Final Report

Advanced Scientific Computing Research
Office of Science, U.S. Department of Energy
DOE# SN10051 DE-SC0012471

Data Structures for Extreme Scale Computing

Simon Kahan, PhD
skahan@cs.washington.edu
206 383 1653

August 18, 2017

DUNS# 605799469
University of Washington, Seattle WA
9/1/2014 - 6/30/2017

ACCOMPLISHMENTS

Goals and Objectives

The primary project objective was to support -- or invalidate -- the hypothesis that Aggregating Data Structures (ADS) can overcome the challenges posed by conventional data structures at exascale. The implications for exascale computation are foundational: parallel applications, libraries, and system software all depend upon shared data representations; whether managed implicitly via remote access or explicitly via messages, they encounter these challenges at scale. Success would suggest that scaling up non-trivial or “irregular” exascale applications is possible using ADS.

A secondary objective was to establish the beneficial role of latency tolerant runtimes in enabling ADS. The scalability of ADS is not without cost: ADS optimizes throughput at the expense of higher worst-case response time. All else being equal, longer latencies could degrade application performance. However, we believe that a latency tolerant runtime system can fully mitigate this deficiency of ADS, providing higher application performance by decreasing sensitivity to latency. In combination with a positive result from our primary objective, achieving our second objective would suggest that latency tolerant runtime systems combined with ADS are a potent combination towards scaling up irregular exascale applications.

Achieving the first and second objectives would represent merely a proof of concept: a third objective is to pave the way for more research in ADS and latency tolerant runtime systems in the exascale agenda. The space of data structures is vast. A great deal more work would be needed to provide well designed ADS even for just the most common data structure use cases, and the impetus to do so would be as great as that to prove that real applications can in fact achieve exascale performance.

Specific Accomplishments

Activity

Our primary activity during the past reporting period was to focus on an application of ADS and the latency tolerant runtime system, Grappa[1], to establish potential of the approach on an important application that is notoriously difficult to scale up: LU factorization of the irregular sparse linear systems that arise in analysis of electronic circuits. Such circuit matrices are included in Tim Davis’ sparse matrix collection. Even with preconditioning, factorization of these matrices tends not to form dense blocks during fill-in, so the ratio of floating point operations to memory operations remains low throughout most of the factorization process as does the efficiency of parallelization strategies that depend upon formation of blocks to express large-grained parallel work. Even though parallelism is abundant in theory, its granularity is fine, making it difficult to exploit especially on distributed memory systems.

Objective

Our specific objective was ambitious: to develop a linear solver for distributed memory systems that would out-perform state-of-the-art solvers in factoring these matrices. State-of-the-art solvers we

[1] Nelson, J., Holt, B., Myers, B., Briggs, P., Ceze, L., Kahan, S., & Oskin, M. (n.d.). Latency-Tolerant Software Distributed Shared Memory. Retrieved from http://sampa.cs.washington.edu/papers/grappa-usenix-2015.pdf

compared against were SuperLU_Dist[2], KLU[3], and NICSLU[4]. SuperLU_Dist is a general purpose sparse linear solver package for distributed memory systems that relies on the formation of large blocks during the factorization process to provide spatial locality. Factoring sparse linear systems associated with circuits often does not lead to the formation of large blocks. KLU was subsequently developed to address this deficiency and is used in Sandia's Xyce[5] circuit analysis application. However, KLU is not parallel, so will not scale up to multiple cores let alone multiple nodes. NICSLU was developed in response to KLU as a shared-memory parallel solver and applies a hybrid of left-looking, pipelining, and sequential algorithms to outperform both KLU and SuperLU_Dist on the circuit matrices of interest. Our objective, then, was to outperform NICSLU.

Approach
Towards achieving our objective, we developed a naïve numerical LU factorization algorithm we call GrappaLU. We leveraged a couple of existing sparse linear algebra methods used also in the other solvers. MC64 was used to mitigate the need to pivot during factorization. AMD was used to minimize fill. Both algorithms are efficient enough to run sequentially, though for large matrices, they, too, would need to be parallelized. Doing so is beyond the scope of this work; and is not done in SuperLU_Dist or in NICSLU. We also leveraged the Judy Array, an efficient implementation of dynamic arrays GrappaLU accesses in a sequential context. Beyond that, GrappaLU is unconventionally simple.

We distribute the columns of the matrix to cores in an interlaced fashion. That is, the j'th column is assigned to the (j mod #cores)th core. Each column is represented by a Judy Array resident entirely within the memory space associated with the corresponding core by the Grappa runtime system. This allows efficient location of the element by row index and efficient handling of the inevitable fill-in as the matrix is factored. Because we are using dynamic arrays, we omit the entire symbolic factorization phase typically used to pre-allocate the memory space for fill. As in all ADS approaches, we rely on the latency tolerant runtime, in this case Grappa, to mitigate the impact of non-local references.

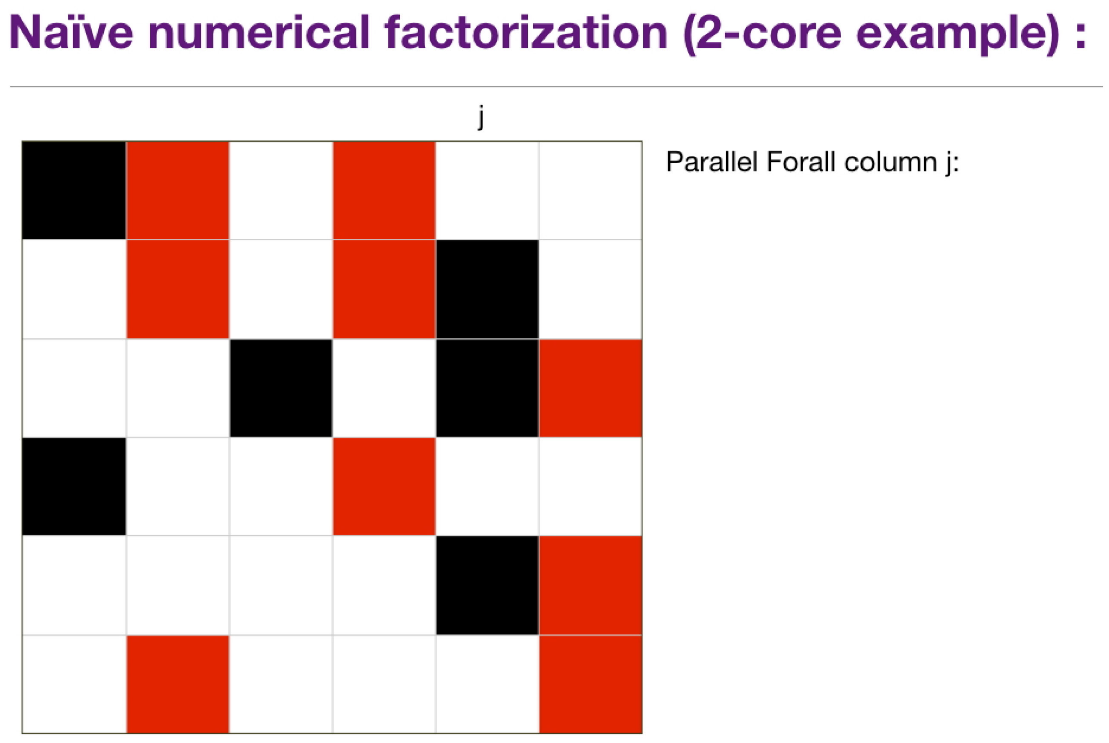
[2] Li, X. S., Demmel, J. W., Gilbert, J. R., Grigori, L., Shao, M., & Yamazaki, I. (n.d.). SuperLU Users' Guide. Retrieved from http://crd-legacy.lbl.gov/~xiaoye/SuperLU/superlu_ug.pdf

[3] Davis, T. A., & Palamadai Natarajan, E. (2010). Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems. *ACM Transactions on Mathematical Software. Association for Computing Machinery*, *37*(3), 36.

[4] Chen, X., Wang, Y., & Yang, H. (2013). NICSLU: An Adaptive Sparse Matrix Solver for Parallel Circuit Simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, *32*(2), 261–274.

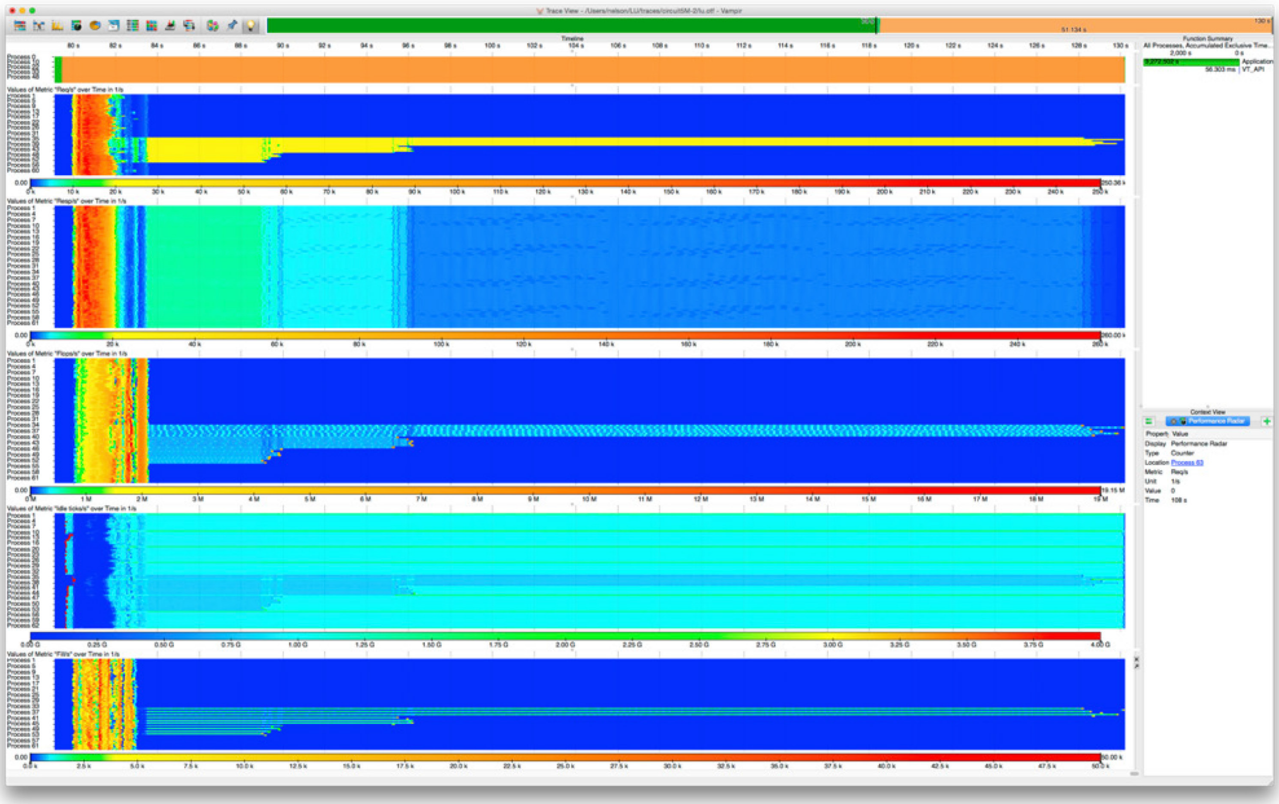
[5] Hutchinson, S., Keiter, E., Hoekstra, R., Watts, H., Waters, A., Russo, T., … Bogdan, C. (2002). THE Xyce PARALLEL ELECTRONIC SIMULATOR AN OVERVIEW. In *Parallel Computing* (pp. 165–172). PUBLISHED BY IMPERIAL COLLEGE PRESS AND DISTRIBUTED BY WORLD SCIENTIFIC PUBLISHING CO.

Execution on two computing cores of a single elimination in the GrappaLU factorization process is illustrated in the animation below (In Microsoft Word, you should be able to double click to run; else **tinyurl.com/y9cf63zg**). Concurrently, every core marches through its assigned columns from left to right. The non-zero elements of each column j above the diagonal are eliminated sequentially, top to bottom: in Grappa, the core owning column j makes a request to the core owning column i for the

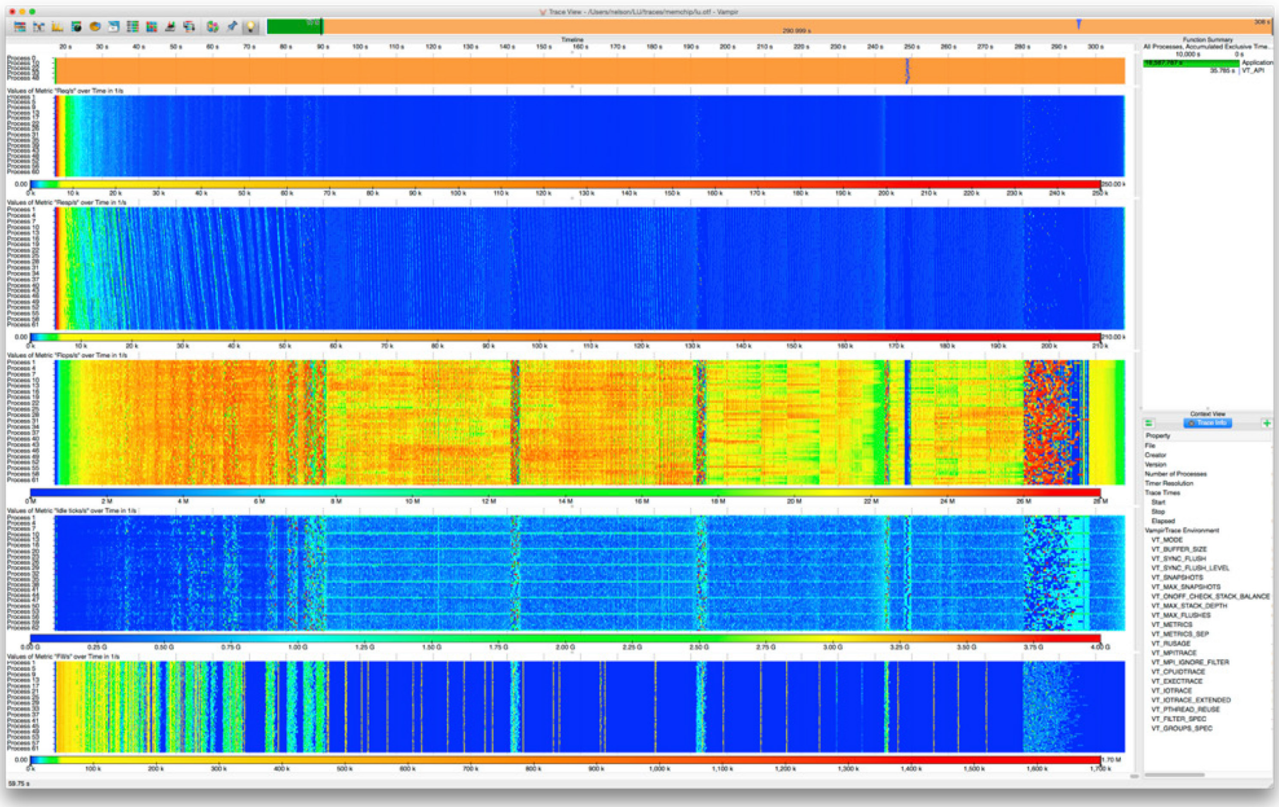

Naïve numerical factorization (2-core example) :
j
Parallel Forall column j:

entire column i. The request is non-blocking: after making the request, the core is free to move along to process other columns it owns. The core receiving the request sends column i once all elements above the diagonal element (i,i) have been eliminated. Upon receipt, column i is applied to column j to eliminate (i,j) and the next non-zero in column j is similarly processed.

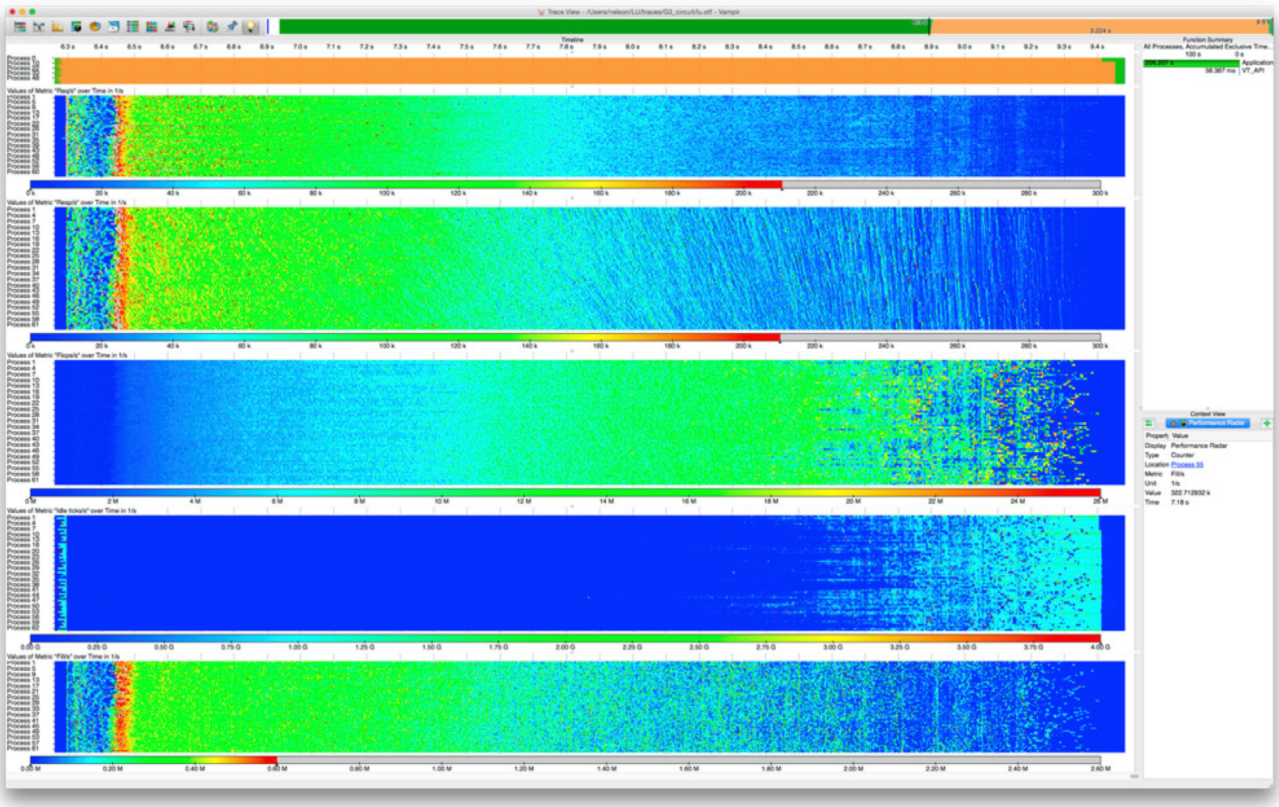
Results
We consider here the behavior on three matrices from Tim Davis' collection: circuit_5M, memchip, and G3_circuit.



Circuit_5M has 5.5M rows and 59M non-zeroes. In this barely readable trace produced by Grappa, the horizontal axis is time and the vertical is a sampling of some of the cores on a re are 6 graphs, each with the traces for a sampling of cores (to each core is bound a process so we think of them as one and the same). The 2nd through 6th of these graphs are of interest: message request rate for the core; message response rate; flop rate; idle rate; and fill rate. The scale is below each graph, dark blue being zero up to bright red being some maximal value. What we see is that even with the fine-grained parallelism expressed, it nonetheless dies down suddenly. There's no hope for scaling on this matrix. Even NICSLU tops out at about 2x speedup no matter how many cores are applied.



memchip has 2.7M rows and 13M non-zeroes. In contrast to Circuit_5M, there is plenty of parallelism as evidenced by the 4th graph down that colorfully displays high flop rate across most cores. The high flop rate relative to communication makes life easy for any of the algorithms. NICLU achieves 6x speedup with 12 cores (remember, NICSLU runs only on shared-m



emory systems).

G3_circuit has 1.6M rows and 7.7M non-zeroes. Here there is plenty of parallelism as in memchip but much lower flop rates, much more sustained communication. This is more like what we expect in exascale for circuit matrices: communication becomes increasingly dominant as computing systems grow.

Our runs were all on a Xeon system:

SuperLU_Dist

Nodes	Cores	R	C	SYMBFACT	Numerical FACTOR	TOTAL FACTOR
1	12	3	4	1.05	5.85	6.9
2	12	4	6	1.14	4.52	5.7
4	12	6	8	1.13	3.64	4.8
8	12	8	12	1.13	3.29	**4.4**

KLU

We did not run KLU ourselves. The NICSLU reports **761** seconds on a Xeon faster than ours.

NICSLU

Sequential factorization time for NICSLU: **3.9** seconds
12-core factorization time: 6.9 seconds

GrappaLU

Nodes	FACTOR
1	13

2	7.2
4	4.1
8	2.9
12	**2.6**

GrappaLU is a dead-simple parallel algorithm that should be extremely *inefficient* from a spatial locality perspective and from an operational count perspective. In fact, it is much slower factoring the first two matrices than the other approaches – we have not even bothered to report the results, as we still have ideas for reducing the operational count. However, despite its simplicity, GrappaLU is faster factoring G3_circuit than these other direct LU factorization methods.

The reason that GrappaLU is faster on G3_circuit is the high ratio of communication needed per unit of work. The need to mitigate communication costs becomes paramount. GrappaLU expresses the available fine-grained per-column parallelism and Grappa exploits that parallelism to tolerate communication latency and to reduce communication overheads by context switching and using its internal ADS to aggregate small communications into larger ones.

We expect on Exascale systems that the communication work to coordinate parallel activities and to share spatially disparate data will be even higher relative to floating point and local operations performed. As Grappa is designed for distributed multicore systems, adapting it to Exascale systems should be feasible. Meanwhile, more exploration on larger systems of Grappa (or other latency tolerant runtimes) and of high-concurrency, less locality sensitive algorithms like GrappaLU designed to exploit Grappa seems warranted.

Training/Professional Development
Supported one postdoc 2015-2016. We have nothing specific to report on training and development.

Dissemination of Results
At the SIAM Conference for Parallel Processing 2016 early results were disseminated.

Plans for Next Reporting Period
This is a final report.

PRODUCTS

Code for GrappaLU is available at https://github.com/simonkahan/LU and can be run on a system with Grappa available from www.grappa.io

Nelson, J., Myers, B., Holt, B., Lee, V. Halperin, D., Howe, B., Briggs, P., Ceze, L., Kahan, S., & Oskin, M. (2016). Grappa: enabling next-generation analytics tools via latency–tolerant distributed shared memory. PGAS Poster, *Supercomputing 2016.*

Kahan, S. (2016). *A systems view of high-performance computing*. **Invited Presentation**, SIAM Conference on Parallel Processing for Scientific Computing, Paris, France.

Nelson, J., Holt, B., Myers, B., Briggs, P., Ceze, L., Kahan, S., & Oskin, M. (2015). Latency-Tolerant Software Distributed Shared Memory. In *USENIX Annual Technical Conference* (pp. 291–305). **(Best Paper Award)**

PARTICIPANTS

The grant largely supported Jacob Nelson as a postdoctoral researcher at the University of Washington who was the lead developer of Grappa, software for scaling data intensive applications expressed as distributed memory programs on commodity clusters.

IMPACT

As computing problems of national importance grow, the government meets the increased demand by funding the development of ever larger systems. The overarching goal of the work supported in part by this grant is to increase efficiency of programming and performing computations on these large computing systems.

In past work, we have demonstrated that some of these computations once thought to require expensive hardware designs and/or complex, special-purpose programming may be executed efficiently on low-cost commodity cluster computing systems using a general-purpose "latency-tolerant" programming framework.

One important developed application of the ideas underlying this framework is graph database technology supporting social network pattern matching used by US intelligence agencies to more quickly identify potential terrorist threats. This database application has been spun out by the Pacific Northwest National Laboratory, a Department of Energy Laboratory, into a commercial start-up, Trovares Inc.

We explore an alternative application of the same underlying ideas to a well-studied challenge arising in engineering: solving unstructured sparse linear equations. Solving these equations is key to predicting the behavior of large electronic circuits before they are fabricated. Predicting that behavior ahead of fabrication means that designs can optimized and errors corrected ahead of the expense of manufacture.

In addition to its research, defense, and commercial contributions, our work contributes also to the education of young scientists. This grant supported a postdoctoral fellow in the Computer Science & Engineering department at the University of Washington who in turn guided graduate students in their related research.

CHANGES/PROBLEMS

The original research proposal was aimed at supporting a graduate student interested in studying the scalability of latency-tolerant aggregating data structures in comparison to other parallel data structure designs. Unfortunately, no student stepped up to take on this challenge directly. The funds were used instead to support a postdoc responsible for the continued development of Grappa, the latency-tolerant distributed memory platform upon which those data structures would be built, and part-time work by the PI in developing a sparse linear solver exploiting Grappa.