# Scalability of Sparse Matrix Dense Vector Multiply (SpMV) on a Migrating Thread Architecture

Brian A. Page
*Dept. of Computer Science and Engineering*
*University of Notre Dame*
Notre Dame, IN USA
bpage1@nd.edu

Peter M. Kogge
*Dept. of Computer Science and Engineering*
*University of Notre Dame*
Notre Dame, IN USA
kogge@nd.edu

***Abstract*—Sparse matrix dense vector multiplication (SpMV), exhibits the memory bandwidth and communication driven nature of many sparse linear algebra operations. Irregular memory accesses from the non-zero structure within a sparse matrix wreak havoc on performance. This paper presents strong scaling for communication avoiding SpMV implementations on a migrating thread system intended to address the lack of locality in sparse problems. We developed communication avoiding SpMV code to attempt to reduce off-node thread migration by using the hypergraph partitioning package HYPE to determine workload distribution. Additionally, we investigate the performance impact of overlapping communication and computation through the use of remote memory operations supported by the architecture. Incorporating remote memory operations with hypergraph partitioning we achieved 6.18X speedup for overall performance.**

***Index Terms*—Emerging Architectures, migrating threads, irregular applications, communication overhead, HPC**

## I. Introduction

Sparse problems such as sparse matrix dense vector multiplication (SpMV) are used extensively in many applications. For instance, SpMV constitutes the bulk of the High Performance Conjugate Gradient (HPCG) [5] code that has become an alternative to LINPACK for rating supercomputers. It is also used extensively in linear solvers such as HYPRE [7], and finite element method applications such as PGFem3D [12], [13]. While dense linear algebra problems have received considerable attention, similar work on sparse problems remains a field rife for improvement.

Several earlier studies have shown that sparse problems, SpMV in particular, exhibit highly irregular memory access patterns [14]–[16]. Due to such irregularity, when strong scaling of SpMV is performed on conventional hybrid or heterogeneous systems the communication overhead associated with inter-process updates eliminates any speedup from adding additional computational capability. We have observed that using accelerators such as Intel Knights Landing as well as GPUs can reduce computational time requirements. However communication with current network interconnects is still several orders of magnitude slower. Fig. 1 from [15] shows the impact communication has on overall performance for SpMV on distributed systems. Even with efforts to reduce or eliminate communication, any remaining communication requirements outpace speedup from strong scaling.

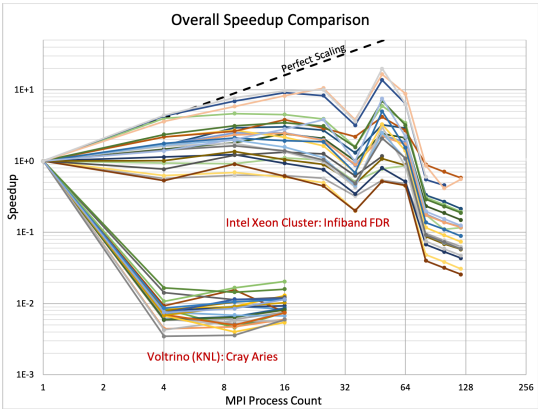



Fig. 1: Overall SpMV Performance on Conventional Architectures

Data reuse and limited memory bandwidth drive the scant performance seen in sparse problems. The new class of **Migrating Thread** architectures [6], [8] provides a means for moving work rather than data throughout the system, and thus avoid using software to perform remote operations. In this study we implemented several SpMV codes on this new architecture so that we could observe and analyze strong scaling. We performed partitioning on our benchmark matrices by using the HYPE partitioner [11], as discussed in greater detail in Sec. V-A. Scaling results are compared against a naive distribution in which non-zeros are placed round-robin across all nodelets. For comparison we also developed an implementation of SpMV using remote memory operations.

Our contributions are two-fold: the use of migrating threads seems to enable better scaling than any observed in our prior studies, and the use of remote memory operations enabled increased computational performance by overlapping memory accesses with computation. In doing so we saw up to 6.18 speedup using remote operations when no data partitioning scheme was used, and 6.15 with hypergraph partitioning.

The remainder of this paper is organized as follows: Section II provides relevant background. Section III overviews related work. Section IV describes the sparse matrices used in our experiments. Section V overviews hypergraph partitioning for SpMV, workload balancing and distribution, and experimental setup of our SpMV implementations. Finally Section VI eval-

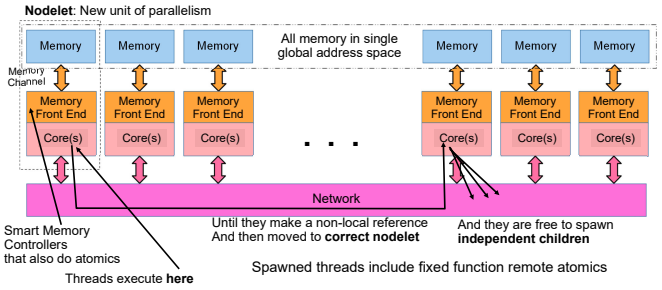



Fig. 2: The Migrating Thread Architecture.

uates the scalablity observations and compares against related works, and Section VII concludes.

Sections II and IV are modifications of similar discussions in our prior studies on different architectures. However, the migrating thread implementation discussed here is new.

## II. Background

### A. SpMV Overview

The simplest sequential implementation of SpMV utilizes two nested loops of which the innermost processes one non-zero at a time from a row in matrix $A$. The matching memory required is $O(nnz + 2m)$, where $m$ is the matrix row dimension.

Performing iterative SpMV in a distributed environment adds additional complexities such as communication overhead which are tied to the workload partitioning of rows as well as the non-zero structure of the sparse matrix. A sparse matrix $A$ is multiplied by a dense vector $x$, with its row results being placed into the result vector $b$. In many applications such as HPCG [10], SpMV is called iteratively on the same $A$, with each result vector reused in some way for the $x$ in the next iteration. In this paper we assume that at some point each row result must be sent to all processing elements that require it to update their dense vectors for the next iteration.

SpMV on a migrating thread system benefits from a shared address space and does not need to send updates throughout the system so that all computations have the latest results after each iteration. Instead once a final row result has been computed, every thread has access to that value by performing a direct access. Thread migration may be required in order to perform this access, however the hardware freely migrates the thread, requiring no explicit control by the programmer.

### B. The Migrating Thread Architecture

A migrating thread architecture [8] has features that help with all these issues. All memory is in a single shared address space where any thread in any physical core can have load/store access to any of these locations. What is different from a conventional shared memory architecture is that when an access is made to a non-local location, the underlying hardware, not software, actually *moves* the state of a thread to a core close to that memory as required during execution.

Fig. 2 diagrams such an architecture as implemented by Emu Solutions [6]. The basic unit, a **nodelet**, is a memory module, its memory controller, and some number of multi-threaded cores. All memory in the collection resides in a common address space. A network connects all nodelets. A thread runs in a core until it makes a memory reference that is not contained in that nodelet's memory. The hardware then puts the thread to sleep, packages it, and moves it over the network to the correct nodelet, and unpacks and restarts it. There is no core-specific data cache. All memory accesses go back to the appropriate memory controller.

Each nodelet's memory controller includes computation logic that allows it to perform atomic memory operations against locations within its memory. The instruction set of a migrating thread includes a rich set of operations that utilize this atomic capability in addition to simple loads, and stores.

A thread can cheaply spawn additional threads who then live existences independently of the parent. These threads can be of several types. First are full-fledged threads capable of executing arbitrary programs. Second are special purpose threads that can only perform some dedicated operations such as fetch-and-op. These latter thread types work directly with the memory controllers at the target nodelet to perform remote atomic operations without moving the whole thread state.

The current prototype has up to 64 nodelets packaged 8 to a board, each with 8GB of memory, a RapidIO-based network, and a dual core POWER microprocessor on each board to run Linux, manage a local SSD, and initiate migrating threads into the system. The memory bus for each nodelet delivers 8 bytes per access (rather than 64 bytes as on conventional systems), meaning that for problems with a high percentage of memory accesses that have little spatial locality[1], the usable bandwidth from them approaches $100\%$. The nodelet logic on each board is implemented via an FPGA. The programming tool chain is based on Cilk, with a prefix to function calls to spawn new threads, a sync primitive to wait for a set of children to complete, and a parallel forall to have a set of independent threads cooperate on a loop. Supported intrinsics include a rich set of atomic operations.

A larger system using the same boards is under development, with additional enhancements such as caches in front of memory[2]. Looking further out, the architecture is an excellent match to 3D memory stacks where the nodelet logic is on a logic chip on the bottom of the stack, and there may be literally dozens of nodelets in each stack.

## III. Related Work

SpMV is known to be a memory bound problem, suffering from poor data reuse and limited memory bandwidth on conventional systems. A previous study developed an analytic projection of potential SpMV performance enhancements using migrating threads [9]. A later study investigated the impact migrating threads may have on bandwidth and thread execution location within the EMU system [17]. Rolinger and

[1]This means for example that out of the data line accessed from memory only a small percentage is used

[2]These caches are in the path for all accesses to the memory they hold so that no coherency traffic is needed between cores on the nodelets

Krieger utilized several data placement techniques, including cyclic and block, to partition their benchmark matrices onto nodelets within the system. They observed that techniques such as METIS and BFS for data reordering saw speedup of only 16% on conventional systems while obtaining up to 70% improvement on EMU.

One important note from this second study was the observation that even random ordering of data across nodelets would achieve superior performance over no reordering at all. As stated in [18] this contradicts traditional behavior observed on conventional architectures in which randomization often dramatically reduces performance by increasing cache misses and their associated penalties.

Partitioning of data among distributed systems is a highly studied area and one that can dramatically increase overall performance when done correctly for the given system and problem. In [17], [18] Rolinger et al attempt to optimize data layout of sparse matrices on EMU nodelets. A 50% increase in performance was obtained by implementing a block-based data layout scheme in preparation for computing SpMV on the benchmark matrices selected in their study. A cost model for the current FPGA based implementation of the EMU migrating thread architecture was developed and compared to observed speedup obtained on an 8 node (64 nodelet) EMU system.

## IV. Matrix Benchmark Suite

Similar to our previous study we have chosen 25 matrices from the Suite Sparse Matrix Collection[3] [3]. Emphasis was on matrices with either extreme sparsity or irregular patterns. Table I lists their characteristics. Matrices were selected based on their average non-zeros per row $nnz_{row}$, as well as overall non-zeros. At least two matrices from each $nnz$ range were chosen with similar size or $nnz$ per row but different structure. Additionally, matrices with significantly different structure were chosen. For example, the non-zeros in *atmosmodd* cluster along the main diagonal, whereas *parabolic_fem* has a wider dispersion. We focused on structural differences to evaluate the impact of matrix structure on communication volume and message size across different load balancing methods. Additionally we chose matrices in a quasi logarithmic fashion to ensure a wide spectrum of sparsities.

Not only were these matrices selected using this rational, but also because these matrices provide a wide range of characteristics while still being small enough to perform rapid file I/O and load balancing for each test. We are aware of much larger matrices which must be evaluated on distributed systems due to their immense size requirements. However evaluating such matrices is often cumbersome due to lengthy matrix read and partitioning times along with increasing cluster allocation sizes. Full scale tests to confirm our findings in this study will be completed but are beyond the scope of this paper.

[3]Currently hosted at https://sparse.tamu.edu/

TABLE I: BENCHMARK MATRIX SUITE

| matrix | rows | nnz | nnz % | nnz row |
|---|---|---|---|---|
| atmosmodd | 1270432 | 8814880 | 5.46E-06 | 6.93 |
| parabolic_fem | 525825 | 3674625 | 1.33E-05 | 6.98 |
| rajat30 | 643994 | 6174244 | 1.49E-05 | 9.58 |
| CurlCurl_3 | 1219574 | 13544618 | 9.11E-06 | 11.10 |
| offshore | 259789 | 4242673 | 6.29E-05 | 16.33 |
| Fem_3D_thermal2 | 147900 | 3489300 | 1.60E-04 | 23.59 |
| nlpkkt80 | 1062400 | 28192672 | 2.50E-05 | 26.53 |
| CO | 221119 | 7666057 | 1.57E-04 | 34.66 |
| gsm_106857 | 589446 | 21758924 | 6.26E-05 | 36.91 |
| msdoor | 415863 | 19173163 | 1.11E-04 | 46.10 |
| bmw3_2 | 227632 | 11288630 | 2.18E-04 | 49.59 |
| BenElechi1 | 245874 | 13150496 | 2.10E-04 | 53.48 |
| t3dh | 79171 | 4352105 | 6.94E-04 | 54.97 |
| F2 | 71505 | 4294285 | 8.40E-4 | 60.05 |
| consph | 83334 | 6010480 | 8.65E-04 | 72.12 |
| SiO2 | 155331 | 11283503 | 4.68E-04 | 72.64 |
| torso1 | 116158 | 8516500 | 6.31E-04 | 73.31 |
| dielFilterV3real | 1102824 | 89306020 | 7.34E-05 | 80.97 |
| RM07R | 381689 | 37464962 | 2.57E-04 | 98.15 |
| m_t1 | 97578 | 9753570 | 1.02E-03 | 99.95 |
| crankseg_2 | 63838 | 14148858 | 3.47E-03 | 221.63 |
| nd24k | 72000 | 28715634 | 5.54E-03 | 398.82 |
| TSOPF_RS_b2383 | 38120 | 16171169 | 1.11E-02 | 424.21 |
| mouse_gene | 45101 | 28967291 | 1.42E-02 | 642.27 |
| human_gene1 | 22283 | 24669643 | 4.97E-02 | 1107.10 |

## V. Experimental Setup

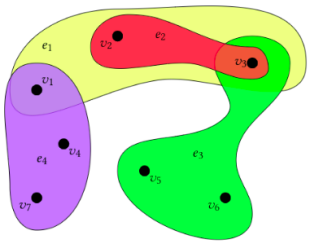
### A. Workload Partitioning with Hypergraphs

The EMU architecture maintains a shared memory address space among the system. As such migrating threads do not require software-driven communication between processing domains. This does not mean that there is **no** communication present. Instead thread migrations perform "communication" by moving the thread state to the nodelet controlling the memory locations it intends to access. Similar to communication, during thread migration useful computation is not taking place and therefore degrading overall latency of computation for that particular computation[4]. Many studies have attempted to reduce communication volume by establishing optimized data placement and have achieved varying degrees of success. We implemented a more generalized approach, using hypergraph partitioning, which is independent of specific matrix characteristics and explores more general approaches that look at the placement of non-zeros without the aid of any apriori knowledge about the matrix.

Hypergraphs are higher dimensional representations of a graph or data set which may reveal exploitable interconnections among data set elements. Hypergraphs contain hyperedges which, unlike traditional graph edges that connect only two vertices, can join any number of vertices. All vertices belonging to the vertex set of a hyperedge share some property as defined by the data set and application in question. Additionally, a vertex may exist in multiple hyperedges when it has properties in common with more than one set of vertices. Formally hypergraph partitioning is the process of finding a partitioning of a hypergraph such that some cost function, such as net cut, or fanout (k-1) is minimized. It is used in

[4]However, unlike conventional architectures, there is no consumption of resources back where the thread left due to a stalled core, thus increasing the useful time available back on that core.

$$A = \begin{bmatrix} 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad A^T = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \end{bmatrix}$$




| Partition | Row Id |
|---|---|
| 1 | 1 2 3 |
| 2 | 5 6 |
| 3 | 1 4 7 |

Fig. 3: Hypergraph Partitioning of a Sparse Matrix

many fields such as VLSI design [1], database storage shard reduction, and distributed graph preprocessing [4].

Any sparse matrix can be treated as a hypergraph by considering an arbitrary $row[i]$ as a hyperedge $h[i]$, and the column id of the non-zeros within that $row[i]$ as vertices belonging to $h[i]$. Partitioning of a hypergraph representation of sparse matrices for use in SpMV have been shown to reduce communication by up to 60%, as it more accurately depicts the communication pattern required by row result updates [2].

Similar to [17] our focus is on avoiding thread migration where possible by implementing optimized data placement techniques. Because of this we chose to evaluate the impact of hypergraph partitioning on overall performance for the matrices included in our benchmark matrix suite. Fig. 3 shows a sample matrix $A$ and its transpose $A^T$. If a hypergraph is generated using $A$ then its rows are treated as hyperedges, with the column ids of the rows' non-zero values treated as the vertices in each hyperedge's vertex set. Therefore the unifying property of each hyperedge's *vertices* is that they all belong to the same row of $A$. Conversely if the transpose of a sparse matrix is used to generate a hypergraph, then the columns become hyperedges and row ids of the non-zeros within each column are the vertices within them. The visual representation of the hypergraph in Fig. 3 illustrates the higher-dimensional and overlapping behavior inherent to hyperedges. Our implementation performs partitioning on hypergraphs generated using the transpose of each benchmark matrix. This was important because we used CSR storage format for local matrix data, which can obtain improved cache performance when all non-zeros of a row are contiguous in memory.

With respect to SpMV, hypergraph partitioning has the potential to reduce interprocess communication by assigning rows which have high column similarity to the same nodelet. Thus the elements requiring that row result in subsequent iterations would often be co-located with the row itself, This enables hypergraph partitioning to often eliminate the need for communication of row results, or allow for reducing the number of partitions to which a result must be distributed if optimal partitioning could not be generated.

The **HYPE** hypergraph partitioner [11] uses neighborhood expansion for efficiently determining optimal vertex assignment. HYPE performs a balanced k-way partitioning by analyzing the vertices within each hyperedge and generating a minimal *core set* of vertices with which to calculate similarity. Additionally HYPE minimizes the K-1 metric, the number of times that neighboring vertices are assigned to different partitions. While HYPE produces balanced partitionings with respect to row (vertex) per partition assignments, the number of non-zeros per partition can vary widely depending on which rows have been assigned. Therefore it is possible that such a hypergraph partitioning may be imbalanced with respect to the computational requirements of the SpMV operation.

In our tests HYPE produced partitions with a hyperedge cut 20%-40% lower than the total row count of each matrix. This indicates that message volume and communication has potentially been reduced by 40% or more, due to reduced fanout of vertex interconnections.

### B. SpMV on Migrating Threads

For this study we developed a migrating thread implementation of SpMV. This implementation is run across select nodelet counts of powers of 2 up to 64 nodelets. Workload distribution uses two methods: hypergraph partitioning or a naive round-robin method. The Hypergraph partitioning for the input matrix and nodelet count is read in from file and checked against non-zeros as they are read in from file. Alternatively the round-robin method stripes non-zeros across all nodelets used in the run such that $non-zero_0$ is placed on nodelet 0, $non-zero_1$ nodelet 1, etc.

Dense vector **x** and result vector **result** are striped across all nodes used in the test. Remember that a thread will migrate any time it attempts to access memory that is not local to the nodelet on which it is current executing. This means that when any arbitrary thread attempts to access elements of the dense vector, non-zero data, or update a row result, a migration may result depending on data placement throughout the system. In this study no attempt was made to have local copies of the **x** or **result** vectors, but instead we allow the hardware the freedom to perform thread migrations among its shared address space.

Non-zero elements are placed on each nodelet according to the distribution method selected. Each nodelet has a local array in which the non-zeros it has been assigned are placed. Threads are spawned on each nodelet and begin working over the locally assigned non-zero elements within this array. This enables threads to access local non-zero data without having to perform a migration. For this study we set the initial thread count for each nodelet to 64, the maximum that can be stored in a nodelet core's thread buffer.

Threads spawned at each nodelet are given an initial id $i$ from 0 to 63. Each thread then computes the partial sum for the $i^{th}$ non-zero element in the nodelet's non-zero array. Once the thread completes computation for each individual non-zero, its id is incremented by the number of threads per nodelet. This process is continued by every thread until all non-zeros assigned to the nodelet have been evaluated. By performing

computation in this manner each non-zero is accessed only once and no locking or access control mechanism is necessary for the non-zero array. During computation a thread may migrate to another nodelet to obtain the dense vector value associated with the column of the non-zero being operated on. A thread that has migrated accesses the desired memory address and then returns to its "home" nodelet where it continues execution.

Rows are updated with the partial sums calculated by the threads. The EMU architecture provides support for remote memory operations such as REMOTE_ADD, REMOTE_AND, REMOTE_XOR. These remote memory operations allows a thread to launch an asynchronous write to a memory address without having to either migrate or wait for an acknowledgement of completion, increasing performance.

Thread migration can be costly as seen in [17]. Therefore we evaluated scalability for both data partitioning methods by allowing thread migration for row updates, and again using the REMOTE_ADD operation to ensure that these updates occurred asynchronously with any remaining computation.

## VI. Performance Evaluation

To show speedup obtained using our SpMV benchmarks we measured the time required to perform any necessary computation and thread migrations. Thread migrations for memory accesses, as well as any latent memory operations which may have been queued for execution within a nodelets memory engine are all included in the timing measurements. Fig. 4 shows the observed speedup for all benchmark matrices used in this study. Each sparse matrix was evaluated with 4 total variations of the SpMV benchmark: naive partitioning (Fig. 4a), hypergraph partitioning (Fig. 4b), naive partitioning using REMOTE_ADDs (Fig. 4c), and lastly hypergraph partitioning with REMOTE_ADDs (Fig. 4d).

As can be seen in Fig. 4 (a) and (c) both implementations of the naive matrix partitioning exhibited nearly identical behavior. Speedups of up to **6.18X** were achieved regardless of the use of remote memory operations for storing row result updates. While good scaling was obtained up to 8 nodelets, performance declined sharply at 16 nodelets, which constitute of two node cards. As observed previously in [17] this is likely due to increases in thread migration latency when migrating between nodes across the network interconnect as opposed to the backplane which links all nodelets inside of a single node. We then see increased speedup as we continued to scale our tests onto 32 nodelets (4 nodes).

Fig. 4 (b) and (d) also illustrates the the high level of similarity between both implementations of hypergraph partitioning as well as hypergraph partitioning when the REMOTE_ADD intrinsic is used for row updates. Overall the behavior is nearly identical with the exception of SiO2 being the only matrix to obtain negative speedup. This occurred at when using 16 nodelets when remote adds are used. As with the naive partitioning method good scaling returns after this initial transition from intra-node to multi-node execution with a maximum achieved speedup of **6.15X**.

While hypergraph partitioning has been shown to reduce communication and therefore increase overall performance in conventional systems, we did not see any substantial gains from its use in our benchmarks. Fig. 5 shows the speedup of hypergraph partitioning with the use of remote memory operations, compared to that of the naive partitioning. For several matrices the hypergraph partitioning method actual degrades performance rather than providing any benefit.

It extremely important to note that while the hypergraph partitioning method is not superior to the naive method in this case, both partitioning methods employed on the EMU migrating thread architecture achieved speedup several orders of magnitude greater than what has been seen on conventional architectures up to this point.

## VII. Conclusion

This study developed a migrating thread implementation of SpMV capable of utilizing hypergraph partitioning in an effort to reduce migrations. We then analyzed the scalablity of our benchmarks across 25 benchmark matrices. For both workload distribution methods employed we saw overall speedup of up to 6.18X. Previous studies which utilize distributed systems requiring communication methods such as MPI struggle to achieve any positive speedup above 1.0 with the average speedup being several orders of magnitude worse than that.

In this study we observed some of the best strong scaling behavior we have seen so far for SpMV. The ability to move work do the data in order to perform computation appears to be a good fit for sparse problems such as SpMV as it allows for the elimination of explicit communication during runtime. However migrating threads still constitute communication in a sense and further research must be performed in order to ascertain to what extent migration overhead can be mitigated, and what the impact on scalablity will be.

## References


[1] C. Ababei, N. Selvakkumaran, K. Bazargan, and G. Karypis. Multi-objective circuit partitioning for cutsize and path-based delay minimization. *Proc. of the 2002 IEEE/ACM Int. Conf. on Computer-aided Design*, pages 181–185, 2002.

[2] U. V. Catalyurek and C. Aykanat. Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. *IEEE Trans. on Parallel and Distributed Systems*, 10(7):673–693, July 1999.

[3] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, Dec. 2011.

[4] K. D. Devine, E. G. Boman, R. T. Heaphy, R. H. Bisseling, and U. V. Catalyurek. Parallel hypergraph partitioning for scientific computing. In *In Proc. IPDPS'06. IEEE*, 2006.

[5] J. Dongarra and M. Heroux. Toward a new metric for ranking high performance computing systems. Sandia Report SAND2013 4744, Sandia National Labs, June 2013.

[6] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. B. 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 *Proc. of the Sixth Workshop on Irregular Applications: Architectures and Algorithms*, IA3 '16, pages 2–9, Piscataway, NJ, USA, Nov. 2016. IEEE Press.

[7] R. D. Falgout and U. M. Yang. hypre: A library of high performance preconditioners. In P. M. A. Sloot, A. G. Hoekstra, C. J. K. Tan, and J. J. Dongarra, editors, *Computational Science — ICCS 2002*, pages 632–641, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

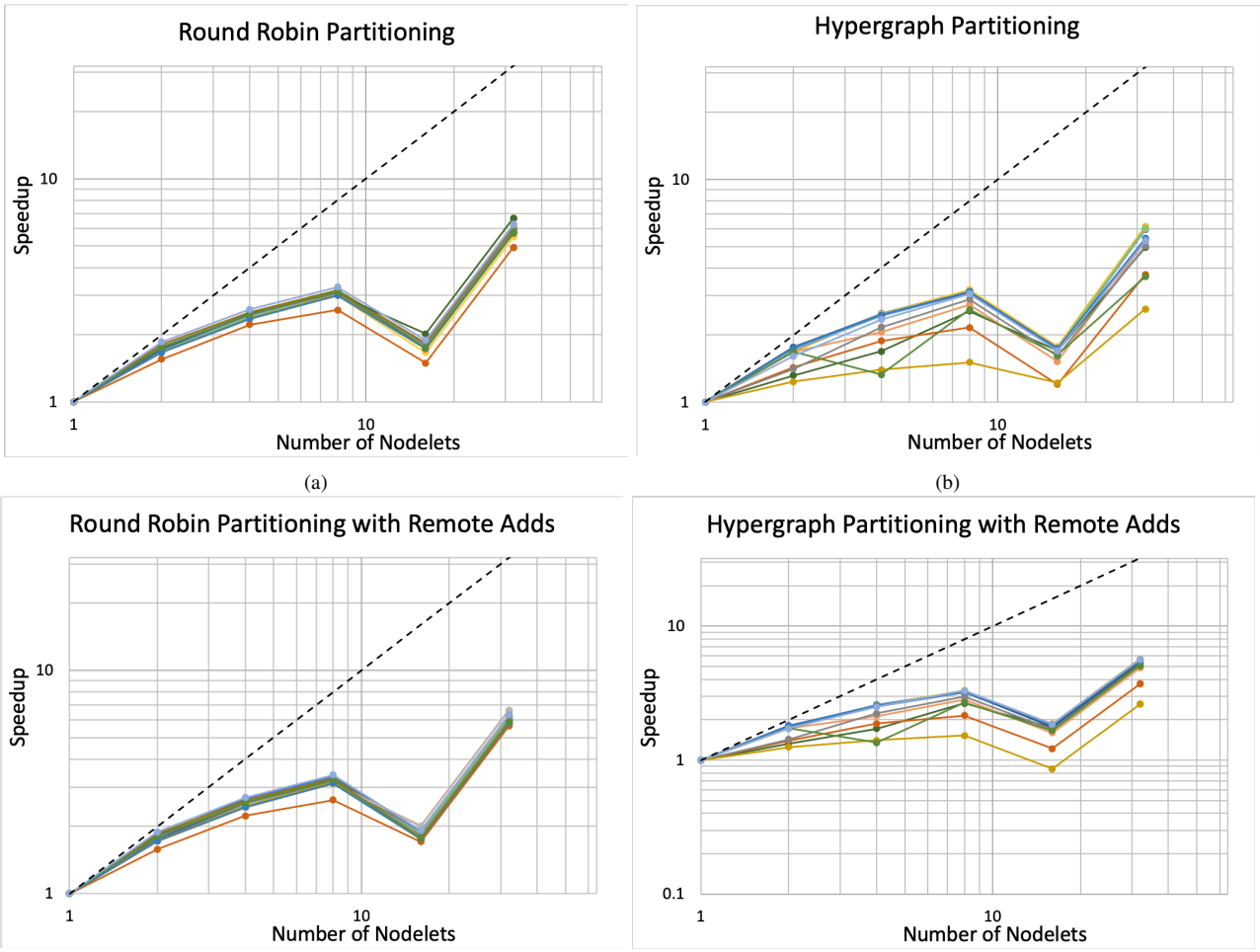



Fig. 4: Observed speedup for all benchmark matrices and nodelet counts evaluated. Each line represents a single sparse matrix.

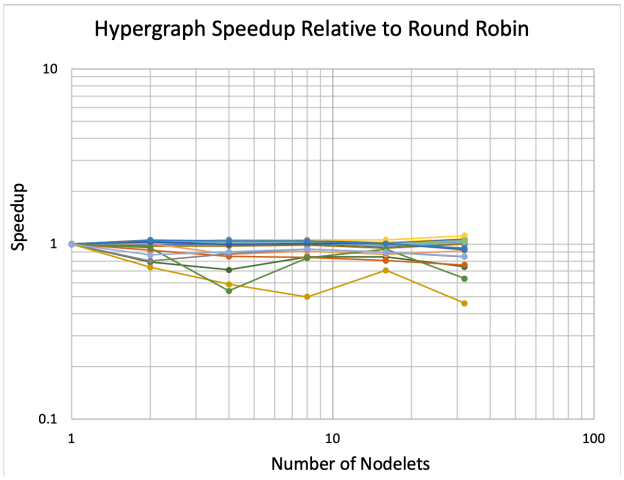



Fig. 5: Speedup Relative to Round Robin non-zero distribution


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

[9] P. M. Kogge and S. K. Kuntz. A Case for Migrating Execution for Irregular Applications. In *Proc. of the 7th Workshop on Irregular Applications: Architectures and Algorithms, with SC17*, IA3 '17, Nov. 2017.

[10] V. Marjanović, J. Gracia, and C. W. Glass. Performance modeling of the hpcg benchmark. In S. A. Jarvis, S. A. Wright, and S. D. Hammond, editors, *High Performance Computing Systems. Performance Modeling, Benchmarking, and Simulation*, pages 172–192, Cham, 2015. Springer Int. Publishing.

[11] C. Mayer, R. Mayer, S. Bhowmik, L. Epple, and K. Rothermel. HYPE: massive hypergraph partitioning with neighborhood expansion. *CoRR*, abs/1810.11319, 2018.

[12] M. Mosby and K. Matouš. Hierarchically parallel coupled finite strain multiscale solver for modeling heterogeneous layers. *Int. Journal for Numerical Methods in Engineering*, 102(3-4):748–765, 2015.

[13] M. Mosby and K. Matouš. Computational homogenization at extreme scales. *Extreme Mechanics Letters*, 6:68 – 74, 2016.

[14] B. A. Page and P. M. Kogge. Scalability of hybrid sparse matrix dense vector (spmv) multiplication. *Int. Conf. on High Performance Computing & Simulation*, Jul 2018.

[15] B. A. Page and P. M. Kogge. Scalability of hybrid spmv on intel xeon phi knights landing. *Int. Conf. on High Performance Computing & Simulation*, Jul 2019.

[16] B. A. Page and P. M. Kogge. Scalability of hybrid spmv with hypergraph partitioning and vertex delegation for communication avoidance. *HPDC in review*, Jul 2020.

[17] T. Rolinger, C. Krieger, and A. Sussman. Optimizing data layouts for irregular applications on a migratory thread architecture. In *2019 IEEE/ACM Workshop on Memory Centric High Performance Computing (MCHPC)*, pages 7–15, Nov 2019.

[18] T. B. Rolinger and C. D. Krieger. Impact of traditional sparse optimizations on a migratory thread architecture. *CoRR*, abs/1812.05955, 2018.