecture. In: Proceedings of the 1999 ACM/IEEE Conference on Supercomputing. SC '99, ACM, New York, NY, USA (1999), http://doi.acm.org/10.1145/331532.331589

[13] Mizell, D., Maschhoff, K.: Early experiences with large-scale cray xmt systems. In: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing. pp. 1–9. IPDPS '09, IEEE Computer Society, Washington, DC, USA (2009), https://doi.org/10.1109/IPDPS.2009.5161108

[14] Patterson, D., Anderson, T., Cardwell, N., Fromm, R., Keeton, K., Kozyrakis, C., Thomas, R., Yelick, K.: A case for intelligent ram. IEEE Micro 17(2), 34–44 (Mar 1997)

[15] Shim, K.S., Lis, M., Khan, O., Devadas, S.: The execution migration machine: Directoryless shared-memory architecture. Computer 48(9), 50–59 (Sept 2015)

[16] Standard, J.: High bandwidth memory (hbm) dram. JESD235 (2013)

[17] Technology, E.: Emu chick specifications, http://www.emutechnology.com/products/emu-chick-specifications/

[18] Zhang, D., Jayasena, N., Lyashevsky, A., Greathouse, J.L., Xu, L., Ignatowski, M.: Top-pim: Throughput-oriented programmable processing in memory. In: Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing. pp. 85–98. HPDC '14, ACM, New York, NY, USA (2014), http://doi.acm.org/10.1145/2600212.2600213

978-1-5386-5989-2/18/$31.00 ©2018 IEEE