Hunting the Needle - The Potential of Innovation in Architecture

Peter M. Kogge
Computer Science and Engr.
Univ. of Notre Dame
Notre Dame, IN 46556
Email: kogge@nd.edu

Janice McMahon
Parallel Computing Consultant
Email: j.onanian.mcmahon@alum.mit.edu

Timothy J. Dysart
Tactical Computing Labs
Muenster, TX, USA
Email: tdysart@tactcomplabs.com

***Abstract*—Subgraph Isomorphism involves using a small graph as a pattern to identify within a larger graph a set of vertices that have edges that match, and is becoming of increasing importance in many application areas. Such problems exhibit the potential for very significant fine-grain parallelism, with individual threads having short lifetimes while touching potentially "distant" memory objects in very unpredictable and irregular fashion. This is difficult for conventional distributed memory systems to achieve efficiently, but an alternative that combines cheap multi-threading with threads that can migrate freely through a large memory is a more natural fit. This paper demonstrates the potential of such an architecture by comparing its execution characteristics for a large graph to that of several conventional parallel implementations on modern but conventional architectures. The gains exhibited by the migrating threads are significant.**

I. Introduction

Computing over graphs is perhaps the most wide-spread non-numerical class of problems studied today, and one for which relatively few novel architectural developments have been targeted (one exception [1]). One of the most difficult of such problems is subgraph isomorphism - finding within a larger graph a set of vertices and edges that match a given "pattern." Such problems exhibit significant amounts of parallelism that occurs dynamically, with computational thread lifetimes that are short and with unpredictable locality. This makes it tough to get efficient execution on conventional systems.

The specific problem studied here originated with the U.S.'s IARPA AGILE program [2], with a government-provided reference implementation, dataset generator, data set, and timing. Unlike many graph isomorphism problems, in this case there is notionally very few, or just one, instance of the subgraph (the "needle") in a much larger "haystack" graph.

Multiple parallel implementations are compared, with a key conclusion that the cost of managing the parallelism in conventional systems far outweighs the actual computations needed to solve the problem. For conventional distributed memory systems this cost seems to stem from the cost of "moving" a computation from one node to another, to support "edge-traversing" where the target vertex is "somewhere else."

In addition to conventional implementations, a multi-node shared memory implementation is also reported using a system where hardware, not software, manages the migration of computation from one physical node to another. The particular feature studied here involves relentless multi-threading where threads are free to migrate anywhere in the system without any explicit software involvement. When coupled with cheap spawns (to support the dynamic parallelism) and very cheap operations to update "at a distance", these features appear extraordinarily effective, with a prototype platform built out of FPGAs for the cores (and running at a mere 225MHz) more than competitive with a more modern system with 8 times as many cores running at a clock rate 11 times faster.

In organization, Section II addresses the specific problem and data set used for evaluation. Section III discusses multiple implementations of this problem on several different conventional parallel platforms. Section IV addresses both the architecture of the migrating thread system used in the experiments and the code written for the given problem. Section V performs a high level comparison of the various codes when run on real hardware. Section VI uses these results to extract information as to the real cost of "crossing node boundaries" in conventional architectures, and the savings possible by adopting alternative architectures. Section VII concludes.

II. The Graph Problem

Subgraph isomorphism has long been recognized as an important kernel for many applications. The MIT HPEC Graph Challenges, for example, host yearly competitions for graph-intensive problems, and one of the five current challenges is Subgraph Isomorphism [3], [4]. In this case the graphs are often unweighted, undirected, and without vertex or edge properties. The kinds of subgraphs searched for typically include triangles and k-trusses[1], both of which may have very many instances in a given graph. There are quite a few algorithms explicitly designed for these kinds of pattern searches [5], [6], [7], [8], [9], [10], [11].

In contrast the specific problem studied here originated with the U.S.'s IARPA AGILE program [2][2], with a government-provided reference implementation, data set generator, refer-

[1]A k-truss is a graph where each edge in it is part of k-2 triangles.

[2]In particular see the slides in this presentation related to "Workflow 2" Exact Match.

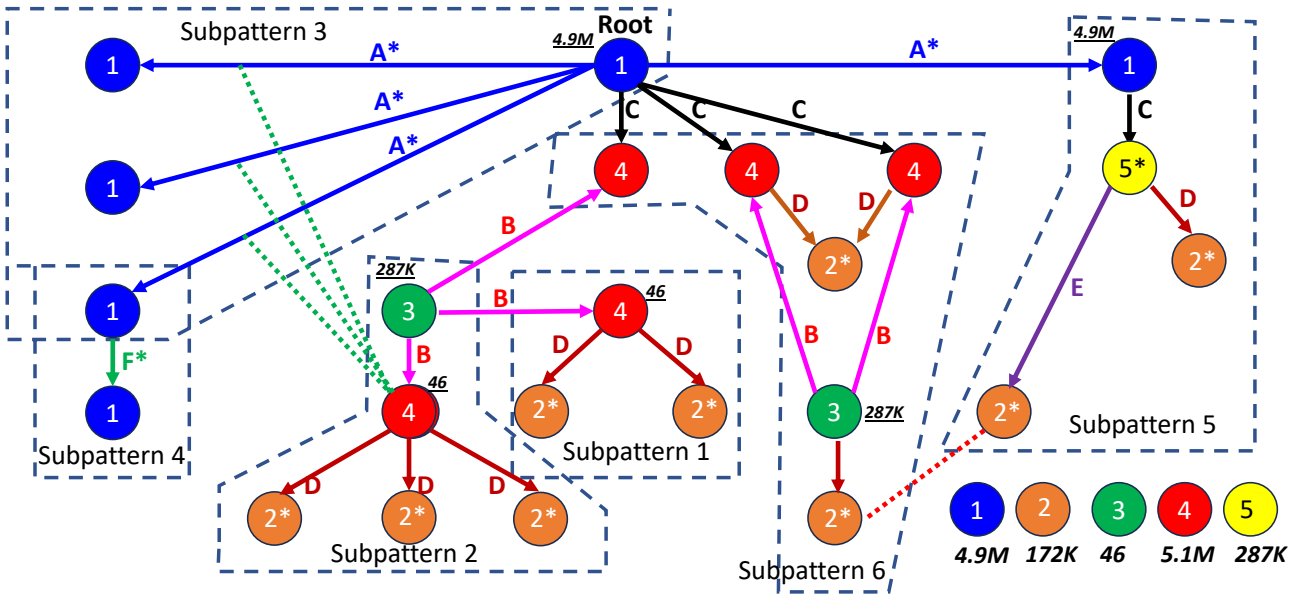

Fig. 1. Subgraph Pattern.

ence data set, and timing on a modern multi-node parallel system. In contrast to triangle or k-truss counting, the pattern for this subgraph is much more complex, and is a variant of a subgraph isomorphism problem where the target graph is built from multiple (five) types of vertices and multiple (six) types of directed edges, all of which have one or more properties associated with them.

Fig. 1 diagrams a simplified version of the subgraph pattern of interest [12]. Vertices of the same type are shown by circles of the same color, with a type label "1" through "5". Edges are also color-coded, with type labels "A" through "F", where for simplicity edges of type "A" are all actually edge type "F" turned around. An asterisk on a vertex or edge means that, to satisfy the pattern, some property of the vertex or edge must have a specific value. The dotted green lines represent that a separate relationship must hold between a property of a type A edge and that of a type 4 vertex. The red dotted line also indicates a specific relationship must hold between the two vertices of type 2.

For each subgraph found in the graph that matches the subgraph pattern, the discovered type 1 vertex labelled "Root" is to be returned. In all graphs used here there is exactly one matching root vertex.

A graph generator was available to create graphs of different sizes. The key on the lower right of Fig. 1 gives the number of vertices for the reference data set used in the rest of this paper that is about 2GB in size. For this graph there were about 10 million vertices and 38 million edges.

The standard approach to finding the matching vertices used by all implementations is to break the pattern into subpatterns 1 through 6, and identify matching parts of the overall graph sequentially one subpattern at a time. Parallelism is invoked within each subpattern.

III. Conventional Parallel Implementations

TABLE I
Implementation Platform Characteristics

Programming Model	Processor	Avail. Nodes	Cores/ Node	Core Clock
SHAD-1	AMD EPYC 7763	16	128	2.45GHz
SHAD-2	E5-2680 v2	16	20	2.8GHz
Cilk	Xeon Silver 4208	1	16	2.1GHz
Mig. Thread	FPGA Custom	16	16	225MHz

This problem was solved on several different conventional systems using a variety of programming models. Table I summarizes these pairings[3]. Column 1 correlates to the programming and execution models described below. The other columns give the high-level hardware features.

A. SHAD

SHAD: (S*calable High-Performance Algorithms and Data Structures* [13] is a C++ library that provides a common shared-memory, task-based, programming model for a variety of hardware architectures ranging from multi-core shared memory systems to multi-node distributed memory systems. It includes a library of logically shared-memory data structure templates designed to offer APIs for common parallel access and data updates regardless of the underlying system architecture. There is also an *Abstract Runtime Interface* that can set up and manage parallel execution of both functions and loops on different physical nodes.

SHAD uses one of two underlying multi-threading packages: *GMT* (Global Memory and Threading) [14] or Intel's *TBB* (Thread Building Blocks) [15].

[3]The architecture of the system used in the last row is discussed in the next section.

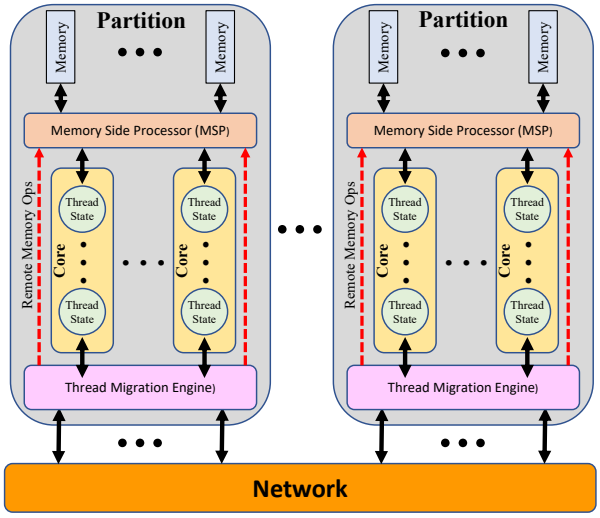

Fig. 2. A generic migrating thread architecture.

B. Cilk

Cilk is an extension of C [16], [17] designed to provide dynamic multi-threading in a multi-core shared memory system. Its main extensions are two keywords:

- A *cilk_spawn* prefix to a normal function call that indicates to the run-time that, if resources allow, the function call can be computed independently of the calling thread.
- A *cilk_sync* statement (no arguments needed) indicates that execution cannot proceed beyond this point until all child threads spawned in this block have completed.

A common addition is a parallel *for* loop that tries to spawn a new thread for each loop iteration.

A Cilk runtime manages the relationship between available hardware threads and program threads, similar to that of GMT or TBB from SHAD.

C. Use in Implementations

The SHAD-1 implementation was on a very modern hardware platform, but with SHAD software that was still immature and not tuned. The SHAD-2 system was on an older system that had a well-tuned SHAD implementation using the GMT package.

In both cases software was responsible for determining when an object being referenced was not local, generate network traffic to tell the physical node where that object resided that a particular computation should be run against it, and maintain any required synchronism with other activities.

The Cilk implementation was on a single dual socket server blade using the OpenCilk package [18]. While no software was needed to communicate between different physical nodes (as only one multi-core node was used), Cilk run-time software was needed to manage the mapping between logical and physical threads, and the synchronization of thread activities as around a cilk_sync.

IV. A Migrating Thread Implementation

As will be shown, there is strong evidence that the inefficiency with executing such problems on conventional architectures is handling the instances where transiting an an edge requires "crossing" a physical node boundary, and software must get in the middle. One possible architectural technique that may help simplify this is based on avoiding having to create messages and data by software, and send them between nodes to manage threads “over there”. Instead, threads are allowed to *migrate* without direct software involvement from any place in the system to any other place, thus avoiding much of the endlessly multiplying software stacks.

A. Migrating Thread Hardware Architecture

Fig. 2 diagrams one such architecture [19], [20]. Here each “node” has multiple cores as in a conventional processor module, but all cores (termed here “*GC cores*”) are far more heavily multi-threaded (cf. [21]) than conventional designs, with each having the capability of interleaving dozens of thread states. Each GC core executes an instruction for each of its threads in an interleaved round-robin like manner.

Further all memory in all nodes is in a shared *logical address space* whereby when a thread in any core makes a memory reference, the node hardware knows if the address is local or not. If not, the hardware in the core (not some software runtime) suspends the requesting thread, packages its state (i.e. register set), and sends the package to the node with the targeted memory, where the thread state is unpacked and restarted on a local GC core, again by hardware with no software involvement. The thread then continues execution with no knowledge that it moved, but the memory reference is now local. No software was needed anywhere in the path. Which GC core on the target node receives an incoming thread is irrelevant, as all such cores on a node have equal access to any local memory.

The heavy multi-threading also permits such architectures to cheaply spawn new child threads that can go their own way, again without much software involvement. This allows inexpensive large-scale asynchronous parallelism.

In addition the current architecture also includes the beginnings of memory-based accelerators in the form of smart *Memory Side Processors* (MSP) where many latency sensitive memory operations (especially atomics) can be performed directly at the memory interface. Further the ability to perform such operations from a distant node without even moving the source thread state is provided by the introduction of *ultra-lightweight* special purpose threads that are hardwired to perform specific operations without need for program intervention, but use the same migration mechanism as normal threads. Once spawned, they become independent and perform the designated operation against the designated memory location, wherever it is, again without software involvement. A simple acknowledgement system allows the spawning thread to determine when all such child threads have completed.

Two such systems produced by Lucata Inc. can be found in Georgia Tech’s CRNCH Center [22]. The system used for

these experiments is the most recent one called Pathfinder. This system has 16 nodes, each with 16 cores and multiple memory channels. A RapidIO network interconnects the nodes, and carries the migrating thread states. All the logic for each node (the 16 cores, memory controllers, MSP logic, and network interfaces) are implemented in a single large FPGA, with a core clock rate of 225MHz (limited by the FPGA). The cores are all single-issue simple designs where the multi-threading allows simple pipelining to provide very high utilization of the core logic, and especially the memory interfaces. The design is such that a re-implementation of a node should fit comfortably into a modern ASIC, with more cores and a much higher clock rate (comparable to a conventional microprocessor).

An introduction to the programming tools for this machine can be found at [23].

B. Migrating Thread Programming Model

This architecture supports a very Cilk-like programming model where the underlying hardware handles threads that must move from one physical node to another without any explicit software. While syntatically close to Cilk, the architecture essentially removes the need for software support for both a multi-threaded runtime on a single node, and any explicit software needed to recognize that data is "not local".

The implementation of the graph search on the Pathfinder platform followed closely that of the conventional codes discussed in Section II, but without the need for a special thread management runtime, and with an intrinsic library that allowed direct access to the unique hardware mechanisms. Data layout is like that of the SHAD implementations in that a hashmap for each vertex type and a multimap for each edge type are partitioned over the multiple physical nodes in the system, with graph objects stored on the node to which they are hashed. However, with the Pathfinder system, the two-level hash is only needed for map operations where a node must be determined from the key, such as arbitrary lookup operations. Local insertions and lookups need only a 1-step hash to a bucket within the node; the migrating threads implementation takes advantage of this locality to simplify most hashmap operations.

In fact, the OpenCilk implementation discussed earlier was actually a simple source modification from the migrating thread code (with a linking in of the threading runtimes not needed in the migrating thread architecture).

V. Comparison

Fig. 3 graphs the measured times for each of the implementations, all executing exactly the same dataset that contained 38 million edges. The x-axis is the number of nodes used in the run (The Cilk implementation ran on just a single multi-core node). For all runs the solid line represents the actual time, and the dashed line an "Amdahl strong scaling model" fitted to the data[4].

[4]This model was constructed by choosing two points and computing two coefficients A and B where $Time(p) = A + B/p$ where p is the number of nodes.

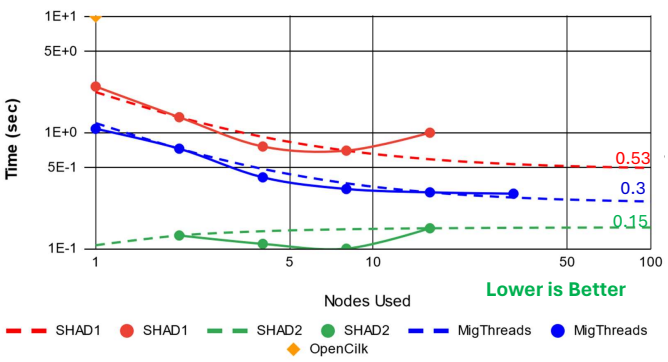

Fig. 3. Execution Time.

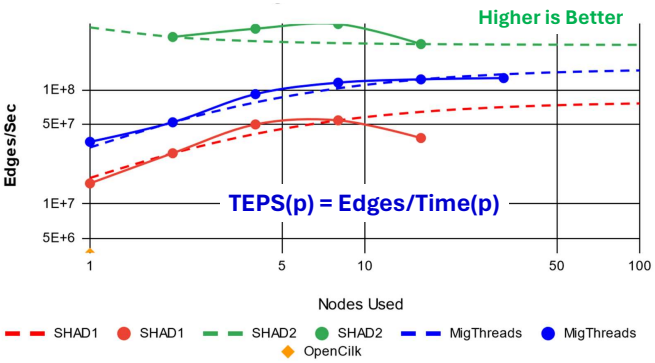

Fig. 4. Traversed Edges per Second.

As can be seen, the OpenCilk times were far outside any of the other implementations. The SHAD-2 implementation (on the older but mature platform) was the fastest but exhibited essentially no parallel speedup. The SHAD-1 implementation (on the faster and more modern platform but with an immature SHAD implementation) did exhibit some parallelism but was significantly slower than the SHAD-2. Both SHAD implementations also exhibited an uprise in execution time at 16 nodes - something that signals a significant scalability issue.

The migrating thread run falls in between the two SHAD implementations, but is fit well by an Amdahl model. Besides the smoothness of the match, what is remarkable about this implementation is that, as shown in Table I, the cores in the Pathfinder platform are considerably slower, by about 10X in clock rate and perhaps another 4X in the issue rate in instructions per cycle. This is due to the implementation technology: FPGAs for the Pathfinder and commercial full custom chips for the others.

Graph benchmark results are often reported in equivalent "Traversed Edges per Second"[5]. Some additional serial implementations of this same problem indicate that the inherent complexity of this particular pattern is nearly linear (i.e. execution time on a single core grows approximately linear in the number of edges - which might be expected from the

[5]See for example "Breadth First Search" (BFS) in the Graph500 benchmarks, or the triangle counting in the MIT Graph Challenges.

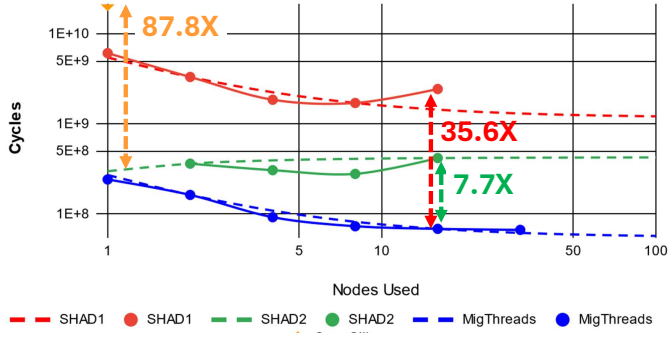

Fig. 5. Timing in terms of processor clock cycles.

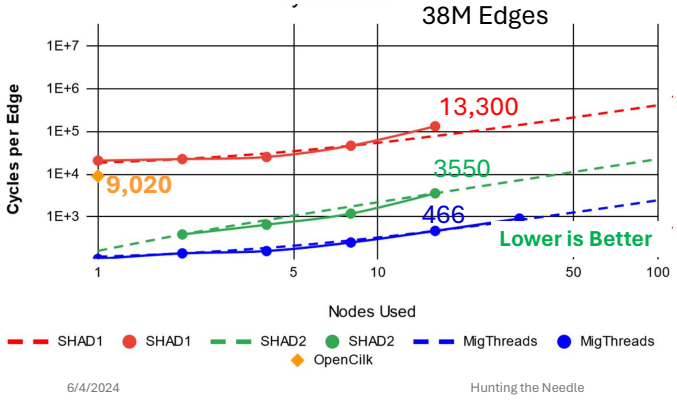

Fig. 6. Compute Cycles per Edge.

tree-like structure of the pattern). Consequently, we can recast the time data into TEPS by dividing the number of edges in the graph by execution time. Fig. 4 does this for the 38M edges of the reference data set.

Trying to draw conclusions from the above two figures is a bit of apples to oranges, because of the significant differences in technology. To try to normalize the difference Fig. 5 converts the time measurements of Fig. 3 into clock units rather than seconds, using the clock rate of each system's core clock. Now the story flips dramatically. The migrating thread case is almost 8X faster than SHAD-2, and almost 100X faster than the OpenCilk system. The migrating thread system takes far fewer clock cycles than the others.

Fig. 6 goes one step further and converts Fig. 4 into "Compute cycles per Edge," where one "compute cycle" is the amount of computation that one core can perform in one clock cycle. Total compute cycles is the overall number of clock cycles needed by the whole system to complete the computation, times the number of cores per node, times the number of nodes. This is a particularly useful metric in that it gives insight into something relatable directly to the problem, namely on average how much overall computing is needed to handle the processing associated with an edge. The numbers in color on this figure are the number of compute cycles when 16 nodes are employed.

VI. Analysis

Fig. 6 allows developing some strong insight into the software overhead suffered in the conventional implementations. Let us assume reasonably that the 466 compute cycles per TEP for the migrating thread case is a reasonable estimate[6] of the minimal number of cycles needed by any 16-node implementation (with migrating threads there is no software overhead on handling inter-node issues). Consequently, it is not unreasonable to allocate the difference between the cycles per TEP for the other implementations to the software costs they must take on to manage any inter-node process (or even just checking if an object is local or not). For the fast SHAD-2 this difference is $3550 - 466 = 3084$ cycles, or 6.6 times the cost of doing the computation.

One step deeper can be taken in this analysis by looking at the number of migrations (or the equivalent) that must be taken per edge. Instrumentation in the Pathfinder hardware allows a direct measurement of this, with a result that for this problem there were 0.42 migrations per edge. This means that the 3084 cycle overhead is per 0.42 migrations, which implies that the actual cost of a single migration in conventional systems is more like $3084/0.42 = 7342$ cycles or 15.6X the cycles needed for computation. The other systems are even worse.

VII. Conclusion

The above results are strong evidence that if we are to significantly improve the performance of computers when performing graph problems and the like where data often straddles nodes, we absolutely must focus on getting software totally out of the path of determining and handling locality of data. The Pathfinder architecture is an indication that this is indeed possible, with a potential savings of around 90% in compute resources.

As additional evidence, other studies [24], [25], [26], [27], [28], [29] have found that such combinations of migrating thread architecture and Cilk-like programming models can significantly improve scalability of parallel codes of all sorts, from streaming to machine learning problems, with the highest advantages again coming when the data sets are large and sparse, where there is a lot of irregularity in access patterns, and computation against individual datums is short. Problems of the latter types, as exhibited here, are particularly good targets for such architectures, as the ability to have huge numbers of threads time-share the same physical cores greatly increases the utilization of such cores, in addition to also avoiding execution of code to "move" the computation.

A logical next step is to repeat this effort on even more problems, or even just the subgraph isomorphism problem but on more complex patterns where complexity is higher. Assuming that such results hold true, then we ought consider "right-sizing" our node designs to not over provision other processing resources such as memory bandwidth or network

[6]The equivalent for the conventional systems may be lower becuase their cores have higher IPC.

injection bandwidth.[7] Such savings could then also greatly improve the energy efficiency of such designs.

ACKNOWLEDGMENT

This research is based upon work supported in part by the Univ. of Notre Dame and in part by the Office of the Director of National Intelligence (ODNI), Intelligence Advanced Research Projects Activity (IARPA), through the Advanced Graphical Intelligence Logical Computing Environment (AGILE) research program, under Army Research Office (ARO) contract number W911NF22C0083. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the ODNI, IARPA, or the U.S. Government.

The authors would also like to thank the personnel at Pacific Northwest National Labs (PNNL) who, through packages such as SHAD, have pushed graph and similar processing as far as possible on conventional architectures.

REFERENCES

[1] W. S. Song, J. Kepner, V. Gleyzer, H. T. Nguyen, and J. I. Krame, "Novel graph processor architecture," in *LINCOLN LABORATORY JOURNAL*, vol. 20, 2013.

[2] W. Harrod, "Advanced Graphical Intelligence Logical Computing Environment (AGILE)." https://www.google.com/url?sa=t&source=web&rct=j&opi=89978449&url=https://www.iarpa.gov/images/PropsersDayPDFs/AGILE/AGILE_-_ISC_2022_Harrod_Updated.pdf&ved=2ahUKEwiS977Rkr-FAxVWIDQIHZKoAF4QFnoECA8QAQ&usg=AOvVaw03eGqkrg01ujfarvVx3-QO, June 2022.

[3] MIT, "Challenges." MIT, 2024. http://graphchallenge.mit.edu/challenges.

[4] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner, "Static graph challenge: Subgraph isomorphism," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, 2017.

[5] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," *National security agency technical report*, vol. 16, no. 3.1, pp. 1–29, 2008.

[6] F. Zhao and A. K. H. Tung, "Large scale cohesive subgraphs discovery for social network visual analysis," *Proc. VLDB Endow.*, vol. 6, p. 85–96, dec 2012.

[7] A. Pavan, K. Tangwongsan, S. Tirthapura, and K.-L. Wu, "Counting and sampling triangles from a graph stream," *Proc. VLDB Endow.*, vol. 6, p. 1870–1881, sep 2013.

[8] A. Azad, A. Buluç, and J. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, pp. 804–811, 2015.

[9] Y. Shao, L. Chen, and B. Cui, "Efficient cohesive subgraphs detection in parallel," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, (New York, NY, USA), p. 613–624, Association for Computing Machinery, 2014.

[10] P. Burkhardt, "Graphing trillions of triangles," *Information Visualization*, vol. 16, 09 2016.

[11] L. Wang, Y. Wang, C. Yang, and J. D. Owens, "A comparative study on exact triangle counting algorithms on the gpu," in *Proceedings of the ACM Workshop on High Performance Graph Processing*, HPGP '16, (New York, NY, USA), p. 1–8, Association for Computing Machinery, 2016.

[12] IARPA, "AGILE Program Workflows." https://www.iarpa.gov/images/PropsersDayPDFs/AGILE/AGILE_Program_Workflows_FINAL.pdf, 2022.

[7]Other measurements from the Pathfinder system reveals that this problem is neither memory nor network bound on either conventional or the Pathfinder systems.

[13] V. G. Castellana and M. Minutoli, "Shad: The scalable high-performance algorithms and data-structures library," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 442–451, 2018.

[14] A. Morari, O. Villa, A. Tumeo, D. Chavarría-Miranda, and M. Valero, "GMT: Enabling Easy Development and Efficient Execution of Irregular Applications on Commodity Clusters," *UPCommons*, 2013.

[15] C. Pheatt, "Intel® threading building blocks," *J. Comput. Sci. Coll.*, vol. 23, p. 298, apr 2008.

[16] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, "Cilk: An efficient multithreaded runtime system," in *Proc. of the Fifth ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPOPP '95, (New York, NY, USA), p. 207–216, Association for Computing Machinery, 1995.

[17] C. Leiserson and A. Plaat, "Programming parallel applications in cilk," *Siam news*, 07 1997.

[18] T. B. Schardl and I.-T. A. Lee, "Opencilk: A modular and extensible software infrastructure for fast task-parallel code," in *Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming*, PPoPP '23, (New York, NY, USA), p. 189–203, Association for Computing Machinery, 2023.

[19] P. M. Kogge, "Of piglets and threadlets: Architectures for self-contained, mobile, memory programming," *Innovative Architecture for Future Generation High-Performance Processors and Systems*, pp. 130–138, Jan. 2004.

[20] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. Brockman, K. Jacobsen, Y. Juan, S. Kuntz, R. Lethin, J. McMahon, C. Pawar, M. Perrigo, S. Rucker, J. Ruttenberg, M. Ruttenberg, and S. Stein, "Highly scalable near memory processing with migrating threads on the emu system architecture," in *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*, pp. 2–9, 2016.

[21] T. Ungerer, B. Robič, and J. Šilc, "A survey of processors with explicit multithreading," *ACM Comput. Surv.*, vol. 35, pp. 29–63, Mar. 2003.

[22] "Center for Research into Novel Compute Hierarchies," 2023. https://crnch-rg.cc.gatech.edu/near-memory-and-in-memory// [Accessed: (11/11/23)].

[23] G. Tech, "Lucata Pathfinder Getting Started." Georgia Tech, 2023. https://gt-crnch-rg.readthedocs.io/en/main/lucata/lucata-getting-started.html.

[24] B. A. Page and P. M. Kogge, "Deluge: Achieving superior efficiency, throughput, and scalability with actor based streaming on migrating threads," in *2021 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, 2021.

[25] B. A. Page and P. M. Kogge, "Scalability of hybrid sparse matrix dense vector (SpMV) multiplication," in *2018 International Conference on High Performance Computing and Simulation (HPCS)*, pp. 406–414, 2018.

[26] B. A. Page and P. M. Kogge, "Scalability of sparse matrix dense vector multiply (SpMV) on a migrating thread architecture," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 483–488, 2020.

[27] B. A. Page and P. M. Kogge, "Scalability of hybrid SpMV with hypergraph partitioning and vertex delegation for communication avoidance," in *Int. Conf. on High Perf. Computing and Simulation (HPCS 2020)*, March. 2021.

[28] B. A. Page and P. M. Kogge, "Scalability of streaming on migrating threads," in *2020 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–8, 2020.

[29] P. M. Kogge and S. K. Kuntz, "A case for migrating execution for irregular applications," in *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*, IA3'17, (New York, NY, USA), Association for Computing Machinery, 2017.