SAND2015-3893C

A Task-Based Linear Algebra Building Blocks Approach for Scalable Graph Analytics

Michael M. Wolf, Jonathan W. Berry, and Dylan T. Stark
Sandia National Laboratories
Albuquerque, NM 87185
{mmwolf,jberry,dstark}@sandia.gov

***Abstract*—It is challenging to obtain scalable HPC performance on real applications, especially for data science applications with irregular memory access and computation patterns. To drive co-design efforts in architecture, system, and application design, we are developing miniapps representative of data science workloads. These in turn stress the state of the art in Graph BLAS-like Graph Algorithm Building Blocks (GABB). In this work, we outline a Graph BLAS-like, linear algebra based approach to miniTri, one such miniapp. We describe a task-based prototype implementation and give initial scalability results.**

I. Introduction/Background

A. miniTri: a Data Analytics Miniapp

Application proxies or miniapps are important driving forces in the architecture-system-application co-design efforts that attempt to ensure good performance for real applications on modern processors and supercomputing systems. They represent a compromise between the simplicity of kernel benchmarks and the complexity of real applications. There are many scientific computing miniapps in Mantevo [1], for example, but few have been written with data science in mind. The *miniTri* miniapp described below was conceived in unpublished work at Sandia to help fill that gap. It will become part of the Mantevo shortly.

HPC data analytics benchmarking applications have been represented over the years by SSCA-2 [2] and Graph500 [3]. These similar benchmarks emphasize graph generation and search. The miniTri miniapp is not based on neighbor set expansion, and offers different challenges than traditional graph search.

The kernel operation within the miniapp is triangle enumeration (not triangle counting). This can be approached several fundamentally different ways, though describing them all is beyond the scope of this paper. The authors of [4] found triangle structure to be a key ingredient present in real data, and [5] provides one of many examples leveraging this structure to improve community detection.

Given a graph, let $t_v(t_e)$ be the number of triangles incident on Vertex v (resp. Edge e). We also call these triangle degrees. Such degrees can be computed by enumerating all triangles in the graph. Such enumeration is usually practical because the global numbers of triangles in large networks is still asymptotically linear for realistic degree distributions [6], despite their local structure. Storing all triangles, however, might overwhelm memory. This last point is a key driver for the fused operations described in Section III.

The objective of the miniTri miniapp is to tabulate "k-counts" over all triangles in this graph. These are numbers related to the largest clique that could contain a given triangle. This k-count of a triangle t is defined to be:

$$\arg\max_k \{(\min_{v\in t} t_v \geq \binom{k-1}{2}) \cap (\min_{e\in t} t_e \geq k-2)\}.$$

Note that any vertex of a k-clique will be incident on $\binom{k-1}{2}$ triangles of that clique, and that any edge will be incident on $k-2$ triangles of that clique. The argmax selects the largest k satisfying these conditions. For example, in Figure 1, the red triangle has k value five and the green triangle k value three. The output of miniTri miniapp is a table of the k-count frequencies. For example, the graph in Figure 1 will result in the following table:

k	1	2	3	4	5
freq	0	0	2	0	10

In words, there are ten triangles with k-count five, and two triangles with k-count three. This table gives us a bound on the largest clique in the graph: it is the largest c such that at least $\binom{c}{3}$ triangles have k-count at least c. In this example, that bound is five.

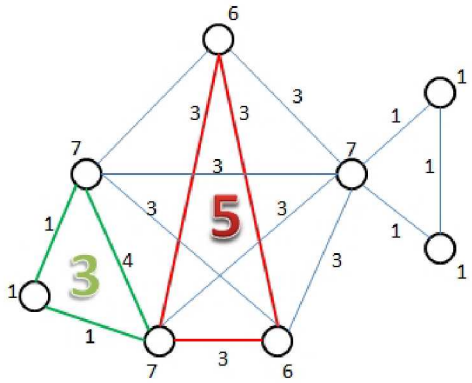

Fig. 1. A graph with vertices and edges labeled with their triangle degrees, and two of the triangles labeled with their k value.

The most obvious way to compute miniTri would be to

perform two passes of triangle enumeration. The first pass would compute t_v and t_e values and the second pass would compute k-counts. These passes are representative of many variations of vertex and edge attribute update and retrieval in social network analysis, where the attributes are related to short cycles.

In this paper, we describe a linear algebra approach to triangle enumeration as part of the larger miniapp, noting that there are several completely different approaches. Our primary contributions are:

- We describe new linear algebraic Graph Algorithm Building Blocks (GABBs) that express concisely not only triangle enumeration, but the computation of all triangle degrees t_v and t_e,
- We describe several challenges with implementing these GABBs on HPC systems,
- We describe how task parallelism can be used to address these challenges, and
- We apply these GABBs to implement miniTri and present preliminary computational results.

B. Graph Algorithms and Linear Algebra Primitives

The success of BLAS and LAPACK for basic linear algebra, combined with strong growth in the fields of graph mining and data science, has led to a community-wide effort to produce Graph Algorithm Building Blocks (GABBs). The most visible outcome of this effort is the developing "Graph BLAS" [7], which promotes linear algebraic building blocks. There are at least three concrete implementations of Graph BLAS: CombBLAS [8], D4M [9], and Graphulo. GABBs within Graph BLAS libraries are designed to be abstract, high-level constructs. They typically generalize traditional matrix and vector operations. Two primary considerations are conciseness and high-performance. Graph analysts who could effectively use Graph BLAS to implement solutions would be quite productive, and would benefit from decades of linear algebra performance work.

The primary alternatives to Graph BLAS are the related communities of high-performance generic graph libraries and cloud-based graph libraries. The former class is represennted by the Boost Graph Library (BGL) [10], MultiThreaded Graph Library (MTGL) [11], and the Parallel Boost Graph Library (PBGL) [12] and its underlying AM++ (active message) layer [13] that enables asynchronous computation. These all support the visitor pattern, allowing programmers to customize vertex, edge, and structural access (for example, triangle visitors for triangle enumeration) without re-implementing any performance-tuned code. Cloud-based solutions such as GraphLab [14] and PowerGraph [15] offer similar granularity in their GABBs, which are typically vertex-centric API's. Such low-level GABBs have, to date, been more flexible than Graph BLAS in the sense that strange, unanticipated computations are possible without redesigning components of the underlying system. As a community, we have been working to bridge the gap betweeen these low-level GABBs and the Graph BLAS. This paper is part of that effort, as is [16]. The leap from triangle counting to triangle enumeration GABBs in the latter was encouraged by this author team.

Sandia is proposing miniTri as a representative of the strange, unanticipated computations mentioned above. Social network analsis gives rise to innumerable computations like miniTri that might be based upon attributes like t_v and t_e. We believe, therefore, that miniTri will stress Graph BLAS in useful ways. We show in this paper how to extend Graph BLAS to accommodate miniTri concisely, increasing the flexibility of the approach.

We believe that linear algebra based GABBs show great promise, but we anticipate challenges ahead. Before they become more mainstream, we believe that several advances are necessary. These include GABB implementations that allow the computations to proceed in highly asynchronous and load-balanced manner, mechanisms for tolerating memory latency, a mechanism for limiting the explosion of state (high memory watermark), and a systematic approach for fusing linear algebra operations. Azad, et al. [16] begin considering the last point, but more generality is still required. In this paper we propose an approach to address the first three requirements and to mitigate the fourth to a lesser degree.

C. Task Parallelism

The importance of dynamic and adaptive task parallel models is being driven by changes in system architecture. A combination of homogeneous, heavy-weight compute cores, relatively flat memory hierarchy, and compute-intensive bulk-synchronous algorithms has favored static, balanced *data parallel* models, exemplified by many MPI and OpenMP applications. Such data parallel models minimize overheads at the expense of flexibility. Unfortunately, the end of Dennard scaling, flatlining of clock rates, increasing node-level parallelism, and decreasing memory per core all contribute to radically different emerging node architectures. New architectures will have many more lighter-weight compute cores and extended multi-level memory hierarchies. Furthermore, part-to-part performance variability will introduce heterogeneous performance characteristics that will reduce the effectiveness of bulk-synchronous algorithms and data parallel threading models, increasing the need for more dynamic and adaptive parallelism.

These new architectures provide advantages and disadvantages for algorithm and application developers, especially in the high-performance data analytics space. Increased node-level parallelism can be used to tolerate high-latency operations to (different levels of) memory or over the network. This has been demonstrated with the XMT line of massively-multithreaded architectures [17]. However, this will require programmers to express sufficient parallelism in their code and the threading model to handle large numbers of potentially short-lived threads. This increase in application parallelism raises issues with scheduling work for locality, balancing time-varying and data-driven workloads, and dynamically managing precious resources (e.g., memory and network). There are a variety of task parallel models and runtime

systems that support these requirements, such as Cilk [18], OpenMP [19], Qthreads [20], High Performance ParalleX (HPX) [21], Grappa [22], and even MPI+X [23], when X is an on-node task parallel runtime. Our work leverages models that support continuations (i.e., stopping and restarting tasks across long-latency events). Our implementations have used Qthreads and HPX.

II. Linear Algebra Based miniTri

A. Algorithm

Our graph analytics miniapp miniTri (presented in subsection I-A) can be formulated in terms of linear algebra in a Graph BLAS-like manner as seen in Figure 2. In the first significant step of this algorithm (line 3), the triangles in this graph are enumerated using an overloaded sparse matrix-matrix multiplication operation (SpGEMM) of the graph adjacency matrix (A) and graph incidence matrix (B). The multiplication is overloaded such that $C(i,j) = (i,x,y)$ iff $A(i,x) = A(i,y) = 1$ and $B(x,j) = B(y,j) = 1$ (by construction $B(*,j) = 0$ for other elements in this column). Otherwise, $C(i,j) = \emptyset$. The intuition behind this operation is that each row x in the adjacency matrix implicitly stores all graph wedges (triplets of vertices v_i-x-v_j connected by a path of length two) with midpoint x and v_i,v_j being all pairs of the nonzeros column numbers in row x. By multiplying those pairs by columns of the incidence matrix, we are attempting to find edges that connect the endpoint of the wedges, and thus complete the triangle. When successful (the two row numbers of nonzeros in $B(*,j)$ correspond to column numbers in row $A(i,*)$), the triangle is completed. It is important to note that this operation enumerates each triangle three times ($C = LB$ where L is the lower triangle portion of A would enumerate each triangle once) and similar linear algebra formulations of triangle enumeration have been proposed [16]. However, this formulation (by construction) enumerates each triangle once for each vertex (rows in C) and each edge (columns in C), which greatly simplifies the triangle vertex and edge degree computations. The overcounting of triangle attributes such as k-counts can be addressed by multiplying the counts by one-third.

1: **procedure** MINITRI(A, B)
2: $\quad \mathbf{k_{cnts}} \leftarrow 0$
3: $\quad C = A \cdot B$ ▷ Enumerate triangles
4: $\quad t_v = C \cdot \mathbf{1}$ ▷ Calculate triangle vertex degree
5: $\quad t_e = C^T \cdot \mathbf{1}$ ▷ Calculate triangle edge degree
6: $\quad$ **for** all $t = C(i,j) \neq \emptyset$ **do** ▷ Compute k for triangles
7: $\quad\quad k_1 = \arg\max_k \{\min_{v \in t} t_v(v) \geq \binom{k-1}{2}\}$
8: $\quad\quad k_2 = \arg\max_k \{\min_{e \in t} t_e(e) \geq k-2\}$
9: $\quad\quad k = \min(k_1, k_2)$
10: $\quad\quad \mathbf{k_{cnts}}(k) = \mathbf{k_{cnts}}(k) + \frac{1}{3}$
11: $\quad$ **end for**
12: **end procedure**

Fig. 2. Linear Algebra Formulation of miniTri.

Next, we calculate the vertex and edge triangle degrees by multiplying matrix C (which contains the triangles) and C^T, respectively, by vectors of ones (lines 4 and 5). This is equivalent to counting the number of nonzeros in each row to obtain the triangle vertex degrees and the number of nonzeros in each column to obtain the triangle edge degrees. As previously explained, each triangle is represented in C once for each edge and each vertex, which is why these row and column nonzeros counts correspond to the triangle vertex and edge degrees. Finally, we loop over each triangle (three times) stored in C and compute its k value using the triangle vertex and edge degrees. We tabulate the counts of the k values with $\frac{1}{3}$ ($\frac{1}{3}$ to account for the overcounting) being added to a particular k value's tally each time that triangle $C(i,j)$ has that k value.

For simplicity, we have presented this algorithm at an abstract level, leaving out many implementation details. It is important to note that this algorithm can be implemented with standard linear algebra data structures (with integer elements).

B. Worked Example

In this subsection, we walk through an example (graph shown in Fig. 3(a)) to better illustrate this linear algebra based algorithm. The adjacency matrix A and incidence matrix B for this graph are:

$$A = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 \end{pmatrix}$$

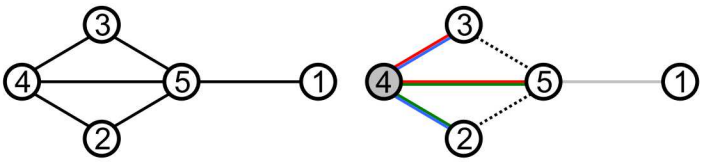

Fig. 3. (a) Undirected graph, G. (b) Wedges in G that contain 4 as midpoint: 2-4-3 (blue), 2-4-5 (green), 3-4-5(red).

Let us look at row 4 of the adjacency matrix. Row 4 has nonzeros in columns 2,3,5, which implicitly corresponds to the wedges: 2-4-3, 2-4-5, and 3-4-5 (see Fig. 3(b)).

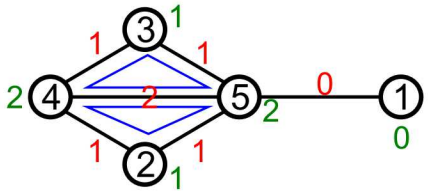

Fig. 4. Triangle vertex and edge degrees for G.

Multiplying A and B together to form C, we get an enumer-

ation of the triangles:

$$C = \begin{pmatrix} \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & (2,4,5) \\ \emptyset & \emptyset & \emptyset & \emptyset & \emptyset & (3,4,5) \\ \emptyset & (4,2,5) & (4,3,5) & \emptyset & \emptyset & \emptyset \\ \emptyset & \emptyset & \emptyset & (5,3,4) & (5,2,4) & \emptyset \end{pmatrix}$$

Multiplying row 4 of the adjacency matrix by the columns in the incidence matrix, we find that edge 2-5 (column 2 in B) completes the 2-4-5 wedge to form the (2,4,5) triangle and edge 3-5 (column 3 in B) completes the 2-3-5 wedge to form the (3,4,5) triangle (dotted lines in Fig. 3(b) correspond to these edges). In the next step, we multiply C and C^T by vectors of ones to obtain the triangle vertex and edge degrees:

$$t_v = C \cdot 1 = (\ 0 \quad 1 \quad 1 \quad 2 \quad 2\ )^T$$
$$t_e = C^T \cdot 1 = (\ 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 2\ )^T$$

We see that these match the degree labels (green and red for vertex and edge degrees, respectively) on Fig. 4. Finally, we calculate the k values for each triangle in C and tabulate the counts of each value.

C. Challenges with Linear Algebra Based Approach

This approach is quite concise, as we can express most of the miniTri miniapp in three lines of code (lines 3-5 in Fig. 2). However, there are still several challenges related to parallel computation. In particular, the use of conventional, bulk-synchronous parallel computation would fundamentally limit performance in irregular computations. Fig. 5 shows a basic outline of such a conventional data parallel approach, where typically the graph data and work is divided up into a number of parts equal (or similar to) the number of processing cores. Barriers halt progress between each pair of linear algebra kernels, leading cores to suffer from starvation and become idle during irregular computation. Load-balancing is a well-known problem in graph computations, whether implemented with linear algebraic GABBs or not.

For multithreaded data parallel approaches, work stealing can help within kernels but not between kernels. It is possible to remove global barriers from these approaches, but not without significant effort and complication of the library APIs.

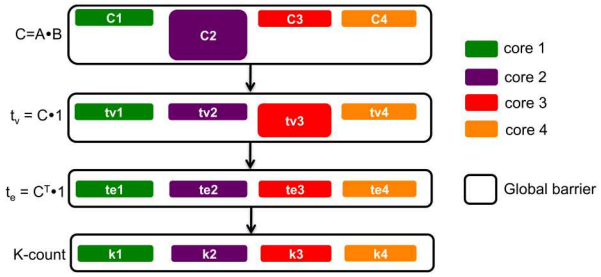

Fig. 5. Data flow of linear algebra based implementation on miniTri using traditional data parallel paradigms.

One additional serious difficulty for applying high-level linear algebra based approaches to data-centric analytics is that completely storing an intermediate result such as C may overwhelm memory, and may not even be necessary. The miniTri computation, for example, does not require storing all triangles. During enumeration, a triangle can be seen once and discarded after t_v and t_e updates are completed. The difficulty is that linear algebra based implementations with traditional parallel paradigms would do just that (in Fig 2, every triangle would be stored three times when the C matrix is constructed), severely limiting the size of solvable problems.

III. Task Parallelism and Linear Algebra miniTri

A. Task Parallel Approach

Task parallelism can be used in a linear algebra based approach to address the difficulties outlined in the previous section. Our proposed approach should yield significantly better performance (e.g., through latency hiding) and allow us to solve larger problems (by using resource-constrained scheduling of tasks to mitigate the explosion of state intrinsic to complex analytics such as triangle enumeration).

Fig. 6 shows a simplified example of this task-parallel approach to a linear algebra-based miniTri. A key aspect of this approach is the overdecomposition of each linear algebra kernel into many light-weight tasks such that for sufficiently large graphs the number of tasks is much greater than the number of computational cores that we are targeting. This overdecomposition allows the scheduler flexibility to address starvation and hide high remote memory latency by switching out tasks that are waiting for memory or IO requests. This approach also provides load-balancing support by allowing work to be moved from busy cores to idle cores. Although there is overhead for switching tasks, schedulers currently can be effective with this approach for medium grain tasks on shared memory regions. It is likely that schedulers will continue to improve performance (especially with assistance in hardware) for finer grain tasks and over multiple shared memory localities. Instead of explicitly specifying communication as in an MPI+X approach, dependences between the tasks are specified, which leads to a more natural expression of the data-centric computations. These dependencies are represented in Fig. 6 as the arrows between tasks (only a small number of dependencies are shown here for visual clarity).

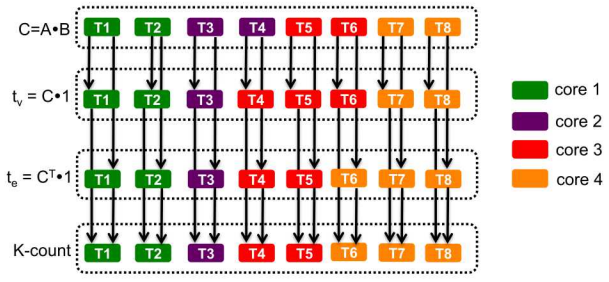

Fig. 6. Illustrative example of task parallel approach to linear algebra based miniTri. Arrows represent dependencies between tasks.

Another key aspect of task parallelism is that there are no global barriers in this approach to the miniTri miniapp.

Instead, there are fine-grain synchronization points based on the dependencies of the tasks. This allows progress to be made across the linear algebra kernel boundaries, allowing tasks in subsequent kernels to run (or even complete) before all of the tasks in the earlier kernels complete. This should help the performance for the miniTri miniapp and should be even more significant for larger graph applications that have numerous parallel sparse linear algebra kernels.

The lack of global barriers in this approach also allows us to address the severe size limitation that this Graph BLAS-like linear algebra approach imposes on the miniTri miniapp when implemented in a data parallel manner. The key is to use resource-constrained schedulers that prioritize tasks in order to optimize memory management. In the context of miniTri, tasks that compute the triangle degrees and k-count are prioritized over the triangle enumeration tasks. This leads to sets of the triangles being enumerated (blocks of C being computed) in order to fulfill the dependencies of the the computation of blocks of elements in t_v and t_e, which are needed to compute k. Once these k are computed the corresponding triangles can then be freed from C. Through this tight coordination between the linear algebra kernels and dynamic, adaptive runtime system capabilities for resource-constrained problems, we can enable effective resource management that allows us to execute miniTri on much larger graphs (with the memory footprint being closer to the size of the graph than to the enumerated set of triangles). In general, this task parallel approach enables Graph BLAS-like building blocks to obtain both more flexibility and scalable performance.

B. Preliminary Results

The focus of this work has been on the formulation of an efficient linear algebra based approach to miniTri, and we have developed several implementations of this approach, including a serial reference implementation, an OpenMP multi-threaded implementation, and an MPI based implementation. In addition to these baseline implementations, we have implemented two task-parallel variants of the linear algebra based approach: one HPX-based implementation and one Kokkos [24] and Qthreads based implementation. Our current implementations exploit task parallelism (as previously described) but do not yet include the advanced resource management for optimizing memory usage.

Figs. 7 and 8 show some preliminary numerical experiments, comparing our HPX-based implementation (using HPX version 0.9.10) of miniTri with a baseline data parallel OpenMP implementation for the SNAP/soc-sign-epinions graph (131k vertices, 841k edges) obtained from the University of Florida Matrix Collection [25]. The implementations were compiled using gcc version 4.8.3, and the experiments were run on a 16 core single shared memory node (dual processor Intel Xeon E5-2630, @2.40GHz). In general, our task parallel approach to miniTri shows slight improvement over the data parallel approach. Both approaches show significantly less than ideal speedup for 8 and 16 threads, with the task parallel approach showing slightly better speedup for 16 threads than the data parallel approach. We saw no improvement when running on 32 threads.

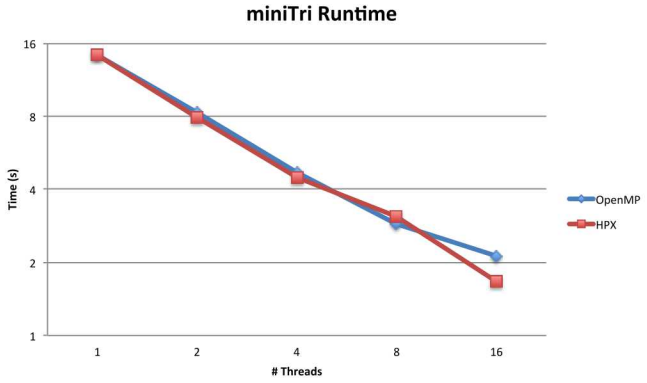

Fig. 7. miniTri runtime for HPX and OpenMP linear algebra based implementations.

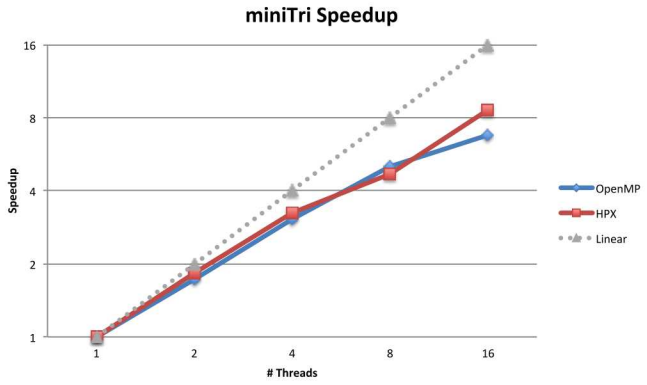

Fig. 8. miniTri speedup for HPX and OpenMP linear algebra based implementations.

IV. Summary/Conclusions

Linear algebra based building blocks such as those proposed by the Graph BLAS forum show great promise to become a useful standard for developing high performing data analytics applications. In this paper, we presented such a linear algebra-based approach to miniTri, a data analytics miniapp that has been developed as part of the Mantevo project with application to network analysis.

There are several parallel performance challenges related to this approach, including load imbalance and high remote memory access costs, both which can greatly decrease the parallel efficiency especially when multiple kernels are strung together to form applications. An equally serious challenge is the explosion of state, which can occur when using these linear algebra constructs in graph computations and greatly limit the size of the solvable problems as we saw with the triangle portion of miniTri. Although low-level graph analysis building blocks can address this problem by explicitly fusing several kernels and operating on a small portion of the graph

(thus constraining the memory usage), this is more difficult to accomplish with the high-level linear algebra building blocks, at least with conventional parallel programming paradigms.

In this paper, we proposed a task-based parallel approach to address these problems. By overdecomposing the problem into many tasks and replacing global barriers between kernels with finer grain synchronization points, better load-balance for these very irregular computations can be achieved and the latency for expensive remote memory accesses can be hidden. Removal of global barriers between the linear algebra kernels allows us significantly more flexibility in regards to what tasks can be scheduled. By prioritizing tasks that have the capability to free memory, we see a path forward through task parallelism to address the problem size limit that is imposed by traditional programming paradigms. This added flexibility should allow users to solve larger problems with this linear algebra based framework. The hope is that with the added flexibility and performance provided by this approach will help Graph BLAS become an extremely productive tool for graph analysts.

ACKNOWLEDGMENT

We thank John Gilbert (UCSB), Adam Lugowski (UCSB), and Aydın Buluç (LBNL) for helpful discussions about triangle counting and enumeration algorithms as well as suggesting related triangle enumeration algorithms. This work was supported by the Office of Advanced Scientific Computer Research, Office of Science, U.S. Department of Energy, as part of the XPRESS Project in the 2012 X-Stack Research Program. The research presented in this paper was also funded through the Laboratory Directed Research and Development (LDRD) program at Sandia National Laboratories, in the context of the Extreme Scale Grand Challenge (XGC) and the User-Accessible Unified Manycore Performance-Portable Programming Model Projects. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

REFERENCES

[1] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," *Sandia National Laboratories, Tech. Rep*, 2009.

[2] "SSCA#2 v2.1 Specification," http://www.graphanalysis.org/benchmark/, 2007.

[3] R. C. Murphy, K. B. Wheeler, B. W. Barrett, and J. A. Ang, "Introducing the graph 500," *Cray Users Group (CUG)*, 2010.

[4] T. G. Kolda, A. Pinar, T. Plantenga, and C. Seshadhri, "A scalable generative graph model with community structure," *SIAM Journal on Scientific Computing*, vol. 36, no. 5, pp. C424–C452, 2014.

[5] J. W. Berry, B. Hendrickson, R. A. LaViolette, and C. A. Phillips, "Tolerating the community detection resolution limit with edge weighting," *Physical Review E*, vol. 83, no. 5, p. 056119, 2011.

[6] J. W. Berry, L. K. Fostvedt, D. J. Nordman, C. A. Phillips, C. Seshadhri, and A. G. Wilson, "Why do simple algorithms for triangle enumeration work in the real world?" in *Proceedings of the 5th conference on Innovations in theoretical computer science*. ACM, 2014, pp. 225–234.

[7] Tim Mattson et al., "Standards for graph algorithm primitives," in *Proc. IEEE High Performance Extreme Comp. Conf.*, 2013.

[8] A. Buluç and J. R. Gilbert, "The Combinatorial BLAS: Design, implementation, and applications," *The International Journal of High Performance Computing Applications*, vol. 25, no. 4, pp. 496 – 509, 2011.

[9] J. Kepner, W. Arcand, W. Bergeron, N. T. Bliss, R. Bond, C. Byun, G. Condon, K. Gregson, M. Hubbell, J. Kurz, A. McCabe, P. Michaleas, A. Prout, A. Reuther, A. Rosa, and C. Yee, "Dynamic distributed dimensional data model (d4m) database and computation system." in *ICASSP*. IEEE, 2012, pp. 5349–5352.

[10] J. G. Siek, L.-Q. Lee, and A. Lumsdaine, *Boost Graph Library: User Guide and Reference Manual, The*. Pearson Education, 2001.

[11] J. W. Berry, B. Hendrickson, S. Kahan, and P. Konecny, "Software and algorithms for graph queries on multithreaded architectures," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*. IEEE, 2007, pp. 1–14.

[12] D. Gregor and A. Lumsdaine, "The parallel bgl: A generic library for distributed graph computations," *Parallel Object-Oriented Scientific Computing (POOSC)*, vol. 2, pp. 1–18, 2005.

[13] J. J. Willcock, T. Hoefler, N. G. Edmonds, and A. Lumsdaine, "Am++: A generalized active message framework," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*. ACM, 2010, pp. 401–410.

[14] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "Graphlab: A new framework for parallel machine learning," *CoRR*, vol. abs/1006.4990, 2010.

[15] J. E. Gonzalez, Y. Low, H. Gu, D. Bickson, and C. Guestrin, "Powergraph: Distributed graph-parallel computation on natural graphs." in *OSDI*, vol. 12, no. 1, 2012, p. 2.

[16] A. Azad, A. Buluç, and J. R. Gilbert, "Parallel triangle counting and enumeration using matrix algebra," in *Proceedings of the IPDPSW, Workshop on Graph Algorithm Building Blocks (GABB)*, 2015.

[17] J. Feo, D. Harper, S. Kahan, and P. Konecny, "Eldorado," in *Proceedings of the 2nd Conference on Computing Frontiers*, ser. CF '05. New York, NY, USA: ACM, 2005, pp. 28–34. [Online]. Available: http://doi.acm.org/10.1145/1062261.1062268

[18] 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 *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95. New York, NY, USA: ACM Press, 1995, pp. 207–216.

[19] *OpenMP Application Program Interface*, 2nd ed., OpenMP Architecture Review Board, May 2008.

[20] K. B. Wheeler, R. C. Murphy, and D. Thain, "Qthreads: An API for programming with millions of lightweight threads," in *Proceedings of the 22nd International Symposium on Parallel and Distributed Processing*, ser. IPDPS/MTAAP. IEEE Computer Society Press, April 2008, pp. 1–8.

[21] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14. New York, NY, USA: ACM, 2014, pp. 6:1–6:11.

[22] J. Nelson, B. Myers, A. H. Hunter, P. Briggs, L. Ceze, C. Ebeling, D. Grossman, S. Kahan, and M. Oskin, "Crunching large graphs with commodity processors."

[23] M. P. I. Forum, "MPI: A message passing interface standard," *International Journal of Supercomputer Applications (Special Issue on MPI)*, vol. 8, no. 3/4, 1994.

[24] H. C. Edwards, C. R. Trott, and D. Sunderland, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, pp. 3202–3216, 2014.

[25] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.