Accelerating Graph Community Detection with Approximate Updates via an Energy-Efficient NoC

Karthi Duraisamy
School of EECS, WSU
Pullman , USA
duraisam@eecs.wsu.edu

Hao Lu
School of EECS, WSU
Pullman , USA
hlu@eecs.wsu.edu

Partha Pratim Pande
School of EECS, WSU
Pullman , USA
pande@eecs.wsu.edu

Aananth Kalyanaraman
School of EECS, WSU
Pullman , USA
ananth@eecs.wsu.edu

ABSTRACT

Community detection is an advanced graph operation that is used to reveal tightly-knit groups of vertices (aka. communities) in real-world networks. Given the intractability of the problem, efficient heuristics are used in practice. Yet, even the best of these state-of-the-art heuristics can become computationally demanding over large inputs and can generate workloads that exhibit inherent irregularity in data movement on manycore platforms. In this paper, we posit that effective acceleration of the graph community detection operation can be achieved by reducing the cost of data movement through a combined innovation at both software and hardware levels. More specifically, we first propose an efficient software-level parallelization of community detection that uses approximate updates to cleverly exploit a diminishing returns property of the algorithm. Secondly, as a way to augment this innovation at the software layer, we design an efficient Wireless Network on Chip (WiNoC) architecture that is suited to handle the irregular on-chip data movements exhibited by the community detection algorithm under both unicast- and broadcast-heavy cache coherence protocols. Experimental results show that our resulting WiNoC-enabled manycore platform achieves on average 52% savings in execution time, without compromising on the quality of the outputs, when compared to a traditional manycore platform designed with a wireline mesh NoC and running community detection without employing approximate updates.

CCS CONCEPTS

•Theory of computation→ Graph algorithms analysis; **•Hardware → Network on chip**

KEYWORDS

Graph Community Detection, Wireless Network-on-Chip

1. INTRODUCTION

In the era of big data, graph analytics is starting to take a center-stage. Data collected from numerous scientific, social and internet-scale applications render themselves in ways that suit graph-theoretic modeling and analysis. For instance, in social networks, users are represented as nodes and their interactions represented as edges. Such modeling opens the door for numerous venues in pattern discovery and posing structural queries about the data. Community detection is an exemplar of such a discovery operation. Given a graph G(V,E), the goal of community detection is to identify tightly-knit subgroups of vertices (aka. "communities"), as illustrated by the example in Fig. 1. The community detection operation [1] reveals the natural divisions that typically exist in real-world networks, and finds application in a number of scientific domains including (but not limited to) social network analysis, collaboration networks, bioinformatics and electric power grid. Despite its broad application base and the recent developments in manycore processing, executing community detection over large real-world graphs remains challenging. Community detection algorithms typically have an iterative structure, where all vertices are scanned (in an arbitrary order) at every iteration and at each vertex, information from all neighboring vertices are queried to make local decisions. This imposes a significant overhead due to the repetitive need for communicating the latest of updates among neighbors, coupled with the fact that the neighbors need *not* necessarily be co-located in memory. Furthermore, as shown later in Sec. 5.2, the on-chip traffic pattern that arise from community detection computation is long-range heavy and irregular in nature.

To scale an advanced operation such as community detection to large networks, it is necessary to innovate at both the software and the hardware layers. More precisely, given that irregular data movement is the primary limiting factor in the performance of graph applications, we posit that reducing the cost and/or the volume of data movement will be critical in scaling up graph applications. Furthermore, in scaling up graph applications, the cost of energy consumption cannot be ignored, and exploring ways to reduce data movement and making it more effective will impact the energy cost of such applications. To this end, in this paper, we explore complementary strategies–at the software and hardware layers–as a way to derive a combined benefit in time and energy performance of the community detection operation on a single- chip manycore platform. More specifically, we present i) a way to *reduce the intensity* of data movement by deploying an approximate update technique at the software layer; and ii) an efficient design of a Network-on-Chip (NoC) that is aimed at *reducing both the latency and energy consumption.*

2. RELATED WORKS

Implementing graph analytics on specialized parallel architectures has been a relatively recent pursued task. The Graph500 benchmark [4] has focused on evaluating modern day supercomputing platforms based on their performance for Breadth First Search (BFS)–a fundamental operation involving

DAC '17, June 18-22, 2017, Austin, TX, USA © 2017
ACM. ISBN 978-1-4503-4927-7/17/06…$15.00
DOI: http://dx.doi.org/10.1145/3061639.3062194

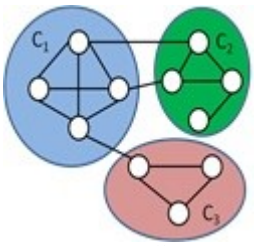


Fig. 1: Illustrative example for community detection. Given a graph G(V,E), this figure shows a possible community-wise partitioning of vertices in V.

graph traversal. A detailed study on using microarchitectures for graph analytics has been presented in [5]. In [6], the performance of Symmetric Multiprocessor (SMP) systems for large-scale graph analytics has been analyzed. There are several previous works exploring the efficiency of SMPs for applications with inherent irregularities [7][8][9]. Parallel implementations on multicore and manycore platforms for advanced graph operations such as graph community detection and graph coloring have been presented in (e.g., [10][11][12]). GPU implementation for executing community detection using label propagation has been already explored [13]. While GPUs have their performance advantages, codes need to be data-parallel to take full advantage of the platform. Graph applications, however, are latency-bound applications that inherently generate large volumes of irregular memory lookups due to edge dependencies. Consequently, manycore NoC architectures such as the one presented in this work are more naturally suited for optimized execution of such graph applications. Furthermore, a SIMD platform such as a GPU is ill-equipped to take advantage of heuristics such as early termination since they tend to generate variable work rates, and would require dynamic subset/data selection and load distribution capabilities.

A recent study concluded that the small-world network-enabled WiNoC architectures are better suited for handling the irregular data movements exhibited by the graph-computation applications than the conventional wireline mesh and high-radix NoC architectures. However, this work [2], did not consider the impacts of any approximate update scheme and cache coherence protocols on the execution time and energy dissipation profile of the manycore chips running graph analytics. The work presented in this paper, represents the first of its kind in characterizing the role of these two software and hardware aspects in designing manycore-based platforms for community detection.

3. COMMUNITY DETECTION

A scalable parallel implementation for executing community detection on conventional manycore architectures has been proposed recently [14]. This method, called *Grappolo*, implements a multi-phase, multi-iteration heuristic to maximize the modularity [15] of the output partitioning. In what follows, we describe the algorithm for the original version of Grappolo.

3.1 Grappolo

Given a graph *G(V,E)*, with *n* vertices and *m* edges, Grappolo iteratively tries to improve the overall modularity by running multiple "phases", and multiple iterations within each phase, until modularity gain becomes negligible. The steps of a phase are:

1. Initially, each vertex is assigned to a community of its own.
2. Within each iteration the vertices are scanned in parallel and for each vertex, a decision is made to determine whether or not to migrate it to one of its neighboring communities (as defined by the communities of its neighbors). For executing this step, the latest community information from all neighboring vertices are pulled by a given vertex. Locking is used to prevent concurrent and potentially conflicting updates from happening at the neighbors. In this model, locking needs to happen at two data structures – one at the level of a neighboring vertex and another at the level of the community that holds that vertex. This software version is labelled "fully synchronized update" version.
3. The algorithm proceeds to subsequent iterations until the gain achieved in modularity becomes negligible. Reaching convergence by this criterion marks the end of current phase and the algorithm constructs a compacted graph G'(V',E') by collapsing every community detected in G into a single meta-vertex in G', and creating edges with weights corresponding to intra- and inter-community links in G.
4. Subsequently, the algorithm initiates the next phase on the newly compacted graph G', until no more appreciable modularity gain is achieved (i.e., convergence).

One of the major contributors to the computational cost of the above algorithm is its local neighborhood querying in step (2). The fully synchronized update model ensures that the most recent community information of a vertex is made available to any of its vertex neighbors for its migration decision. However, the cost of such full synchronization manifests itself in a pattern of large volume and potentially irregular data traffic. Therefore, this fully synchronized version is expected to yield a fast convergence (i.e., less number of iterations due to latest information availability) but at the possible expense of more time cost per iteration.

Graphs	Vertex Count	Edge Count	Execution Time (sec)			Modularity	
			FS	FS+ET	Gain	FS	FS+ET
ASJ	22,963	48,436	0.7608	0.3437	55%	0.66216	0.66001
ASTRO	16,706	121,251	0.2998	0.1854	38%	0.73206	0.73216
COND	31,163	120,029	0.3174	0.11961	62%	0.76238	0.76397
HEP	8,361	15,751	0.0535	0.0343	36%	0.84969	0.84667
PGP	10,680	24,316	0.0829	0.0623	25%	0.88198	0.88282

Table 1: Evaluation of the approximate update scheme (via ET) on community detection's execution time and quality (modularity). All runs were performed on a 32-core Intel multicore platform.

3.2 Approximating Updates

As explained above, within each iteration of Grappolo algorithm using the fully synchronized update model, the computation time is dominated by the time taken to decide on a vertex migration (step 2). For this paper, we designed an alternative implementation of Grappolo that deviates from making precise updates. More specifically, unlike classical heuristics that mainly focus on reducing the runtime, our proposed scheme focuses on reducing the intensity of data lookups, thereby generating a potential to reduce both time and energy caused by the data

No. of Edge Traversals and Community Lookups in Phase-1						
Graphs	NET			NCL		
	FS	FS+ET	Reduction	FS	FS+ET	Reduction
ASJ	871,848	307,696	64.71%	603,953	242,874	59.79%
ASTRO	2,667,522	812,821	69.53%	850,865	364,424	57.17%
COND	3,600,870	814,024	77.39%	1,601,311	474,852	70.35%
HEP	283,518	104,532	63.13%	170,780	75,148	56.00%
PGP	437,688	161,352	63.14%	224,060	101,532	54.69%

Table 2: Evaluation of the reduction in number of edges traversals (NET) and number of community lookups (NCL), induced by ET.

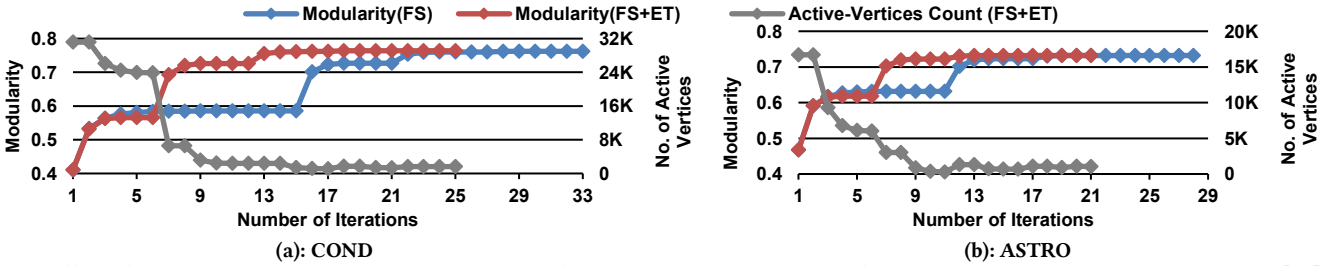
movements. The main idea hinges on the key observation of a diminishing returns property in quality as the iterations progress [14]. As shown by the results on real-world graphs (Fig. 2), the gain in modularity plateaus after a few initial iterations, because of significantly fewer community updates for vertices during the later iterations. We take advantage of this observation and include a counter, *nElapsed*, at every vertex to denote the number of iterations that have elapsed since its last successful migration (i.e., change in community). If the counter *nElapsed* exceeds a certain threshold τ, then we "terminate" that vertex–i.e., we (optimistically) stop considering that vertex during any subsequent iteration of that phase implying that the vertex is locked into that community. This heuristic has two performance advantages: a) it enables a faster convergence of modularity by reducing both the number of vertices that need to be processed at every iteration and the total number of iterations required, and b) it also reduces the volume of memory lookups generated from each vertex because terminated vertices will stop seeking information from their neighborhoods. However, this also runs the risk of possibly degrading the final quality achieved (measured by modularity). We tested with a number of graph inputs to study this precision vs. performance tradeoff. Specifically, we use a set of five DIMACS10 [16] clustering instance graph data sets, Hep-th (HEP), Astro-ph (ASTRO), Comd-mat-2003 (COND), PGPgiantcompo (PGP) and as-22july06 (ASJ). The vertex and edge counts for all the five considered graphs are listed in Table 1. For our experiments, we set τ=3. First, we compare the convergence rates (measured in the number of iterations) of the fully synchronized ("FS") and the fully synchronized with early termination ("FS+ET") versions of Grappolo, for the ASTRO and COND input graphs in Figs. 2(a) and 2(b), respectively. As it can be seen, the number of iterations reduces from 28 iterations in FS to 20 iterations in FS+ET implementation for ASTRO (Fig. 2(a)). COND also follows a similar trend (Fig. 2(b)). The faster convergence in the FS+ET version is caused by the early-termination of non-migrating vertices, which results in less changes to accumulate as iterations progress and subsequently reach peak modularity in reduced number of iterations. Aside from the lowered number of iterations, it should also be noted that with the early termination, the execution time per iteration significantly reduces as the algorithm progresses. As the iterations grow, more vertices settle in their communities, allowing a large number of vertices to be terminated within each iteration. This statement is corroborated by Figs. 2(a) and 2(b) in which the active number of vertices reduces with the increase in number of iterations. The expected reduction in overall runtime is confirmed in the results of Table 1. Notably this table shows that the runtime savings are achieved without compromising on the output quality (modularity). Table 2 demonstrates the reductions achieved by the use of early termination in the number of edges traversed and the number of community lookups, both of which are also the dominant contributors toward memory traffic.

4. NETWORK ON CHIP PLATFORM

As explained in Sec. 3, the computation of community detection involves migration of vertices between communities, and thereby a high number of shared memory accesses. Employing early termination accelerates Grappolo by lowering the volume of memory accesses. In addition to this software-driven speedup, further acceleration can be achieved by reducing the latency associated with the memory accesses, using an efficient NoC. Aside from the nature of the application, the on-chip traffic patterns that arise from the memory accesses also depend on the adopted cache-coherence protocol. To explain this further, we consider the traffic patterns induced by two highly contrasting cache-coherence protocols, the two level MESI Directory protocol (henceforth referred to as "Directory") and the AMD's Hammer based Hyper Transport coherence protocol (henceforth referred to as "Hammer") [17]. For each data block available on-chip, the Directory protocol employs a full bit directory indicating all the sharer and owner nodes. Hence, whenever there is a read/write request associated with a data block, the Directory protocol easily identifies and communicates with the one (or) more sharer nodes to inform the data block updates. On the other hand, the Hammer protocol avoids using full bit directories and simply broadcasts all read/write requests that are associated with the on-chip data blocks. Thus, the Hammer protocol involves dense broadcasts while the directory protocol mainly generates a unicast-heavy on-chip traffic pattern that involves sparse multicasts [17]. However, it should be noted that the on-chip area overhead associated with implementing the Hammer protocol is much lower than that needed for the Directory [17]. Hence, though Directory protocol is conventionally employed in manycore platforms, the Hammer provides more scalability for large chips and is preferable for systems with area constraints. In this work, we evaluate the impact of the early termination technique explained above in the presence of both the directory and the Hammer protocols.

4.1 Suitable NoC Architecture

The NoC architecture is the communication backbone for any manycore chip and it must align well with the application's traffic



(a): COND (b): ASTRO

Fig. 2: Effect of early termination on the convergence rate of modularity, and the number of vertices processed per iteration. Following [13], iterations continue until the attained modularity saturates up to the third decimal place.

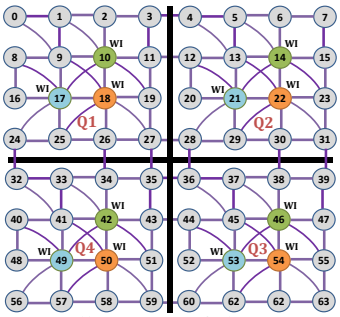


Fig. 3: Illustration of the 4-region (Q1-Q4) WiNoC indicating WI placements and the wireline connections. Color of the WI indicates the wireless channel.

patterns. As shown later in Section 5.2, in case of the Directory protocol, about 60% of total traffic is exchanged over NoC routers that are at least 5 inter-router hops apart (a hop is defined as the distance between two adjacent routers in a mesh NoC). Moreover, the Grappolo on-chip traffic patterns do not indicate presence of any high intensity data-transfers between specific pairs of cores and the data-exchanges are actually well distributed. On the other hand, memory accesses under the Hammer protocol induce broadcasts which must be distributed across all the NoC routers. Thus, the on-chip traffic exhibited by Grappolo in presence of Directory and Hammer coherence protocols is long-range-heavy. The wireless channel in WiNoC is a broadcast communication medium capable of establishing one-hop links even between physically distant nodes [3]. Hence, for both Directory and Hammer induced traffic, WiNoC provides an efficient solution.

4.2 NoC Design

The WiNoC architecture follows a small-world architecture [18]. Primarily, we leverage the small-world property while constructing the NoC topology to design a low hop count on-chip interconnection network. Essentially, a small-world NoC incorporates mostly short-range links along with a few long-range shortcuts following the power-law distribution of the edges in a small-world graph [18][3]. The links are established such that the overall inter-router hop count is minimized. When implemented with conventional metal wires, the long-range shortcuts in the small-world interconnects are extremely costly in terms of power and delay. Hence, in WiNoC, power efficient wireless links are used to connect the routers that are separated by long distances [3]. We use the 3 mm-wave wireless channels operating in the 30, 60 and 90GHz range here. The on-chip wireless interfaces (WIs) are designed following [3]. Fig. 3 illustrates a 64 core WiNoC architecture indicating the placement of the WIs, channel assignments and the wireline connectivity.

4.2.1 Data-Transfer Protocols

In WiNoC for unicast packets, we follow the deadlock-free adaptive-layered shortest path routing. In this work, for Hammer broadcast packets, we follow a region based message distribution and acknowledgment (ACK) aggregation strategy. In this strategy, the whole system is first divided into multiple equal sized non-overlapping regions (As an example, WiNoC illustrated in Fig. 3 uses four regions). Upon injection, a broadcast message is first forwarded from the source node to the nearest WI in its own region and is then transmitted wirelessly. The message is then received simultaneously by the WIs in all other destination regions (WIs that are operating in the same channel as the transmitting WI). Finally, the message is distributed from each region destination WI to the final destination nodes using wireline links using a tree multicast mechanism [20]. In this mechanism, a destination WI first forwards the received broadcast message to a set of pre-defined intermediate nodes (Ex: nodes 13, 14, 21, 22 in Q2 of the NoC depicted in Fig. 3). The message is then replicated at these intermediate nodes and are finally forwarded to the final destination nodes using intra-region wireline links.

In WiNoC, each broadcast message uses an additional field to indicate the state of the broadcast distribution (whether the packet is on the way to source WI starting from the source node or the message is undergoing a regional distribution starting from a destination WI). This state field can only be modified at the source node and the source region WI. With the use of this additional field and the cycle-free tree broadcast distribution, the WiNoC broadcast communication eliminates any possible deadlocks.

For cache coherence-induced broadcasts, each packet is associated with a set of acknowledgment (ACK) messages that are transmitted from each destination back to the source [21]. In WiNoC, the acknowledgments are aggregated at the intermediate nodes. Initially, each destination forwards its acknowledgment message to the same intermediate node that forwarded the original multicast message. The ACKs are then aggregated at each region WIs and finally get forwarded towards the source node.

The WiNoC uses a distributed MAC protocol as proposed in [19] to resolve the channel access contention among the WIs.

5. EXPERIMENTAL RESULTS

To evaluate the effectiveness of our WiNoC architecture for implementing community detection with approximate updates, we compare against a more traditional and an industry standard NoC, the mesh architecture. For this mesh NoC, we use dimension ordered XY routing for unicast packets and XY–tree broadcast mechanism for Hammer broadcasts. We also use ACK aggregation for the mesh following the method proposed in [21].

5.1 Experimental Setup

For all the following evaluations, we use the same set of five clustering instance graphs that were used in Section 3.2 (ASJ, ASTRO, COND, HEP, PGP). We use GEM5 full system simulator to obtain detailed processor and network information [22]. All the gem5 simulations were performed in the full-system mode where we leveraged the RUBY and GARNET modules of GEM5 to implement detailed memory and on-chip interconnection models. The FS and FS+ET software implementations of the community detection algorithm are parallel (using OpenMP) and are 64-way multithreaded for our GEM5 simulations. The simulated system is comprised of sixty-four x86-cores operating at 2.5GHz arranged in a 20mm×20mm die. The memory system is comprised of private 64KB L1 instruction and data caches and a shared 16MB L2 cache (256KB distributed L2 per core). For both mesh and WiNoC, we use a generic three stage router capable of multiport multicast replicating [21]. Energy dissipation of the NoC routers, inclusive of the MAC block, was obtained from the synthesized netlist by running Synopsys™ Prime Power using commercial 28nm nodes. Energy dissipated by wireline links is obtained through Cadence SPECTRE simulations, taking into consideration the length and layout of wired links. The processor-level statistics generated by

the GEM5 simulations are incorporated into McPAT (Multicore Power, Area, and Timing) to determine the processor power [23].

5.2 Analysis of Traffic Patterns

Before presenting the detailed performance evaluation, first we analyze the traffic patterns generated by Grappolo. Fig. 4 shows the percentage of the total on-chip traffic exchanged between the communicating routers that are separated by a certain number of mesh NoC hops (indicated by *h*), while executing the FS and FS+ET versions of Grappolo. As seen, both versions of Grappolo generate significant amount of long-range traffic with both Directory and Hammer. Around 50% of the total traffic under Directory protocol requires a minimum of 6 mesh hops while with Hammer protocol, about 62% of the total traffic requires at least 6 hops. These long-range traffic patterns that arise form Grappolo computations can be attributed to shared-memory accesses caused by neighborhood graph lookups and vertex migration. Another contributor to long-range traffic is Hammer coherence induced broadcasts (constituting about 25% of all traffic), which can require up to 14 inter-router hops in a 64-core NoC.

Fig. 5 shows the percentage of the traffic associated with each of the top six traffic hotspots in the system, for running Grappolo. With Directory, the top-six traffic hotspots are responsible for about 26% of the total traffic. This injection originates mostly from the master cores that handle the graph compaction– a largely serialized step owing to the need to gather vertices belonging to various communities. In case of Hammer, shared memory accesses induce broadcasts that need to be communicated with all the NoC routers. The broadcasts are then followed with the injection of ACK messages from each of the destination node, sent back to the source node. Because of these broadcasts and ACK messages, the impacts of the master cores are less pronounced in Hammer protocol and the volumes of messages injected by all the NoC routers remain fairly close to each other (Fig. 5).

These long-range and hot spot traffic patterns are observed consistently over all the graphs considered and for both FS and FS+ET implementations. WiNoC is highly suitable to handle irregular traffic patterns involving long-range communication and traffic hotpots [2][3]. Hence, it is a natural candidate as the communication backbone for manycore platforms in this work.

5.3 Execution Time Evaluation

Fig. 6(a) compares the execution times of the mesh and WiNoC interconnected manycore systems running FS and FS+ET Grappolo versions with both Directory and Hammer protocols. All the values in Fig. 6(a) are normalized with respect to the execution times observed in the baseline system with mesh running FS version. As it can be observed from this figure, regardless of the NoC platform and cache coherence protocol employed, the use of early termination technique greatly benefits the execution of Grappolo. When compared to the FS version, the software implementation using approximate updates (i.e., FS+ET) achieves on an average of 40.7% and 39.1% execution time savings with the Hammer and Directory cache coherence, respectively.

In order to clearly demonstrate the benefit of employing the WiNoC for Grappolo, in Fig. 6(b) we compare the runtimes of the FS+ET version of Grappolo in Mesh and WiNoC. The WiNoC achieves on an average of 22.4% and 15.96% reduction in execution times with Hammer and Directory protocols respectively, when compared to the mesh NoC. The reason for the variation in gain achieved for the two cache coherence protocols can be understood from the NoC latency plot shown in Fig. 7. When compared to the Directory protocol, the Hammer protocol induced on-chip messages give rise to a higher network latency in the baseline mesh NoC because of the fact that about 25% of the injections are broadcasts. Thus, in a mesh NoC, memory accesses cause higher penalties on the overall execution time with the Hammer protocol than with the Directory protocol. Hence, accelerating the memory

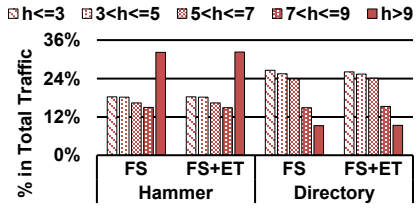


Fig. 4: Distribution of the on-chip traffic based on communication distances. Parameter *h* in the legend indicates the value of hop counts in a XY routing based mesh. Shown values are averages for all the 5 graphs used.

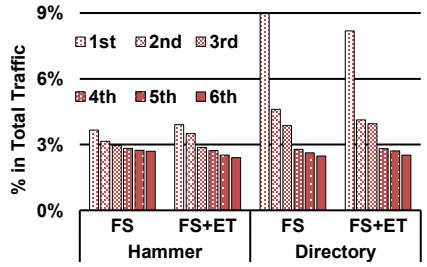


Fig. 5: Percentage of traffic injected by the top-6 hotspots. Values are averages for all 5 graphs considered in this work.

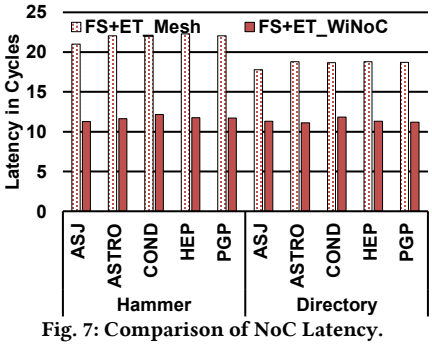


Fig. 7: Comparison of NoC Latency.

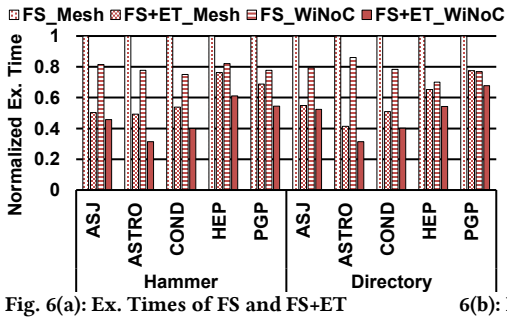
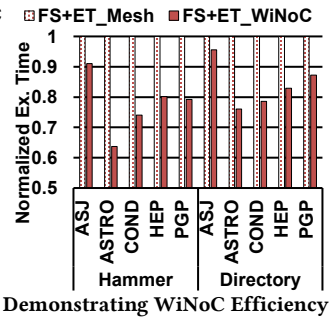


Fig. 6(a): Ex. Times of FS and FS+ET



6(b): Demonstrating WiNoC Efficiency

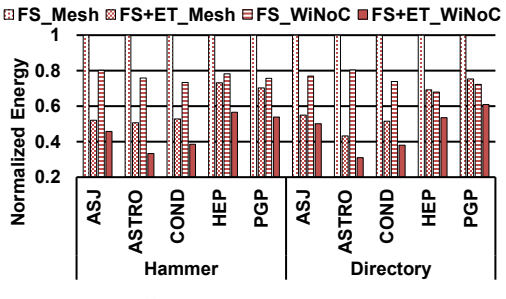


Fig. 8: Full System Energy Consumption

accesses using the WiNoC leads to more gain in execution time with Hammer than with the Directory protocol (Fig. 6(b)). Among the considered input graphs, ASJ and ASTRO involve the lowest and highest traffic injection rates, respectively [2]. Hence, ASJ and ASTRO achieve the lowest and highest execution time gains, respectively, using WiNoC (Fig. 6(b)). On overall, when compared to the baseline architecture (mesh running FS), our manycore implementation (WiNoC running FS+ET) lowers the execution time by 52.15% for the considered input graphs (Fig. 6(a)).

5.4 Full System Energy Consumptions

Fig. 8 compares the energy consumptions of the manycore platforms executing the FS and FS+ET versions of Grappolo, employing Mesh and WiNoC architectures. All the values in this figure are normalized with respect to the energy consumption of the baseline architecture, the mesh NoC running the FS version.

Both the cache-miss rates and the subsequent NoC traffic caused by the cache-misses can be significantly reduced by the adopting early-termination. Compared to the baseline implementation (FS), the FS+ET implementation achieves from 29% (with ASJ graph) to 51% (with COND) lower cache miss induced NoC traffic, owing to its reduced number of shared memory accesses and lower run times. This reduced NoC traffic and lowered runtimes helps in enhancing the energy consumption profile of the manycore platform running FS+ET Grappolo. Even with mesh NoC, FS+ET achieves around 41% savings in full system energy consumptions, compared to FS version of Grappolo.

Next, we focus on comparing between mesh and WiNoC. Compared to the mesh, the WiNoC uses more energy efficient wireless links for long-range communication. Also, the average hop count of WiNoC is less than that of mesh due to its small-world connectivity. These enable WiNoC to reduce the network (and ultimately full-system) energy consumption. In addition, when compared to the mesh, the WiNoC architecture enables a lesser execution time (as shown in Fig. 6(b)) leading to improved CPU energy consumptions. Specifically, for running the FS+ET version, when compared to the mesh, WiNoC achieves 23.7% and 20.6% reduction in full-system energy consumption with Hammer and Directory coherence protocols, respectively. Due to the communication-intensive nature, the Hammer protocol achieved a higher improvement than the Directory protocol by using the WiNoC. Finally, when compared to the baseline mesh NoC running FS version, our WiNoC interconnected manycore platform running FS+ET version of Grappolo achieves about 53.9% savings in full system energy consumptions. These results (Figs. 6-8) demonstrate that our software and hardware level innovations lead to a significant acceleration of the community detection operation, while also making the proposed manycore platform more energy-efficient than the baseline architecture.

6. CONCLUSION

Designing an efficient manycore platform to execute the graph community detection problem presents two key challenges: (a) reducing the computation complexity of the community detection heuristics and (b) handling the irregular on-chip memory access patterns associated with the large-scale graph computations. To overcome these challenges, in this work, we have proposed an approximate update scheme for the community detection algorithm that lowers the computation complexity and reduces the volume of on-chip memory accesses. Furthermore, we have designed an efficient wireless NoC architecture that accelerates the on-chip memory accesses under both unicast- and broadcast-heavy cache coherence protocols. For the input graphs and the cache coherence protocols considered in this work, the proposed manycore implementation provides on an average of 52% execution time savings and 53.9% lower energy consumption to perform community detection computation, over a baseline architecture using wireline mesh NoC and running the traditional community detection (without any approximate updates). The techniques proposed here have the potential to be extended to many other data-bound irregular graph applications as well.

ACKNOWLEDGMENTS

This work was partially supported by NSF grants CNS 1564014, CCF 1514269 and CCF 1162202 and DOE award DE-SC-0006516.

7. REFERENCES

[1] S. Fortunato "Community detection in graphs." *Physics Reports*, 486(3): 75-174.
[2] K. Duraisamy, at al., "High-Performance and Energy-Efficient Network-on-Chip Architectures for Graph Analytics". *ACM Transactions on Embedded. Computing Systems*, 15(4), Article-66, 26 pages.
[3] P. Wettin, et al. "Design Space Exploration for Wireless NoCs Incorporating Irregular Network Routing," *IEEE TCAD*, 33(11).
[4] Graph500 : http://www.graph500.org/.
[5] D. Ediger, "Analyzing Hybrid Architectures for Massively Parallel Graph Analysis", Ph. D. Dissertation, Georgia Institute of Technology, May-2013.
[6] D. A. Bader, et al. "On the architectural requirements for efficient execution of graph algorithms". In *Proc. of ICPP, 2005*, pp.547–556.
[7] M. Castro, et al. "Analysis of computing and energy perfor- ´ mance of multicore, NUMA, and manycore platforms for an irregular application". In *Proc. of 3rd Workshop on Irregular Applications: Architectures and Algorithms*.
[8] E. Francesquini, et al. "On the energy efficiency and performance of irregular application executions on multicore, NUMA and manycore platforms". *Journal of Parallel and Distributed Computing*, 76, pp.32–48.
[9] M. Frasca, et al.. "NUMA-aware graph mining techniques for performance and energy efficiency". In *Proc. of IEEE Intl. Conf. on High Performance Computing. Networking, Storage and Analysis (SC)*, pp.1–11.
[10] H. Lu, et al. "Balanced coloring for parallel computing applications", *Proc. IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2015.
[11] D. Chavarría-Miranda, et al., "Scaling graph community detection on the Tilera manycore architecture," in *HiPC 2014*, 11 pages.
[12] E. J. Riedy, et al. "Parallel community detection for massive graphs." in *Parallel Processing and Applied Mathematics*, pp. 286-296. Springer, 2012.
[13] J. Soman and A. Narang, "Fast Community Detection Algorithm with GPUs and Multicore Architectures," In *Proc. of IPDPS*, 2011, pp. 568-579.
[14] H. Lu, M. Halappanavar, A. Kalyanaraman, "Parallel Heuristics for Scalable Community Detection", *Parallel Computing*, vol. 47, pp. 597-606, 2015.
[15] M. E. J. Newman, "Modularity and community structure in networks." *Proceedings of the National Academy of Sciences*, 103(23), pp.8577-8582, 2006.
[16] DIMACS10, The 10th DIMACS implementation challenge – graph partioning and clustering. URL: http://www.cc.gatech.edu/dimacs10/
[17] A. Ros, et al. "Dealing with Traffic-Area Trade-Off in Direct Coherence Protocols for Many-Core CMPs." *Advanced Parallel Processing Technologies*, Springer, pp. 11-27,
[18] T. Petermann, P. Los Rios, "Spatial Small-World Networks: A Wiring Cost Perspective", arXiv: condmat/0501420v2
[19] K. Duraisamy, R. Kim, P. Pande, "Enhancing Performance of Wireless NoCs with Distributed MAC Protocols", in *Proceedings of ISQED*, 2015.
[20] M. P. Malumbres, and Duato, J. "An efficient implementation of tree-based multicast routing for distributed shared-memory multiprocessors". *Jour. of Sys. Arch.*, 46(11), pp.1019-1032.
[21] T. Krishna, et al. "Towards the ideal on-chip fabric for 1-to-many and many-to-1 communication," *In Proceedings of the MICRO*, 2011, pp. 71–82.
[22] N. Binkert, et al., "The GEM5 Simulator", *ACM SIGARCH Computer Architecture News*, 39(2):1-7, 2011.
[23] S. Li, et al., "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures", In *Proc., of the 42nd Annual IEEE/ACM International Symp. on Microarchitecture*, pp. 469-480, 2009.