Recursive Algebraic Coloring Engine

Poster Summary

Christie Louis Alappat
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
christie.alappat@fau.de

Georg Hager
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
georg.hager@fau.de

Gerhard Wellein
Friedrich-Alexander-Universität Erlangen-Nürnberg
Erlangen, Germany
gerhard.wellein@fau.de

Olaf Schenk
Institute of Computational Science
Lugano, Switzerland
olaf.schenk@usi.ch

Abstract

Many iterative numerical methods for sparse systems and important building blocks of sparse linear algebra feature strong data dependencies. They may be loop-carried dependencies as they occur in many iterative solvers or preconditioners (e.g. Gauss-Seidel type, Kaczmarz solver) or write conflicts as they show up in the parallelization of building blocks such as symmetric sparse matrix vector or sparse matrix transpose vector. Scalable, hardware-efficient parallelization of such kernels is known to be a challenge. Most of the typical schemes currently available to parallelize such kernels suffer from performance issues on modern hardware or are are highly matrix specific or require changes in entire matrix storage format.

In this poster we show a novel method called Recursive Algebraic Coloring (RAC) that achieves high hardware efficiency on modern multi-core architectures and at the same time use simple data storage formats like compressed row storage (CRS). RAC uses a recursive level based method that aims at finding optimal permutations while preserving good data locality. This method is implemented and consolidated into a user friendly library called Recursive Algebraic Coloring Engine (RACE). A thorough performance analysis shows that RACE out-performs traditional Multicoloring methods and Intel MKL implementations with a factor of 2-2.5x. We are on par with Algebraic Block Multicoloring (ABMC) for small matrices, while for large matrices we gain almost a factor of 1.5-2x.

Keywords Shared memory parallel, Distance-k dependency, Graph algorithm, Sparse matrices, Sparse linear algebra, Performance

1 Motivation

- Dependencies for sparse linear algebra kernels ranging from iterative solvers to basic building blocks.
- FEAST eigen value solver which has wide range of application like electronic structure calculation, needs a robust inner linear solver. To date only Kaczmarz (KACZ) has proved to be effective here for large scale simulation. But this has dependency.
- Another example is SymmGS preconditioning in HPCG.
- Commonly we observe two kinds of dependency distance-1 and distance-2. In this poster we concentrate on distance-2 dependencies like that occur in SymmSpMV and KACZ.

2 Related Work

- Locking methods
- Thread private vector and reduction [5]
- Different storage format [1, 9]
- Matrix reordering we concentrate on this because it explicitly doesn't have disadvantages mentioned for others. Two widely used schemes in this area for resolving such dependencies are Multicoloring (MC) [4, 7] and Algebraic Block Multicoloring (ABMC) [6].
- Application of Multicoloring for distance-2 dependent sparse kernels has been shown in [3].
- We extend ABMC for distance-2 dependent kernels.
- But still with these methods we see that the performance is not upto the mark when compared to performance models like roofline [12].
- Further investigations show that with reordering one destroys data locality, needs more synchronizations, increases false sharing especially for distance-2 kernels and so on.

3 Recursive Algebraic Coloring (RAC) method

3.1 RAC objectives

- The method is derived with hardware efficiency and performance as the central concept.

SC'18, November 2018, Dallas,TX USA
2018.

- The method aims to improve the problem of data locality, reduce synchronizations, generate sufficient parallelism and at the same time enabling the implementations on simple data formats like Compressed Row Storage (CRS).

3.2 RAC method

- Its a recursive level based method.
- Its a general method in the sense its applicable to distance-k dependency.
- Currently RAC is limited to matrices having symmetric structure (undirected graph).
- It has four steps, and works on the graph of the matrix.
  1. Level construction : Here we do a BFS[8] on the graph of the matrix, this is a commonly used pre-processing step for normal sparse operations like SpMV, because it's well known such permutations can improve the data locality [10], which is also our aim. One could also substitute this stage with more complicated algorithm like RCM[2].
  2. Permutation: Once the levels in the graph has been found one has to permute the matrices in the order of levels. In order for coloring to work on top of this we should additionally store the information on levels which we do by using a data structure known as `level_ptr`. This basically is an array containing the nodes corresponding to the first element of each level.
  3. Distance-k coloring: With these levels L(i) one can easily show that $L(i)$ and $L(i+(k+j))$ are distance-k independent for all $j \geq 1$. For example in case of k=2, this would mean if we leave at least a gap of two levels between any two group of levels ($T(a)$ and $T(b)$) they are distance-2 independent. Consequently this would mean one could work on $T(a)$ and $T(b)$ in parallel, but serially within each *level group*. This leads to one possible configuration seen in the figure where we have two colors and note that each *level group* in red color is separated by at least two levels of blue color and vice-versa. But this as such is not an efficient manner to parallelize since as one could easily see from the figure that this could lead to load imbalances. Therefore as a final step we do a load balancing.
  4. Load balancing: In this step the main idea is to resolve the distance-k dependency as required by the algorithm in hand but at the same time equally distribute non-zeros within each thread. In order to do this in most efficient way for a given hardware we plug-in also the information from hardware side like number of cores. Then we try to generate the required parallelism for the hardware while keeping data locality and dependency in mind. As seen in figure for the example shown we see that with this algorithm one assigns more levels to the region where each levels have few nodes, but at bigger levels we assign here the minimum requirement of two levels needed to respect the dependency. The load balancing algorithm works in such a way that it tries to reduce the variance of non-zeros (n_{nz}) in the two colors (red and blue) by acquiring or giving levels from the corresponding *level group*.
- Recursion: It might happen that with the above four steps one does not attain sufficient parallelism to satisfy the hardware underneath, therefore in order to achieve more parallelism one has to apply the concept of recursion. Here we first choose a sub-graph (typically a *level group*) based on a global load balancing algorithm which determines splitting a *level group* for making two or more threads to operate on it will be beneficial. Then all the four steps is applied on this sub-graph to generate more parallelism from this sub-graph. In practice the thread which is assigned to the parent sub-graph has to spawn two or more threads to work on each of the splitted part of this sub-graph.

4 RACE library

We have consolidated all this idea of RAC method into a library called Recursive Algebraic Coloring Engine (RACE), which can basically carry out all the procedures explained above. RACE supports user in both pre-processing and processing phase. In the pre-processing phase one has to input it with matrix details, kernel requirements (like distance-1 or distance-2) and hardware settings. The library then generates a permutation and stores the recursive coloring information in form of a tree called `level_tree`. Internally the library also creates an internal pinned thread pool which will be used later for the processing phase. In the processing phase user just needs to provide a serial kernel (with dependency) and supply its arguments, then the library can execute the function in parallel with the help of thread pool created and respecting all the dependencies.

5 Performance

In order to compare the performance we use the test setup as mentioned in the poster and AD sheet. In this poster we show performance of two kernels SymmSpMV and KACZ sweeps on two Intel architectures: modern SkyLake and Ivy Bridge. We compare our results against ABMC, MC and MKL implementations. From the performance measurements it is clear that RACE has an upper hand for almost all the matrices, and next comes ABMC followed by MKL and MC versions. Especially for large matrices where the memory traffic and data locality becomes crucial here RACE clearly outperforms others even getting over 2.5 × speed-up in performance. The reason for this can be clearly seen also with

data traffic measurements using hardware counters (here we used LIKWID [11]) as shown for an example matrix.

When it comes to iterative solvers like KACZ it is also important to study the convergence behavior of the method since re-ordering the matrix leads to change in convergence. Relative iterations required to achieve the same absolute error as that of exact kernel (serial KACZ) is shown in the spider plot. Here we see RACE is on par with ABMC and in some case has slight advantage in terms of convergence, while MC method follows after that. Combining the performance and iterations we could then finally arrive at the actual benefit one could achieve from RACE which is shown as inverse time to solution.

6 Future Work

References

[1] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. 2009. Parallel Sparse Matrix-vector and Matrix-transpose-vector Multiplication Using Compressed Sparse Blocks. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures (SPAA '09)*. ACM, New York, NY, USA, 233–244. https://doi.org/10.1145/1583991.1584053

[2] Elizabeth Cuthill. 1972. *Several Strategies for Reducing the Bandwidth of Matrices*. Springer US, Boston, MA, 157–166. https://doi.org/10.1007/978-1-4615-8675-3_14

[3] Martin Galgon, Lukas Krämer, Jonas Thies, Achim Basermann, and Bruno Lang. 2015. On the Parallel Iterative Solution of Linear Systems Arising in the FEAST Algorithm for Computing Inner Eigenvalues. *Parallel Comput.* 49, C (Nov. 2015), 153–163. https://doi.org/10.1016/j.parco.2015.06.005

[4] Assefaw H. Gebremedhin, Duc Nguyen, Md. Mostofa Ali Patwary, and Alex Pothen. 2013. ColPack: Software for Graph Coloring and Related Problems in Scientific Computing. *ACM Trans. Math. Softw.* 40, 1, Article 1 (Oct. 2013), 31 pages. https://doi.org/10.1145/2513109.2513110

[5] T. Gkountouvas, V. Karakasis, K. Kourtis, G. Goumas, and N. Koziris. 2013. Improving the Performance of the Symmetric Sparse Matrix-Vector Multiplication in Multicore. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*. 273–283. https://doi.org/10.1109/IPDPS.2013.43

[6] Takeshi Iwashita, Hiroshi Nakashima, and Yasuhito Takahashi. 2012. Algebraic Block Multi-Color Ordering Method for Parallel Multi-Threaded Sparse Triangular Solver in ICCG Method. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium (IPDPS '12)*. IEEE Computer Society, Washington, DC, USA, 474–483. https://doi.org/10.1109/IPDPS.2012.51

[7] Mark T. Jones and Paul E. Plassmann. 1994. Scalable Iterative Solution of Sparse Linear Systems. *Parallel Comput.* 20, 5 (May 1994), 753–773. https://doi.org/10.1016/0167-8191(94)90004-3

[8] C. Y. Lee. 1961. An Algorithm for Path Connections and Its Applications. *IRE Transactions on Electronic Computers* EC-10, 3 (Sept 1961), 346–365. https://doi.org/10.1109/TEC.1961.5219222

[9] Michele Martone. 2014. Efficient Multithreaded Untransposed, Transposed or Symmetric Sparse Matrix-vector Multiplication with the Recursive Sparse Blocks Format. *Parallel Comput.* 40, 7 (July 2014), 251–270. https://doi.org/10.1016/j.parco.2014.03.008

[10] L. Oliker, X. Li, P. Husbands, and R. Biswas. 2002. Effects of Ordering Strategies and Programming Paradigms on Sparse Matrix Computations. *SIAM Rev.* 44, 3 (2002), 373–393. https://doi.org/10.1137/S00361445003820

[11] J. Treibig, G. Hager, and G. Wellein. 2010. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. (2010).

[12] Samuel Williams, Andrew Waterman, and David Patterson. 2009. Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM* 52, 4 (April 2009), 65–76. https://doi.org/10.1145/1498765.1498785