RACE[*]

Abstract.

- In this paper we propose a method to parallelize sparse linear algebra kernels having dependencies. The method is called RACE and is a recursive level-based approach.
- The main aim of the paper is to explain the RACE method and procedures involved in it. Furthermore a brief parameter study and performance modeling of kernels executed with RACE is carried out.
- In this paper we test our method for both distance-1 (GS) and distance-2 (SymmSpMV and KACZ) dependencies, and analyze both exact (SymmSpMV) and iterative kernels (GS, KACZ).
- For distance-1 kernels we compare against the existing approaches of MC [16], ABMC [15] and MKL [14] implementations.
- For distance-2 kernels along with comparing against existing approaches of MC [9] and implementations in MKL [14], we also extend the ABMC method for distance-2 coloring. To our knowledge this is the first paper which uses ABMC method for distance-2 coloring.
- Comparisons with SymmSpMV (an exact kernel) enables us to solely study the performance behavior of RACE compared with others, since here effects like convergence does not play a role. Overall we achieve a speed-up of $2-2.5\times$ compared to MC and MKL implementations, while we are on par with ABMC for small matrices and for large matrices we gain almost a factor of $1.5-2\times$ benefit.
- Comparisons with iterative kernels also puts into light the convergence behavior of the method. Results show that the convergence of RACE is better than MC and is competitive with ABMC method.
- Finally we compare our method against a different sparse matrix data format known as RSB, which is a tailored data format for working on kernels having dependencies. How much details on this comparison should we include in the paper?

Key words. example, LaTeX

AMS subject classifications. 68Q25, 68R10, 68U05

1. Introduction.

The introduction introduces the context and summarizes the manuscript. It is importantly to clearly state the contributions of this piece of work. The next two paragraphs are text filler, generated by the `lipsum` package.

Nam dui ligula, fringilla a, euismod sodales, sollicitudin vel, wisi. Morbi auctor lorem non justo. Nam lacus libero, pretium at, lobortis vitae, ultricies et, tellus. Donec aliquet, tortor sed accumsan bibendum, erat ligula aliquet magna, vitae ornare odio metus a mi. Morbi ac orci et nisl hendrerit mollis. Suspendisse ut massa. Cras nec ante. Pellentesque a nulla. Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Aliquam tincidunt urna. Nulla ullamcorper vestibulum turpis. Pellentesque cursus luctus mauris.

Nulla malesuada porttitor diam. Donec felis erat, congue non, volutpat at, tincidunt tristique, libero. Vivamus viverra fermentum felis. Donec nonummy pellentesque ante. Phasellus adipiscing semper elit. Proin fermentum massa ac quam. Sed diam turpis, molestie vitae, placerat a, molestie nec, leo. Maecenas lacinia. Nam ipsum ligula, eleifend at, accumsan nec, suscipit a, ipsum. Morbi blandit ligula feugiat magna. Nunc eleifend consequat lorem. Sed lacinia nulla vitae enim. Pellentesque tincidunt purus vel magna. Integer non enim. Praesent euismod nunc eu purus. Donec bibendum quam in tellus. Nullam cursus pulvinar lectus. Donec et mi. Nam vulputate metus eu enim. Vestibulum pellentesque felis eu massa.

The paper is organized as follows. Our main results are in **??**, our new algorithm

[*]Submitted to the editors DATE.

Funding: This work was funded by the Fog Research Institute under contract no. FRI-454.

is in **??**, experimental results are in **??**, and the conclusions follow in **??**.

2. Related Work.

One of the earliest work on parallelizing kernels having loop-carried dependencies is the red-black Gauss-Seidel scheme [8]. Later Kamath and Sameh introduced a two-block partitioning scheme for parallelizing Kaczmarz method on tridiagonal structures [17]. A general study on the convergence of these methods were done early in 1980 by Elfving [7].

The advent of processors having more parallelism and the need to consider more unstructured matrices have made graph-based approach an important tool for parallelizing such kernels. Multicoloring is one of the most popular approach used in this field [16], but is sometimes not efficient on modern cache-based processors. There have been researches going on to increase the efficiency of multi-coloring and improving the heuristics, an overview of the methods can be found in [21, 10, 11]. One of the most successful and effective method in this regard is the algebraic block multi-coloring [15] proposed by Iwashita et al. in 2012.

Another line of research focuses on parallelizing dependent kernels while maintaining the same convergence behavior of sequential execution. One of the earliest known works in this category is the hyperplane method [26] on FDM (Finite Difference Method) like matrices. Extensions to this approach can be seen in [23] where a hybrid approach between multi-coloring and hyperplane method is used. However the most general method which falls into this category is level-scheduling [26]. Efficient implementation of this method can be attributed to Park et al. with his work on triangular solvers [24].

Most of the above mentioned method have been tested only for their applicability to parallelize distance-1 dependent kernels and some of them are not capable to deal with dependencies like distance-2. The research on parallelizing distance-1 dependent kernels has been strongly accelerated after the introduction of HPCG benchmark [5]. When it comes to distance-2 kernels popular methods seen in the literature are locking based methods, thread private local vectors [12, 6] for kernels like symmetric sparse matrix vector or with the usage of specially tailored sparse matrix data formats like compressed sparse blocks (CSB) [3] or recursive sparse blocks (RSB) [22].

3. Contribution.

The paper focuses on developing an alternative method to parallelize kernels having loop-carried dependencies. The method introduced here is applicable for solving general distance-k dependencies, similar to multi-coloring methods. Currently we focus only on undirected graph i.e., matrices with symmetric sparsity pattern (but not necessarily symmetric entries). The main motivation of the approach is to achieve good hardware performance on modern hardware architecture, by generating sufficient parallelism while preserving good data locality. The method needs no specialized data format, and works basically on simple sparse matrix format like compressed row storage (CRS).

Most of the above approaches explained above in section 2 suffer from performance penalties in one way or the other, for example multi-coloring degrades the data locality, although this can be improved considerably using algebraic block multi-coloring, still for moderately large matrices or with the increase in k of distance-k dependency the method shows deterioration in performance. Similar drawbacks exists for other methods which will be discussed within this paper.

In this work we provide a detailed performance analysis of the method and comparison between different existing methods chosen from representative classes. The comparisons are done both for exact kernels like symmetric sparse matrix vector (SymmSpMV) having distance-2 dependency and iterative solvers like Gauss-Seidel

(GS) and Kaczmarz (KACZ) schemes having distance-1 and distance-2 dependencies respectively. For iterative schemes we further provide comparison between convergence of different methods. The comparisons are done on different hardware architectures ranging from Intel's Ivy-Bridge series to modern Sky-Lake architecture and the AMD Epyc architecture. The comparisons shows the superiority of our method compared to others and the applicability of our method on wide-variety of heterogeneous systems. As far to our knowledge this is the first paper which demonstrates such high efficiency of distance-2 dependent kernels using simple and common CRS matrix storage format on such broad scale of matrices.

The paper is limited to node level, and we use only thread level parallelization. Multi-node parallelization is left for future work. However it should be noted that for iterative kernels like KACZ and GS node-level performance is far more important because commonly such solvers are applied only locally and different approaches are used for parallelizing between nodes [5, 13].

4. Test bed, matrices and kernels.

4.1. Test bed. The tests are conducted on three different multi-core architectures. Two of them being Intel's Ivy-Bridge and modern Sky-Lake architecture, the choice of these architectures enable study of the method on two extreme generation of Intel's processor currently being used on HPC systems. As a third choice we select AMD's recent Epyc architecture, which is competitive to Intel Sky-Lake architecture. This choice enables us to study the effect of our method on chips based on completely different microarchitecture, enabling us to demonstrate the applicability of our method on wide range of architectures. All the tests are conducted on a single socket of these architectures.

- Intel Ivy-Bridge architecture belongs to class of classic Intel's cache-based architecture, which has three inclusive cache hierarchies. All the cache are scalacble and the LLC (L3) being shared among all the cores on one socket. The processor is capable of delivering one full four wide SIMD add, multiply and load in one cycle.
- Intel Sky-Lake architecture belongs to recent generation of Intel family. Contrary to it's predecessors (like Ivy-Bridge), L3 cache is now changed to a non-inclusive victim cache shared by all the cores on a socket. The architecture comes with support for eight wide SIMD operations (AVX-512). The processor is capable of doing two AVX-512 add, multiply and load operations per cycle.
- AMD Epyc is based on AMD's Zen microarchitecture. The basic building block of the architecture consists of Core Complex (CCX) consisting of three cores (can extend upto four on high end models) each having it's own private L1 and L2 cache. The L3 cache is shared between a core complex and is non-inclusive victim cache. A single socket of Epyc consists of eight such core complexes.

The details of architectures along with the measured bandwidths are given in Table 1. The bandwidths are measured using $likwid - bench$ suite.

The code was compiled with newest Intel compiler version 17 and the following compiler flags were set `-fno-alias -xHost -O3`. Furthermore all the measurements were done with CPU clock speeds fixed at frequencies indicated in Table 1.

4.2. External Tolls and Software. Following external libraries are used in this paper. The application of these libraries will be stated at the point it is needed.

Table 1
Test bed

Model name	Xeon® E5-2660	Xeon® Gold 6148	Epyc 7451
Microarchitecture	Ivy Bridge	Skylake	Zen
Clock	2.2 GHz	2.4 GHz	2.3 GHz
Physical Cores per socket	10	20	24
L1d Cache	10 × 32 kB	20 × 32 kB	24 × 32 kB
L2 Cache	10 × 256 kB	20 × 1 MB	24 × 512 MB
L3 Cache	25 MB	27.5 MB	8 × 8 MB
L3 type	inclusive	non-inclusive	non-inclusive
Main Memory	32 GB	45 GB	4 × 16 GB
Bandwidth per socket - load only	47 GB/s	115 GB/s	130 GB/s
Bandwidth per socket - copy	40 GB/s	104 GB/s	114 GB/s
Architecture specific flag	-	-xCORE-AVX512	-

- LIKWID [28] `likwid-perfctr` is used for measuring hardware performance counters and `likwid-bench` for measuring bandwidth.
- COLPACK [11] is used for pre-processing matrix by multi-coloring.
- Intel SpMP [27] is used to perform Reverse Cuthill McKee (RCM).
- METIS [18] is used for graph partitioning.
- MKL [14] is used for performing some reference sparse matrix computations.

4.3. Benchmark Matrices. All the test matrices are taken from SuiteSparse Matrix Collection (former University of Florida Sparse Matrix Collection) [2] and quantum mechanics field (see ESSEX project [1] for more details). The selection of the matrices from SuiteSparse Matrix Collection is mainly done by combining the test matrices from two papers [22, 24]. This enables easy comparison of results. Matrices from ESSEX project are some of the matrices that are of interest in the FEAST eigen value solver. Only matrices having undirected graphs are considered due to scope of the paper as mentioned in section 3. Matrices along with some of their parameters are given in Table 2. Matrices that have been marked with an * symbol indicate they are corner cases and will be discussed in detail.

4.4. Kernels. To test the performance we choose algorithms that are exact as well as iterative. Also we include kernels from both distance-1 and distance-2 dependency classes. All the kernels shown below are based on CRS matrix storage format.

4.4.1. SpMV. Sparse Matrix Vector (SpMV) is a kernel that do not have any dependencies. It acts as a good reference for other kernels to determine their performance upper bound.

Algorithm 4.1 SpMV Find $b : b = Ax$
```
1: for $row = 1 : nrows$ do
2:     for $idx = rowPtr[row] : rowPtr[row + 1]$ do
3:         $b[row] += A[idx] * x[col[idx]]$
4:     end for
5: end for
```

Table 2
Benchmark matrices

Index	Matrix name	nrows	nnz	nnzr	bandwidth		
1	audikw_1	943695	77651847	82.285	925946		SuiteSparse Matrix Collection
2	bone010	986703	71666325	72.632	13016		
3	channel-500x100x100-b050	4802000	85362744	17.776	600299		
4	crankseg_1	52804	10614210	201.011	50388	*	
5	delaunay_n24	16777216	100663202	6.0	16769102		
6	dielFilterV3real	1102824	89306020	80.979	1036475		
7	Emilia_923	923136	41005206	44.419	17279		
8	F1	343791	26837113	78.062	343754		
9	Fault_639	638802	28614564	44.794	19988		
10	Flan_1565	1564794	117406044	75.03	20702		
11	G3_circuit	1585478	7660826	4.832	947128		
12	Geo_1438	1437960	63156690	43.921	26018		
13	gsm_106857	589446	21758924	36.914	588744		
14	Hook_1498	1498023	60917445	40.665	29036		
15	HPCG-192	7077888	189119224	26.72	37057		
16	inline_1	503712	36816342	73.09	502403	*	
17	nlpkkt120	3542400	96845792	27.339	1814521		
18	nlpkkt200	16240000	448225632	27.6	8240201		
19	offshore	259789	4242673	16.331	237738		
20	parabolic_fem	525825	3674625	6.988	525820	*	
21	pwtk	217918	11634424	53.389	189331		
22	Serena	1391349	64531701	46.381	81578		
23	ship_003	121728	8086034	66.427	3659		
24	thermal2	1228045	8580313	6.987	1226000		
25	Anderson-16.5	2097152	14680064	7.0	1198372		ESSEX
26	Graphene-4096	16777216	218013704	12.995	4098		
27	Graphene-short-4096	16777216	67096576	3.999	4096	*	
28	Spin-26	10400600	145608400	14.0	709995		

The arithmetic intensity of the kernel I_{SpMV} is as follows:

$$I_{\text{SpMV}} = \frac{2}{8 + 4 + 8 * \alpha + \frac{16}{N_{nzr}}} \qquad (4.1)$$

where α represents the data locality factor and N_{nzr} non-zeros per row. α depends on the sparsity pattern of the matrix and varies from matrix to matrix. Ideal value of α for sufficiently large matrix is $\frac{1}{N_{nzr}}$. More details on factor α could be found in [19].

4.4.2. SpMTV. Sparse Matrix Transpose Vector (SpMTV) is a kernel having distance-2 dependency.

Algorithm 4.2 SpMTV Find $b : b = A'x$

```
1: for $row = 1 : nrows$ do
2:   for $idx = rowPtr[row] : rowPtr[row + 1]$ do
3:     $b[col[idx]] += A[idx] * x[row]$
4:   end for
5: end for
```

In comparison to SpMV operation, the kernel requires an extra scatter operation, which causes dependency. The arithmetic intensity of the kernel I_{SpMTV} is given as:

$$I_{\text{SpMTV}} = \frac{2}{8 + 4 + 16 * \alpha + \frac{8}{N_{nzr}}} \tag{4.2}$$

In ideal case data traffic for this kernel should remain close to that of SpMV, if N_{nzr} are sufficiently high, and α factor is small enough.

4.4.3. SymmSpMV. Symmetric Sparse Matrix Vector (SymmSpMV) makes use of the symmetric property of the matrix to perform the matrix vector multiplication.

Algorithm 4.3 SymmSpMV Find $b : b = Ax$, where A is an upper triangular matrix

```
1: for $row = 1 : nrows$ do
2:     $diag\_idx = rowPtr[row]$
3:     $b[row] += A[diag\_idx] * x[row]$
4:     for $idx = rowPtr[row] + 1 : rowPtr[row + 1]$ do
5:         $b[row] += A[idx] * x[col[idx]]$
6:         $b[col[idx]] += A[idx] * x[row]$
7:     end for
8: end for
```

To operate on this kernel we just use the upper triangular part of the sparse matrix. The kernel requires only half the data traffic compared to SpMV but requires the same amount of Flops, leading to almost twice the intensity of SpMV operations.

$$I_{\text{SymmSpMV}} = \frac{4}{8 + 4 + 32 * \alpha + \frac{16}{N_{nzr}^{symm}}} \tag{4.3}$$

Note that N_{nzr}^{symm} is the number of non-zeros per row in upper triangular part of the matrix.

4.4.4. GS and SymmGS. Gauss-Seidel (GS) is a solver having distance-1 dependency. Contrary to the above kernels GS is in-exact meaning it is an iterative method. Algorithm 4.4 shows the Gauss-Seidel algorithm where its assumed that the diagonal entries of the matrix are stored as first entry in their corresponding rows.

Algorithm 4.4 GS Solve for $x : Ax = b$

```
1: for $row = 1 : nrows$ do
2:     $x[row] += b[row]$
3:     for $idx = rowPtr[row] + 1 : rowPtr[row + 1]$ do
4:         $x[row] -= A[idx] * x[col[idx]]$
5:     end for
6:     $diag = A[rowPtr[row]]$
7:     $x[row] /= diag$
8: end for
```

Regarding the in-core execution the kernel has same properties as of SpMV, but requires an additional divide operation per row of the matrix. If the locality (α factor) is not disturbed due to pre-processing the kernel requires same data traffic as of SpMV. The arithmetic intensity of GS is the same as that of SpMV, if we neglect the divide operation that occurs once per every row.

$$I_{\text{GS}} = I_{\text{SPMV}} \tag{4.4}$$

In general for most of the algorithms one is interested in symmetric operator therefore commonly one would encounter symmetric variant of Gauss-Seidel, so called symmetric Gauss-Seidel (SymmGS). The algorithm remains same except that instead of just doing forward sweep shown in Algorithm 4.4 one would follow it with a backward sweep i.e.,`row=nrows:-1:1`. The intensity of SymmGS remains same as of GS, as we do two times more flops and bring in proportional data.

4.4.5. KACZ and SymmKACZ.

Kaczmarz (KACZ) is an iterative solver based on row-projection based methods. The solver has a distance-2 dependency.

Algorithm 4.5 KACZ Solve for $x:\ Ax = b$

```
1: for $row = 1 : nrows$ do
2:     $row\_norm = 0$
3:     $scale = b[row]$
4:     for $idx = rowPtr[row] : rowPtr[row + 1]$ do
5:         $scale- = A[idx] * x[col[idx]]$
6:         $rownorm+ = A[idx] * A[idx]$
7:     end for
8:     $scale = scale/rownorm$
9:     for $idx = rowPtr[row] : rowPtr[row + 1]$ do
10:        $x[col[idx]]+ = scale * A[idx]$
11:    end for
12: end for
```

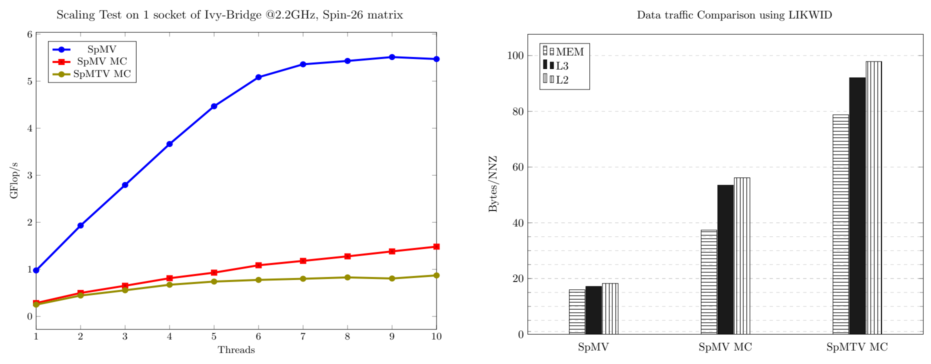
In-core has a mixed behavior of both SpMV and SpMTV similar to SymmSpMV. The solver also requires a divide per row of the matrix. In ideal case the data traffic from memory should remain same as that of SpMTV. But the solver requires thrice the flops compared to SpMTV per non-zero. For brevity of the results we ignore the flops used in *rownorm* computations since, one could also row normalize the sparse matrix before performing the KACZ operation. This leads to an almost two fold higher Arithmetic Intensity compared to SpMTV.

$$I_{\text{KACZ}} = \frac{4}{8 + 4 + 16 * \alpha + \frac{8}{nnzr}} = 2 * I_{\text{SpMTV}} \tag{4.5}$$

Symmetric variant of KACZ is denoted by SymmKACZ, and similar to SymmGS this requires forward sweep followed by a backward sweep.

5. Motivation.

Motivation for developing an alternative method stems from the ESSEX (Equipping Sparse Solvers for Exascale) project [1] where we investigate into solving large eigen-value problems from quantum mechanics field. In this context having a robust iterative solver was inevitable, due to the poor condition number of the matrices that appear in this field. Kaczmarz (KACZ) solver was found to be satisfactory but parallelizing this solver was deemed challenging because of the loop-carried dependencies in the kernel. Previous work on parallelizing the KACZ kernel used multi-coloring (MC) [9] but it was soon found that the kernels do not scale efficiently with this approach.



(a) α effect (b) Data Traffic

FIG. 1. *Effect of Multicoloring*

In order to get a better understanding of the underlying problem it's convenient to choose simple sparse matrix transpose vector (SpMTV) as a benchmark kernel. The particular choice of this kernel is due to the fact that both KACZ and SpMTV have similar kind of dependencies, and it's much easier to compare with our reference kernel namely sparse matrix vector (SpMV) which is embarrassingly parallel. The algorithm for SpMTV and SpMV has been listed in Algorithms 4.1 and 4.2

Figure 1a shows the performance of SpMV kernel on original unpermuted matrix and matrix with MC permutation. Here we see the performance of SpMV on multi-colored matrix is four times worse than that of SpMV on unpermuted matrix. One of the major reason for this drop is due to the increase in α factor seen in the intensity equation (4.2) Since the kernels like SpMV are mainly memory bound increase in α lowers intensity I_{SpMV} leading to a drop of performance as predicted by roofline model [29]. This could easily be demonstrated by measuring the data traffic between different memory hierarchies. We do this using the LIKWID tool [28], and the measurements can be seen in Figure 1b. One can see an increase in data-traffic from all the memory hierarchy compared to SpMV on normal unpermuted matrix. This is basically caused by the bad data locality introduced by multi-coloring permutation.

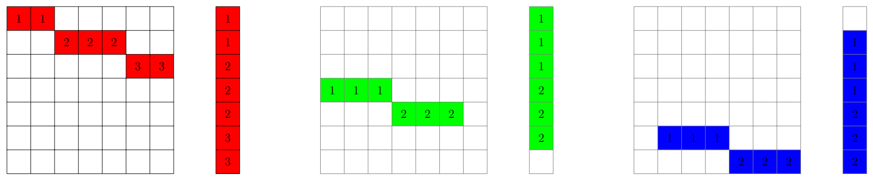


FIG. 2. *Illustration of increase in α by multicoloring, numbers represents thread numbers working on a particular row*

Figure 2 shows an illustration of why data traffic increases for a given matrix. If one assumes last level cache (LLC) can only hold less than six elements and obeys perfect LRU policy, as seen in the Figure 2 for each new color we would need to load the data from main memory. As we will see later this α factor strongly depends on the matrix size and the size of LLC.

As seen in Figure 1b the data traffic further increases for SpMTV due to additional indirect writes (scatter) and this scales up α factor as seen in the denominator of

I_{SpMTV} (see (4.2)), which further decreases performance of SpMTV compared to SpMV on MC matrix.

Other contributors to the drop in performance is global synchronizations and false sharing. These factors strongly depend on the number of colors and in general increase with chromatic number. For the Spin matrix the overhead of synchronization is roughly 10%. For most of the matrices one could also observe a strong positive correlation between false sharing and number of threads for SpMTV kernels, due to the indirect writes in SpMTV.

It was seen that for most of the matrices arising in the project average drop in performance by multi-coloring was almost a factor of two on a single socket of Ivy-Bridge. Although for most of them performance could be improved by algebraic block multi-coloring (ABMC), still the results we obtained were not optimal (especially for large matrices) when compared to performance models which we will see later in section 8. This led to the development of a method which works on a common data format like CRS in which most of the other kernels are written and at the same time preserves data locality, reduce synchronization overheads and false sharing.

6. RACE method.

Keeping in mind the observations from previous section 5, one could observe that it would be best to maintain the non-zeros of matrix close to the diagonal. This has been observed previously in the regard of normal sparse matrix computations like SpMV and has led to the pre-processing of matrix by applying bandwidth reduction algorithms like "Reverse Cuthill McKee " (RCM). Now we aim to develop a method that does not distort this ideal permutations to a large extent but at the same time resolve distance-k dependencies.

Our approach can be seen as a recursive level based method. Each step of the method basically consists of four steps namely:

1. Level construction
2. Permutation
3. Distance-k coloring
4. Load balancing

The method is strongly coupled to the hardware underneath and exploits only the parallelism as required by the hardware. If at the end of all these four steps one does not achieve sufficient parallelism, all the steps are recursively applied to selected sub-graphs of the matrix until sufficient parallelism is attained. This recursive nature of our coloring method led to the naming of the method as "Recursive Algebraic Coloring Engine " in short RACE .

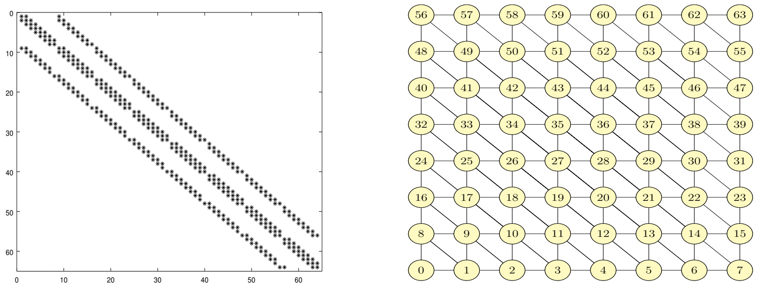
To explain the method in an easier and illustrative way we choose a simple matrix namely the 2D 7pt. stencil. The sparsity pattern and the corresponding graph of the matrix is as shown in Figure 3.

Definitions. The following basic definitions from graph theory are used in the following sections:

- **Graph :** $G = (V, E)$ represents a graph where $V(G)$ belongs to set of vertices and $E(G)$ represents the edges in the graph. Note that here we specifically denote G for irreducible undirected graphs.
- **Neighborhood :** Neighborhood of vertex u represented as $N(u)$ is defined as:

$$N(u) = \{ v : uv \in E \}$$

- **Subgraph :** A subgraph H of graph G in this paper specifically refers to



(a) Sparsity pattern (b) Graph

FIG. 3. *2d-7pt Stencil*

subgraph induced by $V' \subseteq V(G)$ and is defined as

$$H = (V', \{\, uv : uv \in E(G) \text{ and } u, v \in V' \,\})$$

6.1. Level Construction. The first step of the RACE method is level construction. The step concerns with finding different *levels* in the graph, *levels* used here are same to the ones found in "Breadth First Search" (BFS) algorithm [20]. First *level* ($L(0)$) is chosen to consist of a selected root vertex. Rest of the levels ($L(i)$ for $i > 0$) are defined to contain vertices that are in neighborhood of vertices in previous *level* ($L(i-1)$) and not in $L(i-2)$ [4] i.e.,

$$L(i) = \begin{cases} u : u \in N(L(i-1)) \cap \overline{N(L(i-2))} & \text{if } i \neq 0 \\ root & otherwise \end{cases} \tag{6.1}$$

One could easily observe from (6.1) i-th *level* consist of all vertices that have a minimum distance of i from the root node. Algorithm A.1 shows an algorithm to find each nodes minimum distance from root. Total number of levels obtained with this graph traversal will be denoted as n_l. Figure 4a shows *levels* on the 2d-7pt stencil ($n_l = 14$), the main number on each vertex (v) refers to the vertex number and the superscript shows the *level* number, i.e.,

$$v^i \implies v \in L(i) \tag{6.2}$$

Note that this is substantially different to the *levels* in methods like "level-scheduling" [26] where depth (maximum distance) is sought after.

6.2. Permutation. Once the *levels* are known one has to permute the matrix in the order of its *levels*, such that vertices in $L(i)$ appears before that of $L(i-1)$. Till this step the procedure is similar to that of BFS pre-processing for bandwidth reduction. One could also replace BFS with better bandwidth reduction algorithms like "(Reverse) Cuthill McKee ". Figure 4 shows the graph ($G' = P(G)$) of 2d-7pt stencil matrix after this permutation (P) is applied. Observe the difference in node numbering between original lexicographic ordering in Figure 4a and Figure 4b. Now the most important step for resolving dependencies (coloring) is to store the information about *levels*. In order to do this we use a data structure called `level_ptr`. It stores the starting vector of each *levels*, which implies that *levels* on G' can be

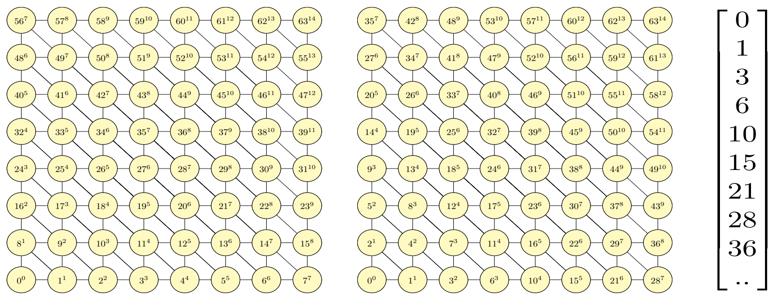

(a) Level construction (b) Permuted graph (G') (c)

FIG. 4. *(a) Levels in 2d-7pt stencil, (b) shows graph G' after permutation and (c) is the associated* `level_ptr` *to G'.*

identified as:

$$L(i) = \{\, u : u \in [\texttt{level\_ptr}[i] : (\texttt{level\_ptr}[i+1]-1)] \text{ and } u \in V(G') \,\}$$

`level_ptr` for 2d-7pt stencil example is shown in Figure 4c, and one could easily read from `level_ptr` that vertices from $\texttt{level\_ptr}(4) = 7$ to $\texttt{level\_ptr}(5) - 1 = 10$ belongs to $L(4)$.

6.3. Distance-k coloring. Two vertices are called distance-k neighbours if the shortest path connecting them consists of at most k edges [10]. This implies u is a distance-k neighbour of v (denoted as $u \xrightarrow{k} v$) if

$$u \xrightarrow{k} v \iff v \in \{\, u \cup N(u) \cup N^2(u) \cup ...N^k(u) \,\} \tag{6.3}$$

Since we consider only undirected graph $u \xrightarrow{k} v$ also implies $v \xrightarrow{k} u$. After having the permuted graph G' one can show that $L(i)$ and $L(i+k+j)$ where $j \geq 1$ are distance-k independent as shown in the following Corollary 6.1:

COROLLARY 6.1. *$L(i)$ and $L(i \pm (k+j))$ are distance-k independent $\forall j \geq 1$.*

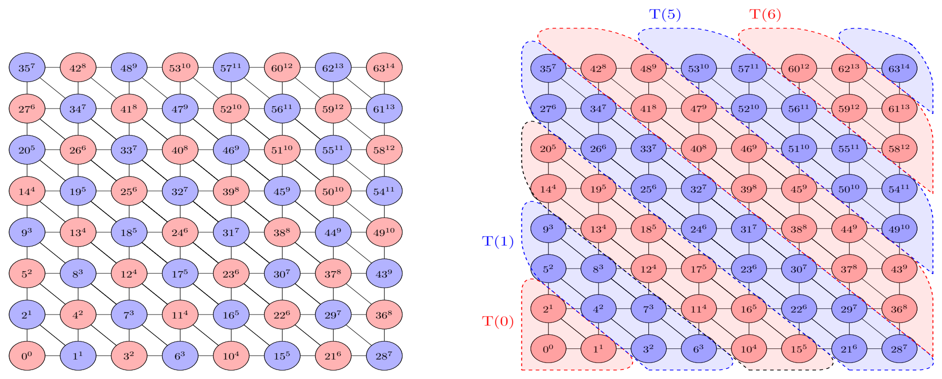
Proof. We prove by contradiction. Let there exist $u, v \in V(G')$ such that $u \in L(i)$ and $v \in L(i \pm (k+j) \forall j \geq 1$. Assume u, v are distance-k neighbours ($u \xrightarrow{k} v$). From (6.1), (6.3) and the fact G' is undirected we get

$$\begin{aligned} u \xrightarrow{k} v &\iff v \in \{\, L(i) \cup L(i \pm 1) \cup ... \cup L(i \pm k) \,\} \\ &\implies v \notin L(i \pm (k+j) \ \forall j \geq 1 \end{aligned}$$

which is a contradiction to the fact $v \in L(i \pm (k+j)) \forall j \geq 1$, this implies u and v are distance-k independent. ∎

Corollary 6.1 implies that if we leave a gap of *at least* one *level* between any two *levels* ($L(i), L(i+2)$ for example) all the vertices between them are distance-1 independent. Similarly if there is a gap of *at least* two *levels* between any two *levels* ($L(i), L(i+3)$ for example) we get distance-2 independent levels.

Due to this weak definition in Corollary 6.1 there exists many possibility to make *levels* independent of each other and Figure 5 shows one such possibility each for distance-1 and distance-2 independent *levels*. One could group some of the nearby



(a) distance-1 independent *level groups* (b) distance-2 independent *level groups*

FIG. 5. *distance-*1 *and distance-*2 *independent level groups.*

levels together to form a *level group*, and make this distance-1 or distance-2 independent of other *level groups*. The i-th *level group* would be denoted by $T(i)$. Difference between *level* and *level group* can be seen in Figure 5b, for Figure 5a *level group* and *level* coincides.

In principle one could compute on all independent *level groups* in parallel, but sequentially within a *level group*, i.e. for example in Figure 5b $T(0)$, $T(2)$, $T(4)$, $T(6)$ can be operated by four different threads in parallel and in the next sweep rest *level groups*. For the configurations seen in Figure 5 this would mean we have $\frac{n_l}{2}$ and $\frac{n_l}{4}$ parallelism for distance-1 and distance-2 kernels respectively.

But the problem with the configurations like the one seen in Figure 5 is that there is load imbalances between threads as the number of rows (n_r) per *level group* is not distributed evenly. As seen here in the case of 2d-7pt stencil the threads working on extreme ends of graph (e.g., $T(1), T(7)$) have small amount of work compared to the threads working on middle (e.g., $T(3), T(4)$).

6.4. Load balancing. Depending on the matrix each *level group* would contain different number of rows, which leads to load imbalances as seen above in subsection 6.3. In order to avoid this problem we employ a load balancing scheme. At this step we plug in detail from hardware side like total parallelism. The idea is to exploit only the parallelism as required by the hardware while at the same time maintain distance-k constraint seen in Corollary 6.1. To balance the load more nearby *levels* would be added to a *level group* which has less number of n_r and at *level group* where we have considerably big *levels* only sufficient amount of *levels* to maintain distance-k constraint would be assigned. Assigning nearby levels instead of a random level further helps in preserving data locality.

An algorithm for load balancing can be found in Algorithm A.2. The aim of the algorithm is to reduce combined variance of number of rows ($n_r(T(i))$) in each *level group* $T(i)$. It does this by calculating mean and variance of T_size in each parallel sweeps, where $T\_size(i) = n_r(T(i))$. For example in Figure 5b we need to calculate mean of T_size of all *level groups* in red sweep and blue sweep separately. The combined variance is then found by summing up the variances in each parallel sweep. In order to reduce this combined variance we select the *level group* that has biggest absolute deviation from mean and try to add/remove levels to/from this *level group* from/to a *level group* that has biggest/least signed deviation. While removing *levels* from a *level group* one has to take care that the distance-k coloring is not violated, for example in case of distance-2 and two sweep scheme as seen in Figure 5b we need to ensure at least two levels remain in a *level group*. To aid this shifting of *levels* to/from *level group* we use the pointers to *level group* denoted by T_ptr. Doing this process in an iterative way finally we end up in a state with lowest combined variance at which no further moves are possible either due to violation of distance-k dependency or due to increase in combined variance. Figure 6 shows step by step procedure involved in load balancing and Figure 7 shows *level groups* after load balancing applied on 2d-7pt stencil example of size 16×16.

One could also do this entire load balancing based on number of non-zeros (n_{nz}) rather than n_r, in this case $T\_size(i) = n_{nz}(T(i))$.

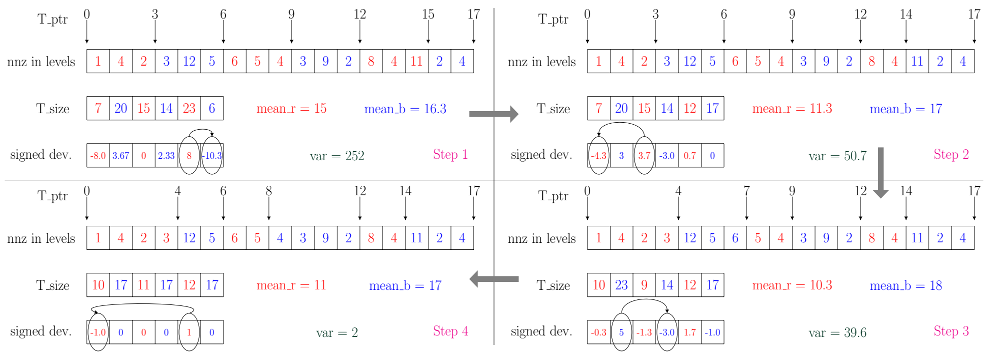


FIG. 6. *Steps in load balancing (clockwise starting from top-left)*

6.5. Recursion. As seen above in subsection 6.3 maximum amount of parallelism by the above approach depends on n_l, also for most of the graphs as we approach the limit of parallelism there is not much room for load balancing, leading to imbalances. Depending on matrix and hardware underneath this might lead to inefficient utilization of resources. In order to avoid this problem we use the concept of recursion and exploit further parallelism if required by the hardware. Idea here is to intelligently select sub-graph(s) of the entire matrix and apply all the four steps recursively on this sub-graph. In the following we will show this concept in the context of distance-1 and later we will extent it to distance-k dependencies. Further we will discuss on the method employed to select proper sub-graph and to have a globally balanced load.

6.5.1. Distance-1. *Level groups* which we constructed till now belongs to stage 1 of recursion and to make the explanations easier the stage number of recursion would be denoted as subscript i.e., $\mathrm{L}_s(i)$ denotes *level* i of stage s. Contrary to methods like multi-coloring we didn't require each nodes in a color to be distance-1 independent of each other rather we had a weak constraint as prescribed by Corollary 6.1. Due to this there can exist more parallelism within a *level group*. For example in Figure 8 we see that within third *level group* ($\mathrm{T}_1(3)$=$\mathrm{L}_1(3)$) vertices 4 $\overset{1}{\nrightarrow}$ 5 (4 distance-1 independent to 5), 4 $\overset{1}{\nrightarrow}$ 6, 4 $\overset{1}{\nrightarrow}$ 7 and 5 $\overset{1}{\nrightarrow}$ 7, implying each of these pairs can be computed in parallel without any distance-1 conflicts. This parallelism couldn't be exploited in stage 1 since vertices in $L_1(k)$ (here k=3) were connected to preceding *level* $L_1(k-1)$ although some of them were not distance-1 dependent within $L_1(k)$.

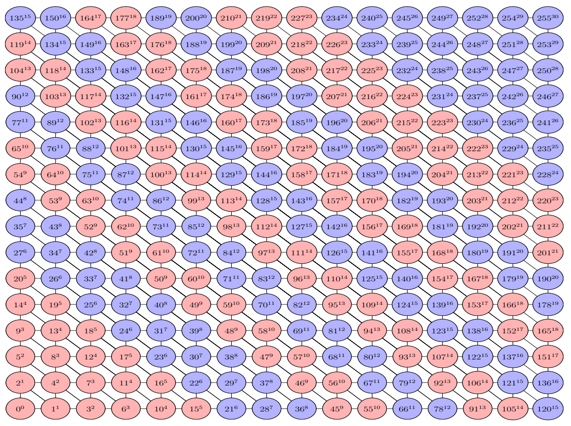


FIG. 7. *After load balancing for five threads and distance-2 dependency on 2d-7pt stencil example, domain size* 16×16. *Note that level groups at extreme end have more levels due to less* n_r *in each level, while level groups in middle having bigger levels maintain two levels to preserve distance-2 constraint.*

In order to exploit this parallelism we use the concept of recursion.

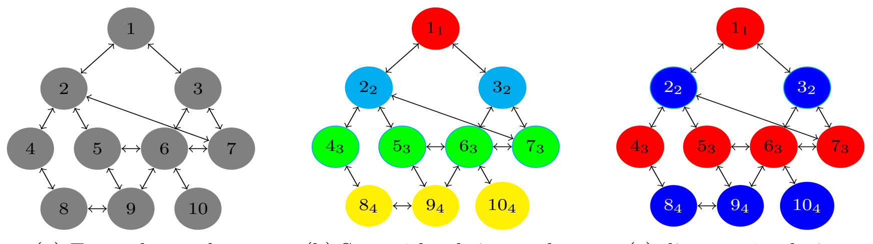


FIG. 8. *Shows potential for more parallelism.* $T_1(2), T_1(3)$ *and* $T_1(4)$ *has more parallelism.*

Recursion begins by selection of a sub-graph of the matrix. A typical choice is a sub-graph induced by vertices in a *level group* of previous stage, more on the selection of sub-graph will be seen later in subsection 6.5.4. For example let's choose sub-graph induced by $T_1(3)$ for recursion. The chosen sub-graph can be isolated from rest of the graph since distance-1 coloring step in stage 1 has already made *level groups* in a sweep independent of each other. Now we just need to repeat all the four step explained previously (subsection 6.1 - subsection 6.4) to exploit parallelism within this sub-graph.

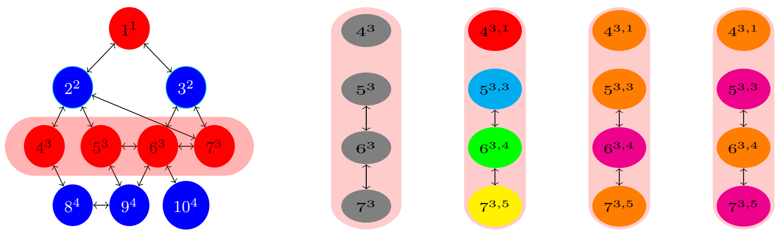


FIG. 9. *Shows recursion being applied to* $T_1(3)$. *Figure 9b shows the selected sub-graph, Figure 9c shows level construction step on the sub-graph, Figures 9d and 9e shows two possibility of distance-1 coloring of the sub-graph*

Figure 9 shows an illustration of applying stage 2 of recursion on $T_1(3)$ to find more parallelism. To incorporate the information of levels after recursion we extent the definition in (6.2) to the following:

$$v^{i,j,k\dots} \implies v \in \{\, L_1(i) \cap L_2(j) \cap L_3(k) \cap \dots \} \tag{6.4}$$

Note that the sub-graphs might have multiple islands (group of vertices in a graph that are not connected to rest of the graph). For example vertex 4 in Figure 9b is an island in the considered sub-graph, similarly vertices 5,6,7 combine to form an island. Since an island is totally disconnected from the rest of the graph it can be executed in parallel to rest of the graph. To take advantage of this the starting node in next island is assigned with an increment of two levels, as seen in Figure 9c. Due to this there exists multiple valid distance-1 configuration (here Figures 9d and 9e) and the selection of the optimal one will be done in the final load balancing step of a particular stage as described in subsection 6.4.

With this recursive process we were able to find independent *level groups* (T_{s+1}) within *level group* of previous stage (T_s) and therefore the thread which works on T_s has to spawn threads to parallelize within T_{s+1}.

6.5.2. Distance-k. For distance-k the same procedure as distance-1 applies, except with a slight difference in selecting the sub-graph. In distance-1 we considered sub-graphs induced by *level groups*, but for distance-k coloring this is not sufficient. As seen in Figure 10 for distance-2 coloring the selection of $T_1(2)$ as sub-graph did not guarantee distance-2 independency between *level group* T_2 within the sub-graph. This is due to the fact for $k > 1$ dependency vertices a, b within a sub-graph might be connected to a common vertex (c) outside the sub-graph leading to a distance-k dependency between a and b. In Figure 10 we see $4 \xrightarrow{1} 2$ & $7 \xrightarrow{1} 2 \implies 4 \xrightarrow{2} 7$, but since vertex 2 was not in the sub-graph considered we missed this dependency.

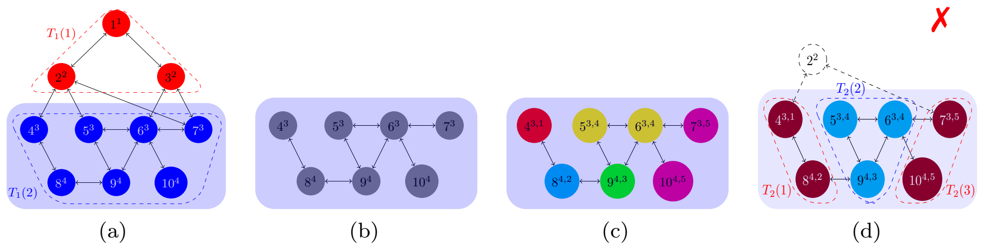


Fig. 10. *Figures 10a and 10b shows level group induced sub-graph selected for recursion in case of distance-2. But applying the four steps to this selected sub-graph does not guarantee a distance-2 independency between level group of same sweep (color) as seen in Figure 10d*

In order to resolve such dependency we have to consider an extra $(k-1)^{th}$ interface level(s) of the selected sub-graph for the level construction step. k^{th} interface level of subgraph $L_s(j)$, denoted as $I^k(L_s(j))$, is defined as follows:

$$I^k(L_s(j)) = \{\, u : u \xrightarrow{k} v \ \forall v \in L_s(j) \text{ and } u \notin L_s(j) \,\}$$

For distance-2 this would mean we have to include 1 interface level, the new selection is illustrated in Figure 11. With the new sub-graph selection for distance-2 coloring as seen in Figure 11a, the result after third step remains correct with respect to distance-

2 coloring. In the example vertices 4 and 7 which had same color previously now gets a different color in (see Figure 11d).

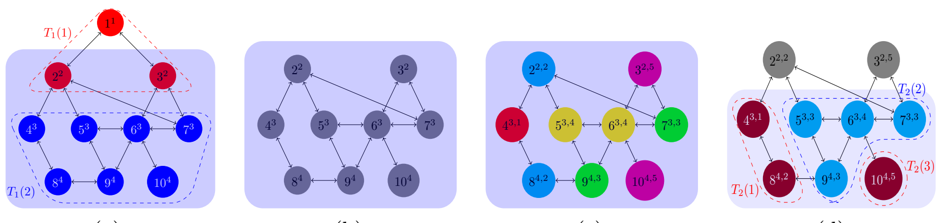
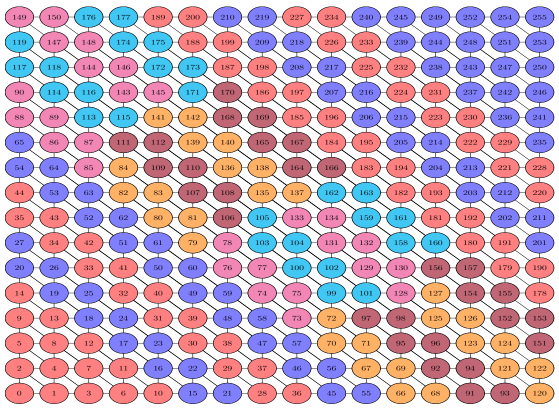


FIG. 11. *Correct procedure of selecting sub-graph for distance-2 coloring. The level group* $T_1(2)$ *and it's* 1*st* *interface level is chosen as the sub-graph as shown in Figures 11a and 11b. Level construction step is then applied to this sub-graph as seen in Figure 11c. Rest of the steps like permutation and distance-k coloring is applied only to the sub-graph chosen for recursion (here* $T_1(2)$*). Finally as seen in Figure 11d we get three level groups at the end of recursion on* $T_1(2)$*.*

Note that the interface levels have to be considered only in the first step namely level construction in the rest of the steps we just need to consider target sub-graphs induced by *level groups* i.e., in Figure 11 the sub-graph induced by $T_1(2)$.



```
for parallel all red
   for parallel all orange
   for parallel all pink
for parallel all blue
   for parallel all brown
   for parallel all cyan
```

FIG. 12. *2d-7pt stencil example for eight threads. Here recursion is applied on level groups* $T_1(5-8)$*, to get more threads. The execution order of different level group is specified in the short code snippet on right. Note nested parallelism being used.*

Figure 12 (left) shows distance-2 coloring of 2d-7pt stencil example for eight threads. Here we see recursion is applied to *level groups* $T_1(5), T_1(6), T_1(7)$ and $T_1(8)$. In this case each of the *level groups* where recursion is applied spawns parallelism for two threads. The choice of *level groups* to refine and number of threads needed from each recursion are determined using a global load balancing technique as will be explained in subsection 6.5.4.

Figure 12 (right) shows the execution order of different *level groups*. Note the usage of nested parallelism i.e., for example thread responsible for $T_1(5)$ spawns two child threads to execute $T_2(1) \subset T_1(5)$ and $T_2(3) \subset T_1(5)$ in parallel, as well as $T_2(2) \subset T_1(5)$ and $T_2(4) \subset T_1(5)$ in parallel. At the end of each `for parallel all color` there is synchronization between threads assigned to *level group* of corresponding color. Since each of the leaf need to synchronize only with it's siblings (leaves of same parent) we use simple point to point synchronization scheme.

6.5.3. Internal representation of recursively generated *level groups*. The recursive nature of our procedure allows to exploit more parallelism. However this introduces more complexity and one has to additionally respect the dependencies between stages and still observe the dependencies within one stage. The best idea is to have a data structure similar to the recursion, therefore we extent the `level_ptr` data structure to a hierarchical tree data structure to store these informations. This data structure is called a `level_tree`. The root of `level_tree` contains information of entire domain, first child leaves of this root i.e., leaves with depth 1 stores information about *level groups* in stage 1 ($T_1(..)$), leaves with depth 2 stores information about *level groups* in stage 2 (T_2) and so on.

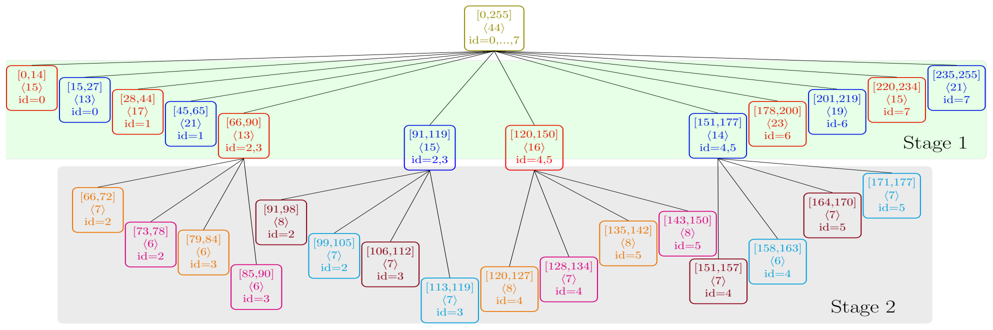


FIG. 13. `level_tree` *corresponding to 2d-7pt stencil example for domain size* 16×16*, and 8 threads. The range (square brackets) specified in each leaves represent the vertices belonging to each level group, the number in bracket (parenthesis) represents the thread assigned to the level group in fill type pinning and the* n_r^{eff} *(see* section 7*) is represented within angle brackets.*

Figure 13 shows a `level_tree` corresponding for 2d-7pt stencil example with 8 threads as seen in Figure 12. The leaves in the tree that correspond to different *level groups* and store various informations like the range of vertices or nodes that belong to this *level group*, the effective number of rows (n_r^{eff}) that describes the quality which will be seen in section 7, and other informations like threads assigned to specific *level groups*. Threads are assigned to each *level group* depending on the pinning strategy used. For example in *fill* type pinning strategy one would pin thread 0 to $T_1(0)$ and $T_1(1)$, thread 1 to $T_1(2)$ and $T_1(3)$, thread 2 to $T_2(0) \subset T_1(4)$, $T_2(1) \subset T_1(4)$, $T_2(0) \subset T_1(5)$ and $T_2(1) \subset T_1(5)$, and so on as seen in Figure 13.

Note that the `level_tree` data structure is an exact replica of the nested parallelism being used as seen in Figure 12 (right). Therefore threads are spawned based on this `level_tree` allowing for easy implementation of point to point synchronizations.

6.5.4. Sub-graph selection and global load balancing. Parallelism required for hardware underneath can be obtained either by expanding the `level_tree` horizontally i.e., increasing *level groups* within a stage or by expanding `level_tree` vertically with the help of recursion. But as we have seen before in subsection 6.3 the horizontal parallelism is limited and after a certain extent this would lead to load balancing. Similarly excessive usage of recursion is also not a good idea since data locality worsens due to local permutations within sub-graph. Therefore it is vital to find a proper balance and choose proper configuration. Furthermore just doing load balancing within a single stage is not the best, for example if we had equally balanced within stage 1 in Figure 13, we would receive no benefit from recursion. Therefore a global load balancing becomes inevitable.

In order to select proper sub-graph and do global load balancing we employ a simple algorithm to find proper weights for each *level group* ($T_s(i)$) in a particular stage, then depending on this weights, denoted as $w(T_s(i))$, we do load balancing with weights in the particular stage (as seen in Algorithm A.2, except weightage is given to *level groups*). Finally if $w(T_s(i)) > 1$ we use recursion to achieve $w(T_s(i))$ parallel work in the next stage of $T_s(i)$. The basic structure of the algorithm employed to find weights is as follows:

1. Find weights, $w(L_s(i))$ for each level in the current stage (s) by

$$w(L_s(i)) = (\texttt{level\_ptr}_s[i+1] - \texttt{level\_ptr}_s[i]) * \frac{n_t}{n_r^{total}}$$

$$n_t : \text{total parallelism required by hardware}$$

$$n_r^{total} : \text{number of vertices in graph}$$

2. Starting from $w(L_s(0))$ sum up weights till they form a number (a) close to whole number (b). The closeness can be controlled by an efficiency parameter for stage s, ϵ_s is defined as:

$$\epsilon_s = 1 - abs(a - b); \tag{6.5}$$

   All the *levels* that are involved in the sum belongs to *level groups* operated by first thread in the current stage. The obtained number b is chosen as weight for these *level groups* i.e., $w(T_s(0)) = w(T_s(1)) = b$. A local search is then done by increasing *levels* in this *level groups* to see if there is a better choice (a close to b) with weight b, finally *level groups* are formed with the best choice. The weight for next *level groups* are found by resetting the sum counter to zero and repeating the procedure with *levels* just after the current *level groups*.

7. Parameter study.

In this section we study the impact of parameter ϵ_s and hardware parallelism on the quality of RACE method. In order to do this we first quantify the quality of the method and finally we use this quantity to do a parameter study. The study gives insights into tuning of parameter ϵ_s based on the given matrix and required parallelism.

7.1. Quantifying quality of RACE.

Quantifying quality of the method in a well-defined way is a primary and most vital step for parameter study. We do this using the concept of *effective parallelism*. From section 6 we saw that even though one tries to achieve parallelism exactly as that required by the hardware, in practice one might not be able to utilize this parallelism to 100 % due to load imbalances. Therefore we use a simple calculation based on the `level_tree` to determine efficiency. This takes into account load imbalances incurred from different stages of recursion. So we first calculate *effective row* for each of the finest leaves (worker leaves) in `level_tree`. The *level groups* (leaves) in `level_tree` that are not further refined form worker leaves as they are responsible for executing the rows (nodes) in their range, the work done by these leaves is therefore directly proportional to the number of rows. Hence the *effective row* of these worker leaves is same as number of rows (n_r), for example in case of $T_1(0)$ *effective row* ($n_r^{eff}(T_1(0)) = n_r(T_1(0))$) is 14 and $n_r^{eff}(T_2(0) \subset T_1(4))$ is 6. After calculating the *effective row* for worker leaves the information is propagated to other leaves in lower stages (up in the `level_tree`) as follows:

$$n_r^{eff}(T_s(i)) = max(n_r^{eff}(T_{s+1}(j) \subset T_s(i))) + max(n_r^{eff}(T_{s+1}(k) \subset T_s(i)))$$
$$\text{for } j \text{ is even and } k \text{ is odd}$$

Such a definition for *effective row* is based on the idea that a parent has to wait until the child leaf with most number of rows in each sweep (color) has finished it's work due to synchronization needed with it's siblings. This has to be handled separately for each of the two parallel sweep (colors) as there is this synchronization happening after each of the sweeps (colors).

Once the information is propagated up the tree and as it reaches the root we have a single *effective row* ($n_r^{eff}(T_0)$) for the entire tree, which has taken care of load balancing happening between all *level groups* in all stages. The ratio of total number of rows (n_r^{total}) in the entire matrix to that of $n_r^{eff}(T_0)$ gives *effective parallelism*, denoted as n_t^{eff}. Efficiency (η) of the method is then defined as ratio of n_t^{eff} to that of required hardware parallelism (n_t).

$$n_t^{eff} = \frac{n_r^{total}}{n_r^{eff}(T_0)} \tag{7.1}$$

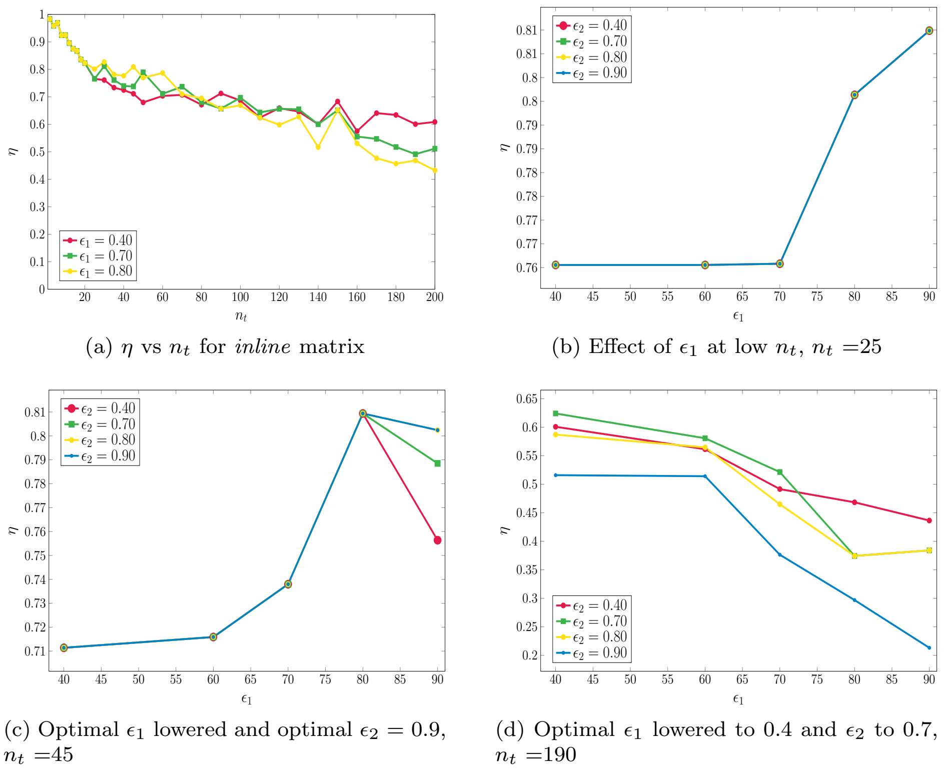
$$\eta = \frac{n_t^{eff}}{n_t} \tag{7.2}$$

For example in our 2d-7pt stencil example, Figure 13 shows n_r^{eff} for each leaves in angular brackets and here $n_t^{eff} = 5.8$ and $\eta = 0.725$. The value of $\eta = 1$ implies there is perfect load balancing which is almost impossible. In general $0 < \eta \leq 1$. This parameter η will be used as a measure of quality in parameter study.

7.2. Case study. A given matrix has a fixed amount of parallelism and as the amount of required parallelism (n_t) increases load balancing degrades due to more threads per stage and imbalances between stages. The rate of degradation can however be controlled to certain extent by the tolerance ϵ_s (see (6.5)) specified while choosing a *level group*. Typical value of ϵ_s is in range of [0.4,0.9]. Having a small ϵ_s (for example 0.4) implies we utilize the current stage 's' to maximum and do not impose high load balancing constraint, a high value on the other hand requires more balanced load from current stage 's'.

Test matrices (see section 4) considered have a varying degree of parallelism, and in order to see the effect of η and ϵ_s we choose the *inline* matrix. The choice is due to the fact that this matrix has relatively small amount of parallelism and this allows us to demonstrate various effect, ranging from good to bad case scenario with small number of parallelism ($n_t < 200$). This limited parallelism can be observed from Figure 14a where efficiency keeps on decreasing with n_t for *inline* matrix. Similar behavior can be observed for *crankseg_1*, *F1* and *ship* matrices, of which *crankseg_1* being the worst. For majority of other test matrices one could observe that efficiency η initially drops but then remains almost constant in the range $\eta = [0.50, 0.80]$ (depending on matrix) for the entire scanned area of $1 \leq n_t \leq 200$.

At small number of threads (n_t) all matrices have high efficiency (like $\eta > 0.8$). As there is a lot of parallelism in this stage compared to requirement, η is insensitive of ϵ_s. The value of n_t upto which such a behavior can be observed varies from matrix to matrix, for example *inline* shows this upto $n_t \approx 20$, while for matrix like *Graphene* this is grater than 200. Further increasing n_t one could observe η starts to vary with ϵ_1. For example in case of $n_t = 25$ one could see in Figure 14b maximum η is achieved with high value of ϵ_1 (0.9) due to good load balancing. But as n_t further increase the optimal ϵ_1 starts shifting towards left (see Figure 14c), since one requires more parallelism from the current stage (s=1) and higher ϵ_1 would be decremental since


(a) η vs n_t for *inline* matrix

(b) Effect of ϵ_1 at low n_t, n_t =25

(c) Optimal ϵ_1 lowered and optimal $\epsilon_2 = 0.9$, n_t =45

(d) Optimal ϵ_1 lowered to 0.4 and ϵ_2 to 0.7, n_t =190

FIG. 14. *Parameter study on* inline *matrix. In* Figures 14b *to* 14d *each lines in the plot are iso-ϵ_2 and impact of η with respect to ϵ_1 is shown.*

it would require the `level_tree` to go more deep and hence load imbalances in next stages will get multiplied. ϵ_2 which till now didn't effect much starts to influence slowly as n_t increments again, for example in case of *inline* till *nthreaads* $= 90$ $\epsilon_2 = 0.9$ was optimal, but then the optimal ϵ_2 reduces and reaches 0.7 at $n_t = 190$ as seen in Figure 14d. η would start to get affected by ϵ_s of next stages in similar manner with increase of n_t.

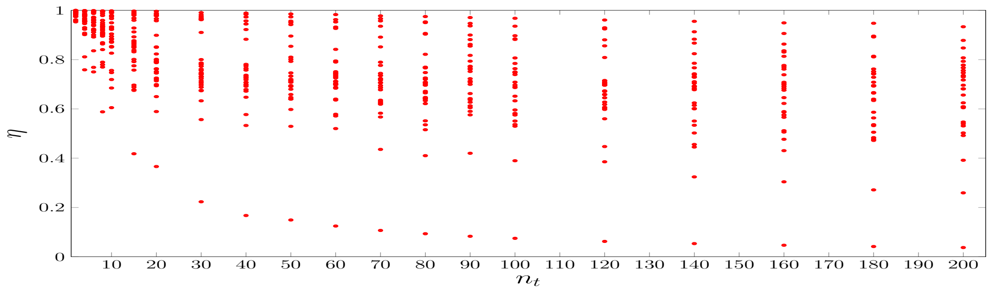


FIG. 15. *Scatter plot of η vs n_t of all test matrices, with ϵ_s = 0.4*

Behavior of other matrices in the test bed follow similar pattern, but n_t at which different phases occur varies from matrix to matrix. Figure 15 gives a broad overview of the efficiency (η) behavior of entire test matrices using scatter plot. Each point at a specific n_t represents efficiency (η) of a matrix. Majority of test matrices having an initial drop in η and then remaining constant is reflected in the statistical plot. The lowest points in the plot correspond to *crankseg_1* matrix, here we achieve only a mere parallelism of eight at maximum ($n_t^{eff} = 8$), while the upper points correspond to matrix having highest parallelism namely *Graphene* matrix.

In practice for a given matrix it's difficult to precisely determine the optimal rate of decrease in ϵ_s without parameter search, and therefore selecting proper ϵ_s for given n_t can be challenging. One idea is to see total levels (n_l) and distribution of non-zeros (n_{nz}) in different levels of current stage 's' and heuristically determine ϵ_s based on the pressure of parallelism from stage 's'. This is not currently done and is part of our future work. Currently for experiments we set $\epsilon_s = 0.8$ for all matrices.

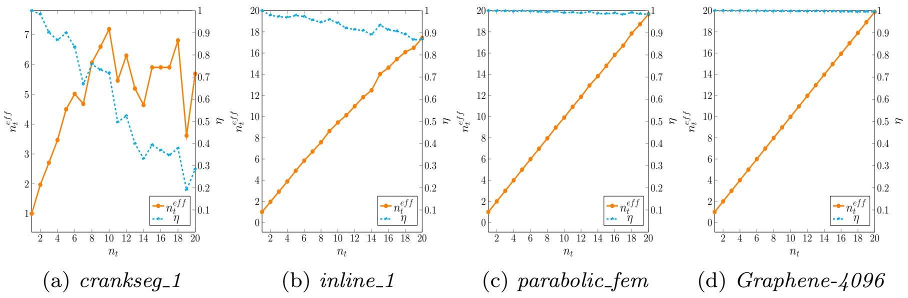


FIG. 16. *n_t^{eff} and η vs n_t for corner case matrices, with the same settings used in experiment runs. n_t^{eff} is defined as $\eta * n_t$.*

In Figure 16 we have plotted n_t^{eff} and η vs n_t for corner case matrices with the settings used in experiment runs. Here we set $\epsilon_{1,2} = 0.8$ and use RCM (Reverse Cuthill McKee) in the *level construction* stage (subsection 6.1). Big fluctuation in *crankseg_1* is due to the fact that we set high load balancing requirement (high ϵ_s factor) and as seen in the example of *inline_1* matrix this is not optimal when we reach the limit of parallelism. The theoretical estimates obtained in Figure 16 will be directly used to compare with experiment runs in the next section (section 8).

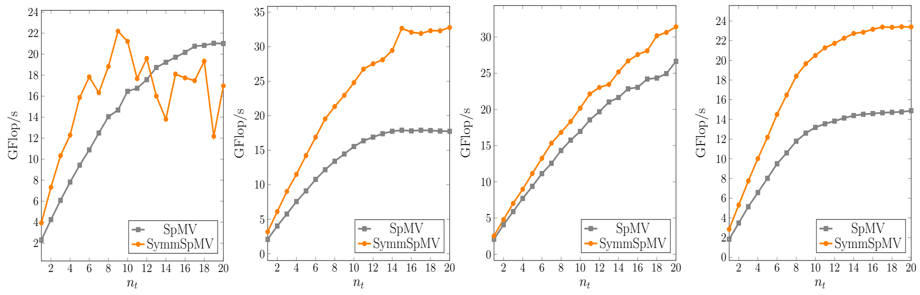
8. Experiments and Results.

The method stated above was implemented and consolidated into a library called RACE . The library provides easy interface for parallelizing kernels, user typically just needs to supply the serial code (with dependency) and hardware settings. Library will then parallelize, pin and run the code in parallel. The library is publicly available in the git repository.

8.1. Test setup.

In the following we present the performance and convergence results obtained using the library, and compare it against state of art methods. Hardware and matrices as described in section 4 is used for the following benchmarks. As mentioned in section 7 parameter ϵ_s is set to 0.8 and RCM is used in level construction stage. All the experiments conducted here are using warmed up caches i.e., we run the kernel for 100 times initially for warm-up and then we measure performance for the next 500 iteration of the same kernel. Mean performance of this 500 iterations is used to plot the results.

The matrix is per-processed with RCM for all the cases (even for SpMV), except for MC and ABMC methods. The exclusion of MC and ABMC method is due to the fact that in most of the cases performance does not improve with RCM pre-processing since they have their own re-ordering while in some case applying MC/ABMC on top of RCM re-ordering leads to performance degradation. Intel SpMP [27] library was used to do this RCM pre-processing.



(a) *crankseg_1* (b) *inline_1* (c) *parabolic_fem* (d) *Graphene-4096*

FIG. 17. *Scaling of SymmSpMV with RACE compared to SpMV on one socket of Sky-Lake architecture, for corner case matrices.*

8.2. Corner Cases. Figure 17 shows the scaling performance of SymmSpMV and SpMV (baseline) kernel for corner case matrices on one socket of Sky-Lake architecture. Chosen corner case matrices represent different aspects and bottlenecks that appear either due to RACE method or because of hardware capabilities.

The *crankseg_1* matrix is the worst in terms of performance. It does not scale well due to it's limited parallelism obtained using the RACE method. This property of *crankseg_1* is well evident directly after doing the theoretical estimate based on η as seen in Figure 16a. One could further see that the actual scaling run of the kernel seen in Figure 17a is exactly in tune with that of the theoretical result. Note that due to this bottleneck of parallelism we didn't achieve much benefit from using SymmSpMV compared to SpMV.

The *inline_1* matrix although being third lowest in terms of parallelism in the entire set of test matrices, but it still achieves a high efficiency ($\eta = 0.85$) for 20 threads (see Figure 16b), leading to good scaling as seen in Figure 17b. The saturation in performance after 15 threads is due to the fact that we hit the memory bottleneck, similar saturation behavior can also be observed for SpMV which is embarrassingly parallel. The saturation occurs at the maximum achievable performance on the given architecture which could easily be verified using the roofline model [29] and intensity equations (see subsection 4.4) as shown below:

$$I_{\text{SpMV}} = \frac{2}{8+4+\frac{8+16}{73}} = 0.162\left[\frac{\text{Flop}}{\text{byte}}\right], \text{ assuming best case : } \alpha = \frac{1}{N_{nzr}}$$

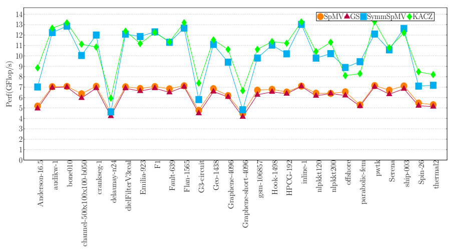
$$P_{\text{SpMV}} = b_s * I_{\text{SpMV}};\ \text{for memory-bound case}$$

$$P_{\text{SpMV}} = 0.162\left[\frac{\text{Flop}}{\text{Bytes}}\right] * 115\left[\frac{\text{GByte}}{\text{s}}\right] = 18.6\left[\frac{\text{GFlop}}{\text{s}}\right]$$

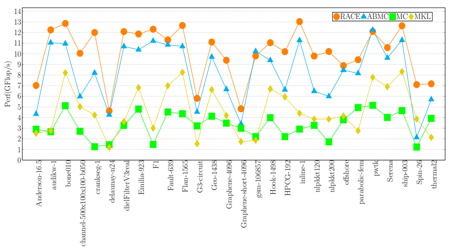
As seen we achieve 17.8 GFlop/s which is close to the theoretical maximum of 18.6 GFlop/s for SpMV. Similar derivation can be done for SymmSpMV and one could see P_{SymmSpMV}= 34.5 GFlop/s, which is approximately twice that of SpMV since I_{SymmSpMV} is almost a factor two higher than I_{SpMV} for matrix with moderate N_{nzr} (see (4.1) and (4.3)). From Figure 17b one can observe that at saturation we reach close to theoretical values. A cushioning effect due to memory bandwidth bottleneck is also evident from Figure 17b, where we see that due to this saturation decrease in η to a certain extent would not effect the socket level performance, it would just shift the knee of saturation towards right.

In the case of *parabolic_fem* matrix we theoretically have a good efficiency as seen from Figure 16c, but here we do not see any saturation in performance (see Figure 17c), even SpMV does not have this saturation behavior. If one calculates the maximum theoretical performance by roofline model and assuming memory-boundedness as shown in previous example one would see that P_{SpMV}=15 GFlop/s and P_{SymmSpMV}=19 GFlop/s, but we achieve more than these values in actual runs 26.5 and 31.5 GFlop/s respectively. This is because the matrix is small enough ($\approx$ 46 MB for full matrix and $\approx$ 23 MB for symmetric storage) to just fit in caches (combined L2 and L3) of the Sky-Lake architecture. Since the caches scales well on this architecture we don't observe the saturation behavior. It should be noted that in this case comparison between SpMV and SymmSpMV cannot be done directly since for SpMV the total data is almost close to cache limits, while for SymmSpMV it would easily fit in cache.

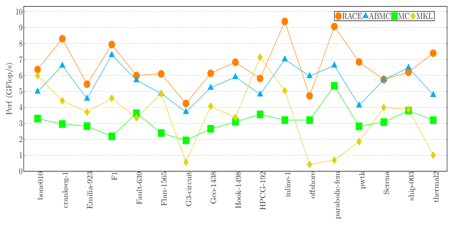
Graphene-4096 matrix on the other hand is a matrix with efficiency similar to *parabolic_fem* but with much larger size ($\approx$ $2GB$) resulting in matrix data always coming from main memory. This therefore shows dominant saturation behavior and since we achieve good efficiency (η) the knee of saturation begins at a well early stage for SymmSpMV compared to the case of *inline_1* where the efficiency was lower in comparison resulting in smaller n_t^{eff}.



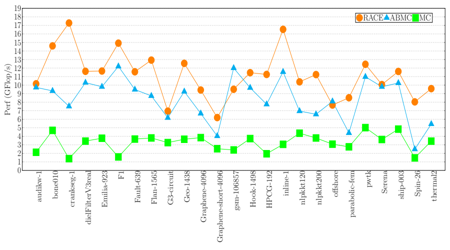
(a) RACE performance compared to SpMV



(b) SymmSpMV Comparison



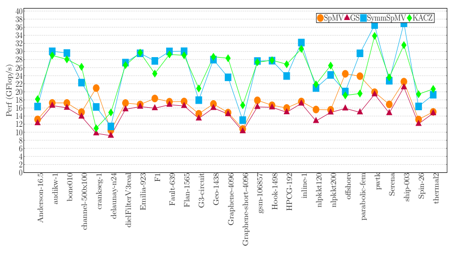
(c) GS Comparison

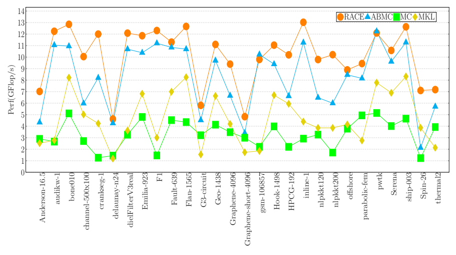


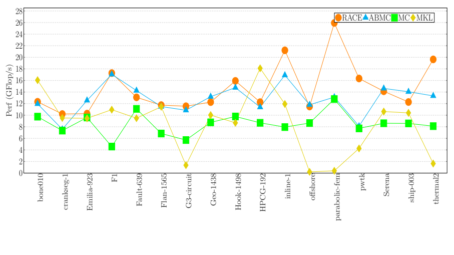
(d) KACZ Comparison

FIG. 18. *Performance results on Ivy-Bridge*

8.3. Performance and comparisons.


(a) RACE performance compared to SpMV


(b) SymmSpMV Comparison


(c) GS Comparison

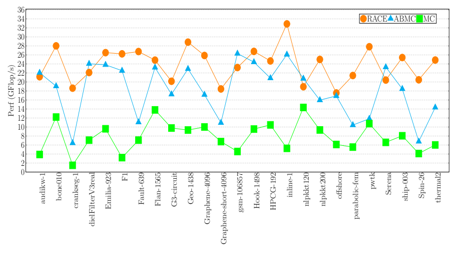
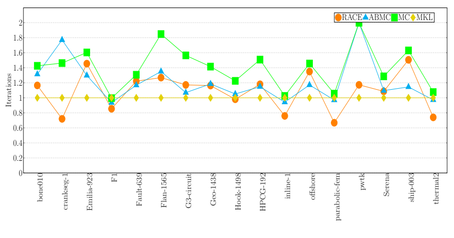

(d) KACZ Comparison

FIG. 19. *Performance results on Sky-Lake*

8.3.1. RACE performance. Here we plot the performance of SymmSpMV, GS and KACZ with RACE compared to SpMV. Figures 18a and 19a will be used here. This is done for entire test matrices and all the hardwares.

8.3.2. Comparison exact kernel. Here we compare RACE with ABMC, MC and MKL for SymmSpMV. Figures 18b and 19b will be used. COLPACK [11] was used for multicoloring (MC). METIS [18] was used for graph partitioning for ABMC, and COLPACK was used for coloring the hyper graph. The blocksize for ABMC is chosen by doing parameter scan over 4 to 128 as shown by Iwashita et al. in [15], and choosing the optimal one. Note that the time for this parameter search is not included in the performance results shown.


(a) SymmGS iterations required by different methods compared to exact MKL kernel

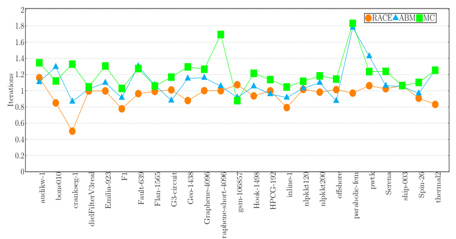

(b) SymmKACZ iterations required by different methods compared to exact Serial kernel

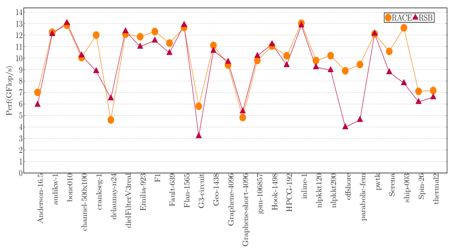
FIG. 20. *Convergence behavior of SymmGS and SymmKACZ at 20 threads*

8.3.3. Comparison iterative kernel. Here we compare RACE with ABMC, MC and MKL for GS and KACZ. Here we include iterations also in performance metric. Figures 18c, 18d, 19c, and 19d will be used. Actual iteration behavior can be seen in Figure 20.

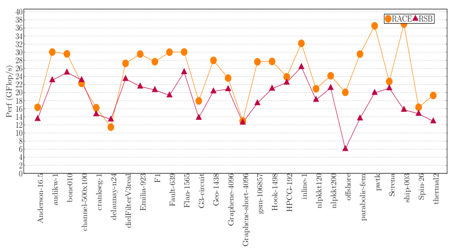
Matrices only compatible with the solvers (GS, KACZ) are shown in performance results.

The exact implementation of MKL for SymmGS is not explicitly stated and is not published. But due to the property of the solver having same convergence as serial case we believe level-scheduling is used. The usage of same kernels in Intel's implementation of HPCG benchmark where the usage of level-scheduling has been stated [25] leads to more confidence in our assumption.

8.3.4. Comparison with tailored data format. Comparison of RACE with RSB data format. Note RSB is pre-processed with RCM, which improves its performance for some cases. Figures 21a and 21b shows this comparison.



(a) Comparison of RACE with RSB on Ivy-Bridge



(b) Comparison of RACE with RSB on Sky-Lake

FIG. 21. *Comparison with RSB data format*

9. Conclusion.

10. Future Work.

Acknowledgments. We would like to acknowledge the assistance of volunteers in putting together this example manuscript and supplement.

REFERENCES

[1] *Equipping Sparse Solvers for Exascale - ESSEX*. https://blogs.fau.de/essex/activities.

[2] *SuiteSparse Matrix Collection*. https://sparse.tamu.edu/.

[3] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, *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, New York, NY, USA, 2009, ACM, pp. 233–244, https://doi.org/10.1145/1583991.1584053.

[4] J. Díaz, J. Petit, and M. Serna, *A survey of graph layout problems*, ACM Comput. Surv., 34 (2002), pp. 313–356, https://doi.org/10.1145/568522.568523.

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

[6] A. Elafrou, V. Karakasis, T. Gkountouvas, K. Kourtis, G. Goumas, and N. Koziris, *Sparsex: A library for high-performance sparse matrix-vector multiplication on multicore platforms*, ACM Trans. Math. Softw., 44 (2018), pp. 26:1–26:32, https://doi.org/10.1145/3134442.

[7] T. Elfving, *Block-iterative methods for consistent and inconsistent linear equations*, Numerische Mathematik, 35 (1980), pp. 1–12, https://doi.org/10.1007/BF01396365, https://doi.org/10.1007/BF01396365.

[8] D. J. Evans, *Parallel S.O.R. Iterative Methods*, Parallel Comput., 1 (1984), pp. 3–18, https://doi.org/10.1016/S0167-8191(84)90380-6.

[9] M. Galgon, L. Krämer, J. Thies, A. Basermann, and B. Lang, *On the parallel iterative solution of linear systems arising in the feast algorithm for computing inner eigenvalues*, Parallel Comput., 49 (2015), pp. 153–163, https://doi.org/10.1016/j.parco.2015.06.005.

[10] A. H. Gebremedhin, F. Manne, and A. Pothen, *Parallel distance-k coloring algorithms for numerical optimization*, in Proceedings of the 8th International Euro-Par Conference on Parallel Processing, Euro-Par '02, London, UK, UK, 2002, Springer-Verlag, pp. 912–921, http://dl.acm.org/citation.cfm?id=646667.699892.

[11] A. H. Gebremedhin, D. Nguyen, M. M. A. Patwary, and A. Pothen, *Colpack: Software for graph coloring and related problems in scientific computing*, ACM Trans. Math. Softw., 40 (2013), pp. 1:1–1:31, https://doi.org/10.1145/2513109.2513110.

[12] T. Gkountouvas, V. Karakasis, K. Kourtis, G. Goumas, and N. Koziris, *Improving the performance of the symmetric sparse matrix-vector multiplication in multicore*, in 2013 IEEE 27th International Symposium on Parallel and Distributed Processing, May 2013, pp. 273–283, https://doi.org/10.1109/IPDPS.2013.43.

[13] D. Gordon and R. Gordon, *Component-averaged row projections: A robust, block-parallel scheme for sparse linear systems*, SIAM Journal on Scientific Computing, 27 (2005), pp. 1092–1117, https://doi.org/10.1137/040609458.

[14] Intel, *Intel math kernel library*, https://software.intel.com/en-us/mkl (accessed 2018/05/22).

[15] T. Iwashita, H. Nakashima, and Y. Takahashi, *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, Washington, DC, USA, 2012, IEEE Computer Society, pp. 474–483, https://doi.org/10.1109/IPDPS.2012.51, http://dx.doi.org/10.1109/IPDPS.2012.51.

[16] M. T. Jones and P. E. Plassmann, *Scalable iterative solution of sparse linear systems*, Parallel Comput., 20 (1994), pp. 753–773, https://doi.org/10.1016/0167-8191(94)90004-3, http://dx.doi.org/10.1016/0167-8191(94)90004-3.

[17] C. Kamath and A. Sameh, *A projection method for solving nonsymmetric linear systems on multiprocessors*, 9 (1989), pp. 291–312, https://doi.org/10.1016/0167-8191(89)90114-2.

[18] G. Karypis and V. Kumar, *A fast and high quality multilevel scheme for partitioning irregular graphs*, SIAM Journal on Scientific Computing, 20 (1998), pp. 359–392, https://doi.org/10.1137/S1064827595287997.

[19] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop, *A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units*, SIAM Journal on Scientific Computing, 36 (2014), pp. C401–C423, https://doi.org/10.1137/130930352.

[20] C. Y. Lee, *An algorithm for path connections and its applications*, IRE Transactions on Electronic Computers, EC-10 (1961), pp. 346–365, https://doi.org/10.1109/TEC.1961.5219222.

[21] H. Lu, M. Halappanavar, D. Chavarra-Miranda, A. H. Gebremedhin, A. Panyala, and A. Kalyanaraman, *Algorithms for balanced graph colorings with applications in parallel computing*, IEEE Transactions on Parallel and Distributed Systems, 28 (2017), pp. 1240–

1256, https://doi.org/10.1109/TPDS.2016.2620142.

[22] M. Martone, *Efficient multithreaded untransposed, transposed or symmetric sparse matrix-vector multiplication with the recursive sparse blocks format*, Parallel Comput., 40 (2014), pp. 251–270, https://doi.org/10.1016/j.parco.2014.03.008, http://dx.doi.org/10.1016/j.parco.2014.03.008.

[23] K. Nakajima and H. Okuda, *Parallel iterative solvers for unstructured grids using an openmp/mpi hybrid programming model for the geofem platform on smp cluster architectures*, in High Performance Computing, H. P. Zima, K. Joe, M. Sato, Y. Seo, and M. Shimasaki, eds., Berlin, Heidelberg, 2002, Springer Berlin Heidelberg, pp. 437–448.

[24] J. Park, M. Smelyanskiy, N. Sundaram, and P. Dubey, *Sparsifying synchronization for high-performance shared-memory sparse triangular solver*, in Proceedings of the 29th International Conference on Supercomputing - Volume 8488, ISC 2014, New York, NY, USA, 2014, Springer-Verlag New York, Inc., pp. 124–140, https://doi.org/10.1007/978-3-319-07518-1_8.

[25] J. Park, M. Smelyanskiy, K. Vaidyanathan, A. Heinecke, D. D. Kalamkar, X. Liu, M. M. A. Patwary, Y. Lu, and P. Dubey, *Efficient shared-memory implementation of high-performance conjugate gradient benchmark and its application to unstructured matrices*, in SC14: International Conference for High Performance Computing, Networking, Storage and Analysis, Nov 2014, pp. 945–955, https://doi.org/10.1109/SC.2014.82.

[26] Y. Saad, *Iterative Methods for Sparse Linear Systems*, Society for Industrial and Applied Mathematics, second ed., 2003, https://doi.org/10.1137/1.9780898718003, https://epubs.siam.org/doi/abs/10.1137/1.9780898718003, https://arxiv.org/abs/https://epubs.siam.org/doi/pdf/10.1137/1.9780898718003.

[27] SpMP Development Team, *Sparse matrix pre-processing library*, https://github.com/IntelLabs/SpMP (accessed 2018/05/22).

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

[29] S. Williams, A. Waterman, and D. Patterson, *Roofline: An Insightful Visual Performance Model for Multicore Architectures*, Commun. ACM, 52 (2009), pp. 65–76, https://doi.org/10.1145/1498765.1498785.

Appendix A. Algorithms.

Algorithm A.1 Construction of levels

```
Choose starting node(s) = $\{n\}$
$marked\_all$ = false
$N = nrows(graph)$
$distFromRoot[1..N] = -1$
$curr\_children.push\_back(n)$;
$currLvl = 0$
**while** !$marked\_all$ **do**
    $marked\_all$ = true
    $nxt\_children = \{\}$
    **for** $i = 1 : size(curr\_children)$ **do**
        **if** $distFromRoot[curr\_children[i]] == -1$ **then**
            $distFromRoot[curr\_children[i]] = currLvl$
            **for** $j$ in $graph[curr\_children[i]].children$ **do**
                **if** $distFromRoot[j] == -1$ **then**
                    $nxt\_children.push\_back(j)$
                **end if**
            **end for**
        **end if**
    **end for**
    $curr\_children = nxt\_children$
    $currLvl = currLvl + 1$
**end while**
```

Algorithm A.2 Load Balancing for two sweep, distance-2

```
1: $num\_sweep = 2$                    % two sweep method
2: $minGap = 2$                        %distance-2
3: $len = num\_sweep * nthread$        % constructing nthread parallel work
4: while !($exit$) do
5:   $T\_size$ = update($T\_ptr$)      %$T\_size$ contains non-zeros in each $level\ group$
6:   $mean\_r$ = sum($T\_size[0 : num\_sweep : len]$) / $nthreads$
7:   $mean\_b$ = sum($T\_size[1 : num\_sweep : len]$) / $nthreads$
8:   $diff[0 : num\_sweep : len] = T\_size[0 : num\_sweep : len]. - mean\_r$
9:   $diff[1 : num\_sweep : len] = T\_size[1 : num\_sweep : len]. - mean\_b$
10:  $var$ = dot_product($diff, diff$)
11:  $absRankIdx$ = sortIdx(abs($diff$)) % sortIdx returns permutation after
12:                                      % sorting from bigger to larger
13:  $rankIdx$ = sortIdx($diff$)
14:  $currRank = 0, newVar = var$
15:  $old\_T\_ptr = T\_ptr$
16:  while $newVar \geq var$ do
17:    $T\_ptr = old\_T\_ptr$
18:    $fail$=true
19:    if $diff[absRankIdx[currRank]] < 0$ then
20:      for $el$ in $rankIdx[(len - 1) : -1 : 0]$ do
21:        if $(T\_Ptr[el + 1] - T\_ptr[el]) > min\_gap$ then
22:          $acquireIdx$ = el
23:          $fail$=false
24:          $break$
25:        end if
26:      end for
27:      shift($T\_ptr, acquireIdx, currRank$) % shifts $T\_ptr$ by 1 from $acquireIdx$
28:                          % to $currRank$ if $currIdx < acquireIdx$ else shift by -1
29:    else if $(T\_ptr[currRank + 1] - T\_ptr[currRank]) > min\_gap$ then
30:      $giveIdx = rankIdx[0]$
31:      $fail$=false
32:      shift($T\_ptr, currRank, giveIdx$)
33:    end if
34:    if !$fail$ then
35:      $newVar$ = calculate_variance($T\_ptr$) % as seen in Line 5 to Line 10
36:    end if
37:    if $(currRank == (len - 1))$ && $(newVar \geq var)$ then
38:      $T\_Ptr = old\_T\_ptr$
39:      $exit$ = true
40:      $break$
41:    end if
42:    $currRank+ = 1$
43:  end while
44: end while
```