RACE*

Abstract. This is an example SIAM LATEX article. This can be used as a template for new articles. Abstracts must be able to stand alone and so cannot contain citations to the paper's references, equations, etc. An abstract must consist of a single paragraph and be concise. Because of online formatting, abstracts must appear as plain as possible. Any equations should be inline.

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 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 [6]. Later Kamath and Sameh introduced a two-block partitioning scheme for parallelizing Kaczmarz method on tridiagonal structures [12]. A general study on the convergence of these block methods were done by Elfving in 1980 [5].

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 [11], but is sometimes not efficient on modern cache-based processors. There have been several researches going on to increase the efficiency of multi-coloring and improving the heuristics, an overview of the methods can be found in [14]. One of the most successful method in this regard is the algebraic block multi-coloring [10] 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 [20]. Extensions to this approach can be seen in [16] where a hybrid approach between multi-coloring and hyperplane method is used. Most recent work in this direction can be attributed to Park et al. on his work with level-scheduling for triangular solvers [17].

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 [4]. When it comes to distance-2 kernels popular methods seen in the literature are locking based methods, thread private local vectors [8] 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) [15].

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 pattern. 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 in detail 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 socket 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 a different approaches are used for parallelizing between nodes [4, 9].

As a final application run we demonstrate the parallelization of an eigen-value solver called FEAST [18], where we use an iterative inner linear solver based on Kaczmarz method. The result presented is the first to achieve such high performance on node level for an iterative solver and is superior to the previous results published [7].

93   **4. Test bed, matrices and kernels.**

94   **4.1. Test bed.** The tests are conducted on three different multi-core architec-
95   tures. Two of them being Intel's Ivy-Bridge and modern Sky-Lake architecture, the
96   choice of these architectures enables a good comparison of our method on both Low
97   Core Count (LCC) and High Core Count (HCC) architectures, which would play an
98   important role in the convergence of the method we develop. As a third choice we
99   select AMD's recent Epyc architecture, which is competitive to Intel Sky-Lake archi-
100 tecture. This choice enables us to study the effect of our method on chips based on
101 completely different microarchitecture, enabling us to demonstrate the applicability
102 of our method on wide range of architectures. All the tests are conducted on a single
103 socket of these architectures.

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

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

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 - load only)	47 GB/s	110 GB/s	130 GB/s
Bandwidth - copy	40 GB/s	110 GB/s	114 GB/s
Architecture specific flag	-	-xCORE-AVX512	-

123 The code was compiled with newest Intel compiler version 18 and the following
124 compiler flags were set `-fno-alias -xHost -O3`. Furthermore all the measurements
125 were done with CPU clock speeds fixed at frequencies indicated in Table 1.

126   **4.2. Benchmark Matrices.** All the test matrices are taken from SuiteSparse
127 Matrix Collection (former University of Florida Sparse Matrix Collection) [2] and
128 ESSEX project [1]. The selection of the matrices from SuiteSparse Matrix Collection
129 is mainly done by combining the test matrices from two papers [15, 17]. This enables
130 easy comparison of results. Matrices from ESSEX project are some of the matrices
131 that are of interest in the FEAST eigen value solver. Only matrices having undirected
132 graphs are considered due to scope of the paper as mentioned in section 3. Matrices
133 along with some of their parameters are given in Table 2. Matrices that have been
134 marked with an * symbol indicate they are corner cases and will be discussed in detail.

TABLE 2
Benchmark matrices

Index	Matrix name	nrows	nnz	bandwidth		
1	audikw_1	943695	77651847	925946		
2	bone010	986703	71666325	13016		
3	channel-500x100x100-b050	4802000	85362744	600299		
4	crankseg_1	52804	10614210	50388	*	
5	delaunay_n24	16777216	100663202	16769102		
6	dielFilterV3real	1102824	89306020	1036475		
7	Emilia_923	923136	41005206	17279		SuiteSparse Matrix Collection
8	F1	343791	26837113	343754		
9	Fault_639	638802	28614564	19988		
10	Flan_1565	1564794	117406044	20702		
11	G3_circuit	1585478	7660826	947128		
12	Geo_1438	1437960	63156690	26018		
13	gsm_106857	589446	21758924	588744		
14	Hook_1498	1498023	60917445	29036		
15	HPCG-192	7077888	189119224	37057		
16	inline_1	503712	36816342	502403		
17	nlpkkt120	3542400	96845792	1814521		
18	nlpkkt200	16240000	448225632	8240201	*	
19	offshore	259789	4242673	237738	*	
20	parabolic_fem	525825	3674625	525820	*	
21	pwtk	217918	11634424	189331		
22	Serena	1391349	64531701	81578		
23	ship_003	121728	8086034	3659	*	
24	thermal2	1228045	8580313	1226000	*	
25	Anderson-16.5	2097152	14680064	1198372		
26	Graphene-2048	4194304	16771072	2048		ESSEX
27	Graphene-4096	16777216	67096576	4096		
28	Spin-26	10400600	145608400	709995	*	

135   **4.3. Kernels.** To test the performance we choose algorithms that are exact
136 as well as iterative. Also we include kernels from both distance-1 and distance-2
137 dependency classes. All the kernels shown below are based on CRS matrix storage
138 format.

139   **4.3.1. SpMV .** Sparse Matrix Vector (SpMV) is a kernel that do not have
140 any dependencies. It acts as a good reference for other kernels to determine their
141 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**

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

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

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 [13].

4.3.2. 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:

$$(4.2) \qquad I_{SpMTV} = \frac{2}{8 + 4 + 16 * \alpha + \frac{8}{N_{nzr}}}$$

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.

4.3.3. 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**

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

161 (4.3)
$$I_{SymmSpMV} = \frac{4}{8 + 4 + 32 * \alpha + \frac{16}{N_{nzr}^{symm}}}$$

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

164 **4.3.4. GS .** Gauss-Seidel (GS) is a solver having distance-1 dependency. Con-
165 trary to the above kernels GS is in-exact meaning it is an iterative method. Alg. 4.4
166 shows the Gauss-Seidel algorithm where its assumed that the diagonal entries of the
167 matrix is 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**

---

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

173 (4.4)
$$I_{GS} = I_{SPMV}$$

174 **4.3.5. KACZ.** Kaczmarz (KACZ) is an iterative solver based on row-projection
175 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**

---

176 In-core has a mixed behavior of both SpMV and SpMTV similar to SymmSpMV.
177 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.

$$(4.5) \qquad I_{KACZ} = \frac{4}{8 + 4 + 16 * \alpha + \frac{8}{nnzr}} = 2 * I_{SpMTV}$$

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) [7] but it was soon found that the kernels do not scale efficiently with this approach.

(a) α effect

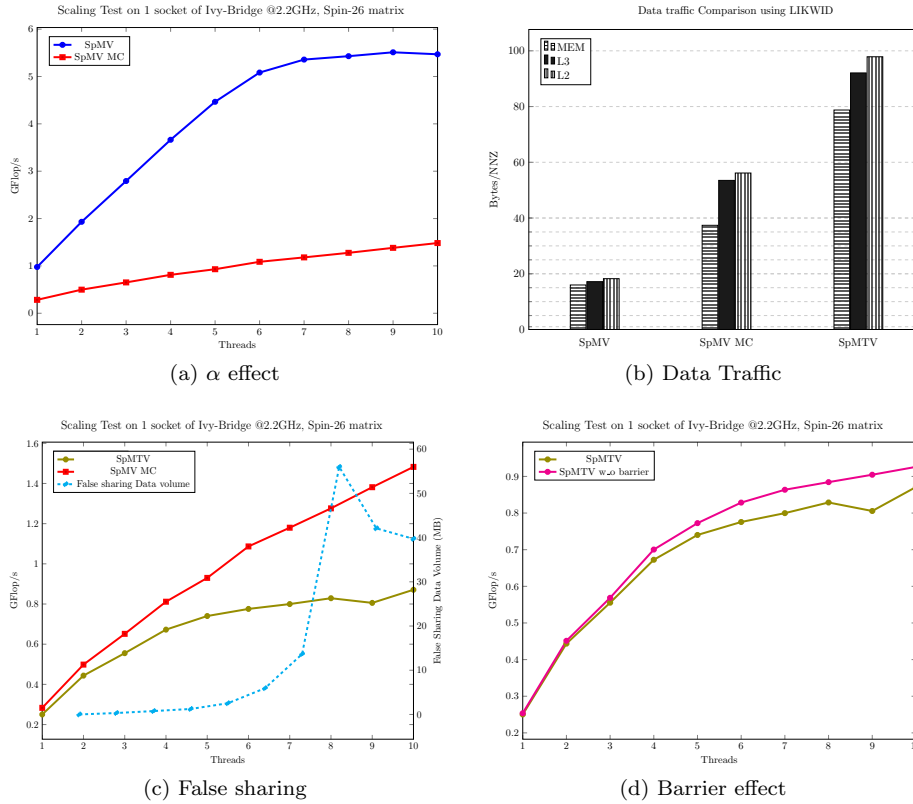(b) Data Traffic

(c) False sharing

(d) Barrier effect

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 five 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 [22]. This could easily be demonstrated by measuring the data traffic between different memory hierarchies. We do this using the LIKWID tool [21], 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.
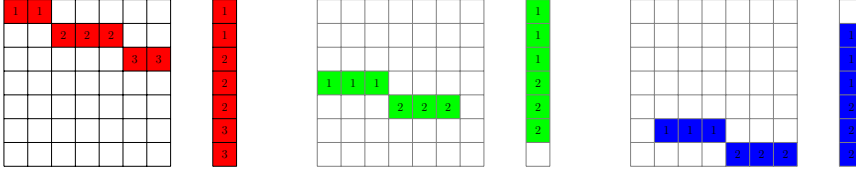

FIG. 2. *Illustration of increase in α by multicoloring, numbers represents thread numbers working on a particular row*

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)). This further decreases performance of SpMTV compared to SpMV on MC matrix which can be verified in Figure 1c.

Other contributors to the drop in performance is the global synchronizations and false sharing occurring while executing the kernel. These factors strongly depend on the number of colors and in general increase with colors required to resolve the dependency. A rough estimate for the barrier/synchronization overhead can be evaluated by comparing performance between kernel with (correct result) and without barriers (incorrect result) as seen in Figure 1d. Figure 1c shows the volume of false-sharing as measured by LIKWID for SpMTV kernels.For SpMV the amount of false sharing is negligible and is close to zero since it does not have indirect writes. It has to be noted that LIKWID measurements for counters like false sharing, might not be 100% accurate, but they provide a good qualitative estimate for comparison, as seen here counter indicates increase in false sharing with number of threads. More on counter validation can be found in [19].

It was seen that for most of the matrices arising in the project the average drop in performance by multi-coloring was almost a factor of two on a single socket of Ivy-Bridge . Although for most of the matrices the performance could be improved by algebraic block multi-coloring (ABMC), still the results we obtained were not optimal mainly for large matrices when compared to performance models which we

236 will see later in section 7. This led to the development of a method which works on
237 a common data format like CRS in which most of the other kernels are written and
238 at the same time preserves data locality, reduce synchronization overheads and false
239 sharing.

240   **6. RACE method.** Keeping in mind the observations from previous section 5,
241 one could observe that it would be best to maintain the non-zeros of matrix close to the
242 diagonal. This has been observed previously in the regard of normal sparse matrix
243 computations like SpMV and has led to the pre-processing of matrix by applying
244 bandwidth reduction algorithms like "Reverse Cuthill McKee " (RCM). Now we aim
245 to develop a method that does not distort this ideal permutations to a large extent
246 but at the same time resolve distance-k dependencies.
247   Our approach can be seen as a recursive level based method. Each step of the
248 method basically consists of four steps namely:
249   1. Level construction
250   2. Permutation
251   3. Distance-k coloring
252   4. Load balancing
253 The method is strongly coupled to the hardware underneath and exploits only the
254 parallelism as required by the hardware. If at the end of all these four steps one
255 does not achieve sufficient parallelism, all the steps are recursively applied to selected
256 sub-graphs of the matrix until sufficient parallelism is attained. This recursive nature
257 of our coloring method led to the naming of the method as "Recursive Algebraic
258 Coloring Engine " in short RACE .
259   To explain the method in an easier and illustrative way we choose a simple matrix
260 namely the 2D 7pt. stencil. The sparsity pattern and the corresponding graph of the
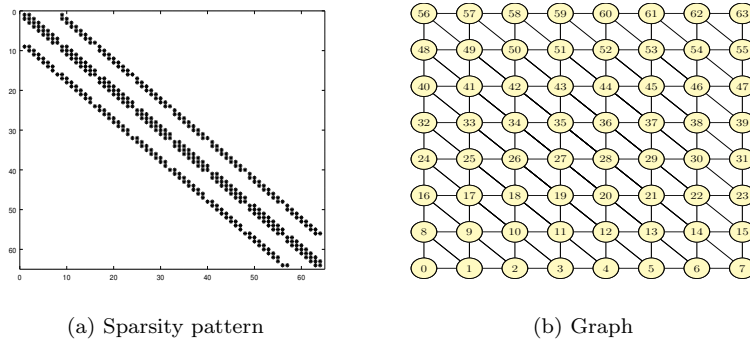261 matrix is as shown in Figure 3.

(a) Sparsity pattern   (b) Graph

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

262   **Definitions.** The following basic definitions from graph theory are used in the
263 following sections:
264   • **Graph :** $G = (V, E)$ represents a graph where $V(G)$ belongs to set of
265   vertices and $E(G)$ represents the edges in the graph. Note that here we
266   specifically denote G for fully connected undirected graphs.
267   • **Neighborhood :** Neighborhood of vertex u represented as $N(u)$ is defined

268   as:

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

270   • **Subgraph :** A subgraph H of graph G in this paper specifically refers to
271   subgraph induced by $V' \subseteq V(G)$ and is defined as

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

273   **6.1. Level Construction.** The first step of the RACE method is level construc-
274   tion. The step concerns with finding different *levels* in the graph. First *level* $(L(0))$
275   is chosen to consist of a selected a root vertex. Level i $(L(i))$ for $i > 0$ is defined to
276   contain vertices that are in neighborhood of vertices in previous *level* $(L(i-1))$ and
277   not in $L(i-2)$ i.e.,

278   (6.1) $$L(i) = \begin{cases} u : u \in N(L(i-1)) \cap N(L(i-2)))' & \text{if } i \neq 0 \\ root & otherwise \end{cases}$$

279   These *levels* are same to the ones found in "Breadth First Search" (BFS) traversal
280   and aims to find the minimum distance from the root node. Note that this is substan-
281   tially different to the *levels* in methods like "level-scheduling" [20] where maximum
282   distance is sought after. Algorithm A.1 shows an algorithm to find each nodes min-
283   imum distance from root, $L(i)$ consists of all the nodes having a distance of i from
284   the root. Total number of levels obtained with this graph traversal will be denoted
285   as *total_level* . Figure 4a shows *levels* on the 2d-7pt stencil ($total\_level = 14$), the
286   main number on each vertex refers to the vertex number and the superscript shows
287   the *level* number, i.e.,

288   (6.2) $$n^i \implies n \in L(i)$$

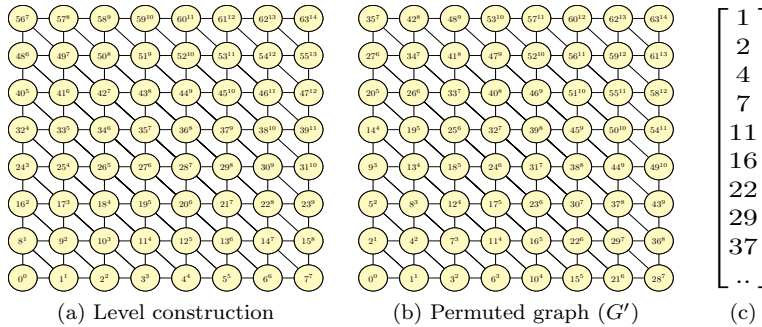(a) Level construction   (b) Permuted graph (G')   (c)

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

289   **6.2. Permutation.** Once the *levels* are known one has to permute the matrix
290   in the order of its *levels* , such that vertices in $L(i)$ appears before that of $L(i-1)$.
291   Till this step the procedure is similar to that of BFS pre-processing for bandwidth
292   reduction. One could also replace BFS with better bandwidth reduction algorithms
293   like "(Reverse) Cuthill McKee ". Figure 4 shows the graph ($G' = P(G)$) of 2d-
294   7pt stencil matrix after this permutation (P) is applied. Observe the difference in

295 node numbering between original lexicographic ordering in Figure 4a and Figure 4b.
296 Now the most important step for resolving dependencies (coloring) is to store the
297 information about *levels* . In order to do this we use a data structure called `level_ptr`
298 . It stores the starting vector of each *levels* , which implies that *levels* on G' can be
299 identified as:

300
$$L(i) = \{\, u : u \in [\texttt{level\_ptr}[i] : (\texttt{level\_ptr}[i+1] - 1)] \text{ and } u \in V(G') \,\}$$

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

304 **6.3. Distance-k coloring.** Two vertices are called distance-k neighbours if the
305 shortest path connecting them consists of at most k edges. This implies u is a distance-
306 k neighbour of v (denoted as $u \xrightarrow{k} v$) if

307 (6.3)
$$u \xrightarrow{k} v \iff v \in \{\, u \cup N(u) \cup N^2(u) \cup ... N^k(u) \,\}$$

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

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

312   *Proof.* We prove by contradiction. Let there exist $u, v \in V(G')$ such that $u \in L(i)$
313 and $v \in L(i \pm (k+j)) \forall j \geq 1$. Assume u, v are distance-k neighbours ($u \xrightarrow{k} v$). From
314 (6.1), (6.3) and the fact G' is undirected we get

315
$$u \xrightarrow{k} v \iff v \in \{\, L(i) \cup L(i \pm 1) \cup ... \cup L(i \pm k) \,\}$$
316
$$\implies v \notin L(i \pm (k+j)) \ \forall j \geq 1$$
317

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

320   Corollary 6.1 implies that if we leave a gap of *at least* one *level* between any
321 two *levels* ($L(i), L(i+2)$ for example) all the vertices between them are distance-1
322 independent. Similarly if there is a gap of *at least* two *levels* between any two *levels*
323 ($L(i), L(i+3)$ for example) we get distance-2 independent levels.
324   Due to this weak definition in Corollary 6.1 there exists many possibility to make
325 *levels* independent of each other and Figure 5 shows one such possibility each for
326 distance-1 and distance-2 independent *levels* . One could group some of the nearby
327 *levels* together to form a *level group* , and make this distance-1 or distance-2 in-
328 dependent of other *level groups* . The i-th *level group* would be denoted by $T(i)$.
329 Difference between *level* and *level group* can be seen in Figure 5b, for Figure 5a
330 *level group* and *level* coincides.
331   In principle one could compute on all independent *level groups* in parallel, but
332 serial within a *level group* , i.e. for example in Figure 5b $T(0), T(2), T(4), T(6)$ can be
333 operated by four different threads in parallel and in the next sweep rest *level groups*
334 . For the configurations seen in Figure 5 this would mean we have $\frac{total\_level}{2}$ and
335 $\frac{total\_level}{4}$ parallelism for distance-1 and distance-2 kernels respectively.
336   But the problem with the configurations like the one seen in Figure 5 is that there
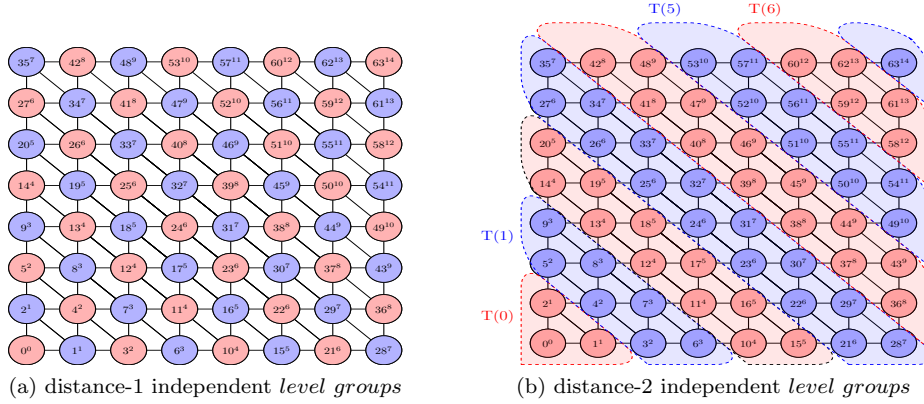337 is load imbalances between threads as the non-zeros per *level group* is not distributed

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

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

338   evenly. As seen here in the case of 2d-7pt stencil the threads working on extreme ends
339   of graph (e.g., $T(1), T(7)$) have small amount of work compared to the threads working
340   on middle (e.g., $T(3), T(4)$).

341   **6.4. Load balancing.** Depending on the matrix each *level group* would con-
342   tain different number of non-zeros, which leads to load imbalances as seen above in
343   subsection 6.3. In order to avoid this problem we employ a load balancing scheme.
344   At this step we plug in detail from hardware side like total parallelism. The idea is
345   to exploit only the parallelism as required by the hardware while at the same time
346   maintain distance-k constraint seen in Corollary 6.1. To balance the load more nearby
347   *levels* would be added to a *level group* which has less number of non-zeros and at
348   *level group* where we have considerably big *levels* only sufficient amount of *levels* to
349   maintain distance-k constraint would be assigned. Assigning nearby levels instead of
350   a random level further helps in preserving data locality.
351   An algorithm for load balancing can be found in Algorithm A.2. The aim of the
352   algorithm is to reduce combined variance of number of non-zeros in each *level group*
353   ($T\_size(i)$ refers to non-zeros in *level group* i). It does this by calculating mean and
354   variance of T_size in each parallel sweeps. For example in Figure 5b we need to
355   calculate mean of T_size of all *level groups* in red sweep and blue sweep separately.
356   The combined variance is then found by summing up the variances in each parallel
357   sweep. In order to reduce this combined variance we select the *level group* that
358   has biggest absolute deviation from mean and try to add/remove levels to/from this
359   *level group* from/to a *level group* that has biggest/least signed deviation. While
360   removing *levels* from a *level group* one has to take care that the distance-k coloring
361   is not violated, for example in case of distance-2 and two sweep scheme like seen in
362   Figure 5b we need to ensure at least two levels remain in a *level group* . To aid this
363   shifting of *levels* to/from *level group* we use the pointers to *level group* denoted by
364   T_ptr. Doing this process in an iterative way finally we end up in a state with lowest
365   combined variance at which no further moves are possible either due to violation of
366   distance-k dependency or due to increase in combined variance. Figure 6 shows step
367   by step procedure involved in load balancing and Figure 7 shows *level groups* after
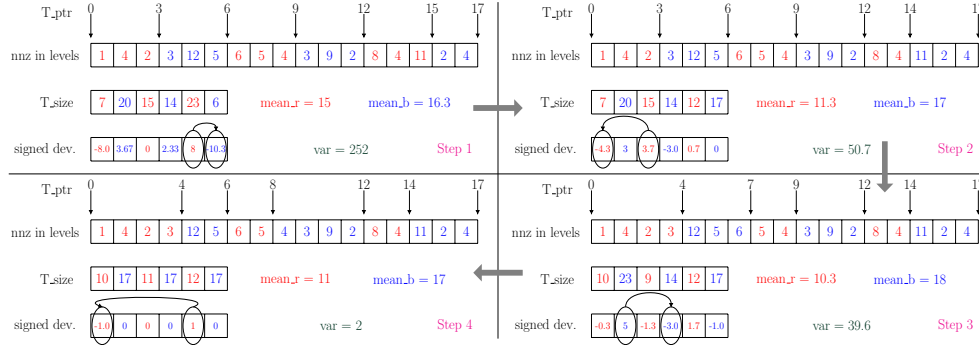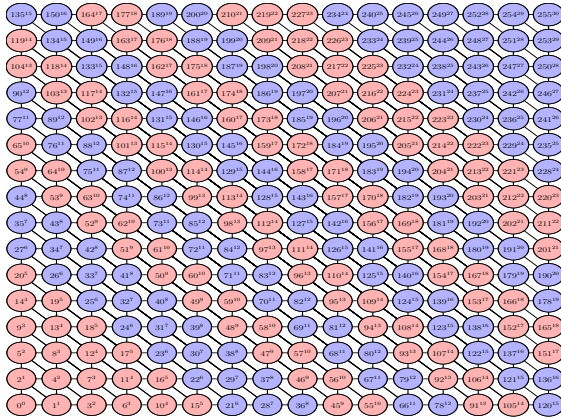368   load balancing applied on 2d-7pt stencil example of size 16×16.

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

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 number of non-zeros in each level , while level groups in middle having bigger levels maintain two levels to preserve distance-2 constraint.*

6.5. Recursion. As seen above in subsection 6.3 maximum amount of parallelism by the above approach depends on *total_level* , 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., $L_s(i)$ 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* $(T_1(3)=L_1(3))$ vertices $4 \not\xrightarrow{1} 5$ (4 distance-1 independent to 5), $4 \not\xrightarrow{1} 6$, $4 \not\xrightarrow{1} 7$ and $5 \not\xrightarrow{1} 7$, 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 $L_1(k)$.
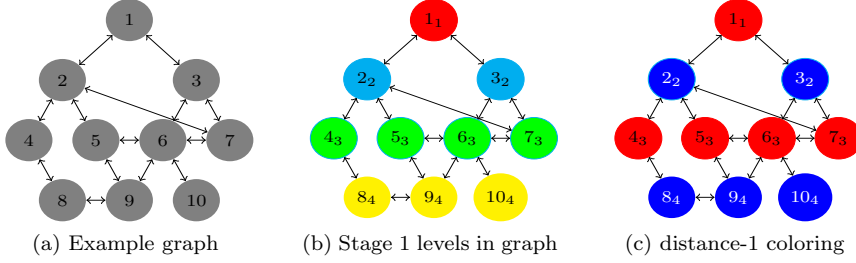In order to exploit this parallelism we use the concept of recursion.

(a) Example graph   (b) Stage 1 levels in graph   (c) distance-1 coloring

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
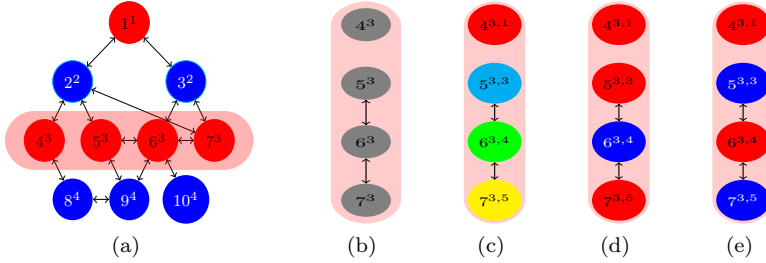explained previously (subsection 6.1 - subsection 6.4) to exploit parallelism within
this sub-graph .

(a)   (b)   (c)   (d)   (e)

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:

(6.4) $$n^{i,j,k\cdots} \implies n \in \{\, L_1(i) \cap L_2(j) \cap L_3(k) \cap \ldots \,\}$$

        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 optimum 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
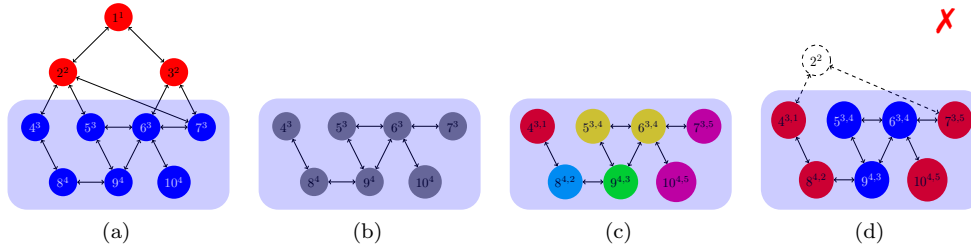since vertex 2 was not in the sub-graph considered we missed this dependency.

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

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* .

6.5.3. level_tree . By recursion we are able to exploit more parallelism. How-
ever this introduces more complexity and one has to also respect the dependencies
between stages in addition to one within stages. 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 the informations. This data structure is
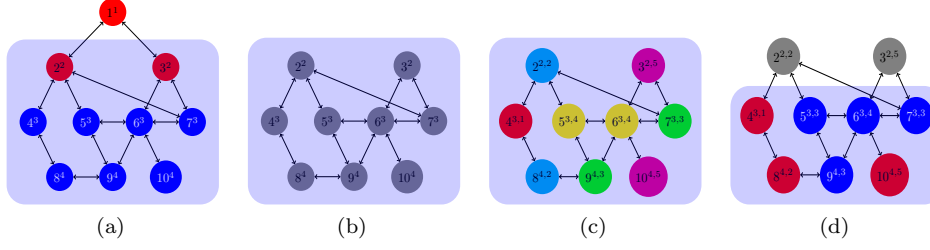called a level_tree . The root of level_tree contains information of entire domain,

FIG. 11. *Correct procedure of selecting sub-graph for distance-2 coloring. The level group $T(2)$ and it's 1st interface level is chosen as shown in Figures 11a and 11b for level construction stage seen in Figure 11c. For rest of the steps only required sub-graph to be parallelised is considered as shown in Figure 11d for distance-k coloring.*

first leaves of this root stores information about *level groups* in stage 1 (T_1), next leaves about *level groups* in stage 2 (T_2) and so on.
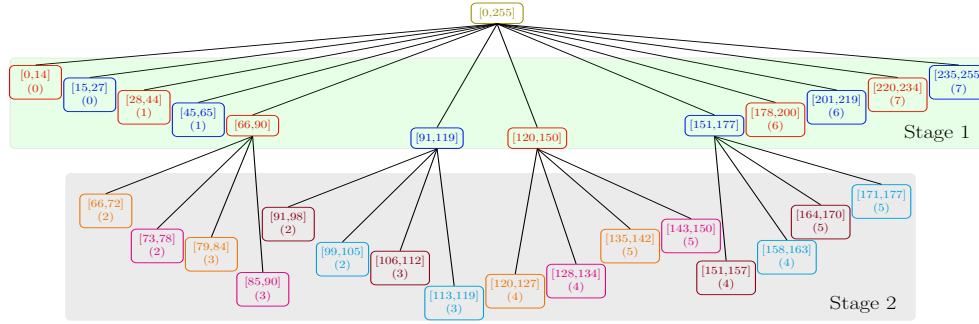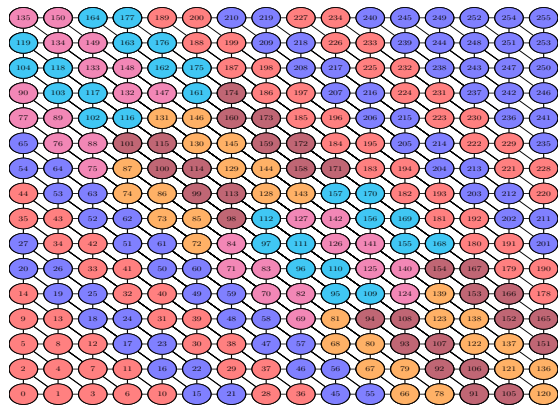

FIG. 12. `level_tree` *corresponding to 2d-7pt stencil example for domain size 16×16, and 8 threads. The range specified in each leaves represent the vertices belonging to each level group , and the number in bracket represents the thread assigned to the level group in fill type pinning.*


```
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. 13. *Graph corresponding to* `level_tree` *in Figure 12. The execution order of different level group is specified in the short code snippet on right. Note nested parallelism being used.*

Figure 12 shows a `level_tree` corresponding for 2d-7pt stencil example. Threads are assigned to each *level group* , order of which depends 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. In order to replicate this tree like parallelisation strategy we use nested parallelism, where threads in stage $k+1$ is spawned by threads in stage k. The graph corresponding to 2d-7pt stencil example is shown in Figure 13, and the execution order is specified in the figure. 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.

6.5.4. 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 12, 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

$$w(L_s(i)) = (\texttt{level\_ptr}_s[i+1] - \texttt{level\_ptr}_s[i]) * \frac{n\_threads}{n\_vertices}$$

$n\_threads$: total parallelism required by hardware

$n\_vertices$: 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:

$$\epsilon_s = 1 - abs(a - b);$$

The obtained number b is chosen as weight for *level groups* operated by first thread in the current stage i.e., $w(T_s(0)) = w(T_s(1)) = b$. 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 group* .

7. 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.

In the following we present the performance and convergence results obtained

using the library, and compare it against state of art methods. Test setup, hardware and matrices as described in section 4 is used for the following benchmarks.
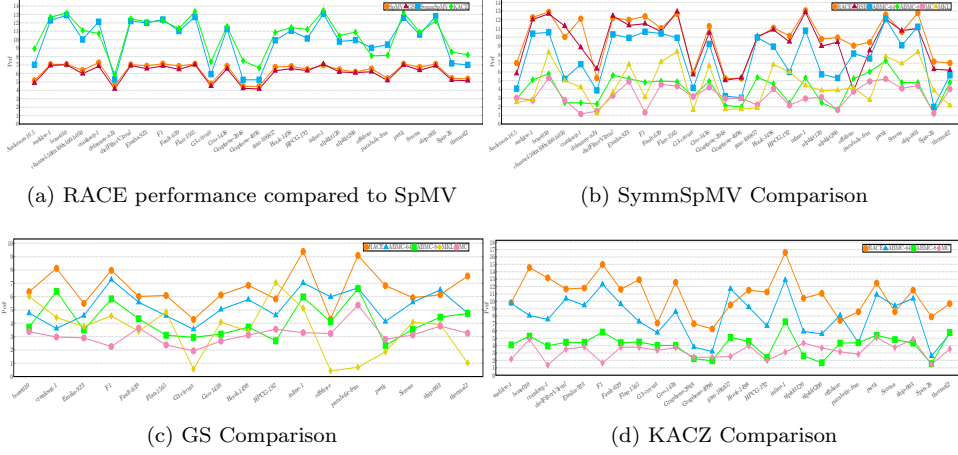

(a) RACE performance compared to SpMV

(b) SymmSpMV Comparison

(c) GS Comparison

(d) KACZ Comparison

FIG. 14. *Performance results on Ivy-Bridge*

(a) RACE performance compared to SpMV

(b) SymmSpMV Comparison

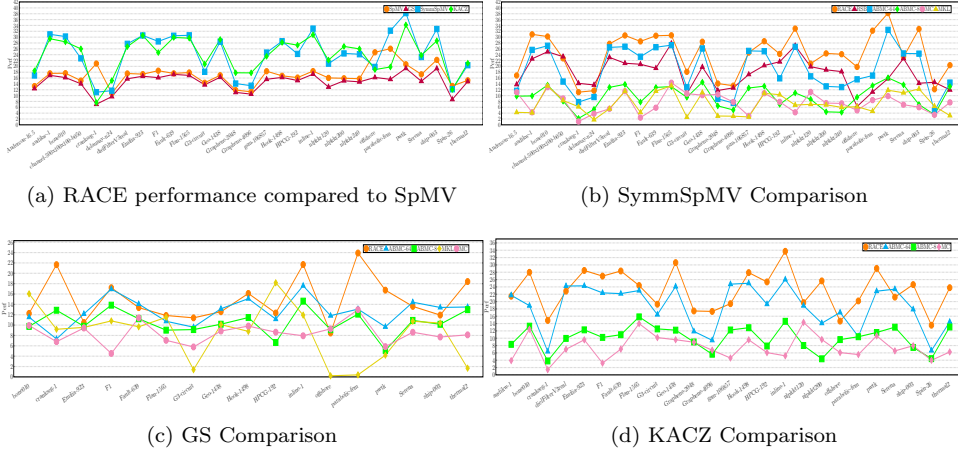(c) GS Comparison

(d) KACZ Comparison

FIG. 15. *Performance results on Sky-Lake*

7.1. Main points to discuss.

- Relate roofline model and the performance graphs of RACE compared to SpMV.
- Point out on Ivy-Bridge we reach close to ideal performance in every case, and on Sky-Lake except for corner cases like crankseg and offshore we reach close to ideal performance. The drop in corner cases like crankseg and offshore on Sky-Lake is due to lack of parallelism attained by RACE and associated load imbalances. This effect shows up on Sky-Lake rather than Ivy-Bridge since Sky-Lake has 20 threads compared to 10 on Ivy-Bridge .
- Point out that for cases like Graphene, Spin, parabolic_fem we don't see 2 fold increase in GFlop/s for KACZ, and SymmSpMV. This is due to the

fact here N_{nzr} is very small like 4, 14 and 7 which causes two problems. For KACZ kernel there is one division per row and this causes a performance drop as evident in Spin matrices, also this effect can be observed for GS kernel. For SymmSpMV kernel the N_{nzr} decreases almost by half since we operate only on upper triangular part and with short loop over N_{nzr} no effective vectorization and modulo unrolling can be done.

- Matrices like crankseg-1, and offshore are also really small making some part of data fit in cache, this is the reason why they achieve performance above RLM.

- Discuss why we chose the methods for comparison. MC and ABMC are common in literature for distance-1 coloring, MKL methods are standard library used in many productive codes, also it uses level-scheduling (not explicitly stated but we believe) for kernels like GS and enables us to compare with methods that do not disturb convergence. RSB enables to compare with methods using different data format and it has been shown this method has an upper hand in this category.

- Comparison with SymmSpMV shows the behavior of different methods for distance-2 coloring. Here we see in almost all of the case RACE and RSB has an upper hand on Ivy-Bridge . Although in some cases like offshore RACE clearly has an advantage. ABMC methods follow these methods. MKL and MC does not deliver good performance. For Sky-Lake architecture RSB falls behind ABMC, we thing this is because of the requirement of ABMC to lock rows and cols of the submatrix on which a thread is working, becoming a bottleneck at high thread counts.

- Maybe tell RSB and 16-bit integer.

- Discuss with methods like ABMC and MC the performance especially drops for large matrices like Graphene, Spin, nlpkkt due to worsening of data locality (α). Show sparsity pattern and LIKWID meaurements.

- Tell GS and KACZ performance includes also takes iterations into consideration (as shown in paper). Tell we do only a distance-1 coloring for GS and distance-2 for KACZ. We use only matrices where GS can be applied and similarly for KACZ. Also we just compare against readily available solutions. Therefore RSB is left out for GS and RSB and MKL left out for KACZ.

- For GS RACE has an upper hand on Ivy-Bridge and on Sky-Lake RACE and ABMC have almost similar performance on Sky-Lake , although for some cases RACE has huge advantage. Reason for this advantage is due to slight decrease in iterations for RACE (put fig) and slight improvement in performance compared to ABMC for distance-1 case. For offshore case RACE performs worser that ABMC, this is because here with RACE one requires more iterations. Also note that all the large matrices which we had are unsuitable for GS sweep as they do not converge, but for large matrices the performance drops again for ABMC method due to degrading of α factor. (Maybe just put perf. pictures).

- Main advantage of RACE method comes with kernels having distance-2 dependencies like SymmSpMV and KACZ since here methods like ABMC require more colors and their locality degrades further since here within a color rows have to be structurally orthogonal (rows shouldn't have common column entries). Performance on KACZ shows this advantage. Here we again see for moderately large matrices the advantage is higher. Iteration behavior between methods remains similar to GS.

557   **8. Application Runs.**

558   **9. Conclusion.**

559   **10. Future Work.**

560   **Acknowledgments.** We would like to acknowledge the assistance of volunteers
561 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. Dongarra and M. Heroux, *Toward a new metric for ranking high performance computing systems*, Tech. Report SAND2013-4744, Sandia National Laboratories, 2013.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] H. Lu, M. Halappanavar, D. Chavarrⅰ-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.

[15] 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.

[16] 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.

[17] 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.

[18] E. Polizzi, *A density matrix-based algorithm for solving eigenvalue problems*, CoRR, abs/0901.2665 (2009), https://arxiv.org/abs/0901.2665.

[19] T. Röhl, J. Eitzinger, G. Hager, and G. Wellein, *Validation of hardware events for successful performance pattern identification in high performance computing*, in Tools for High Performance Computing 2015, Cham, 2016, Springer International Publishing, pp. 17–28.

[20] 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.

[21] J. TREIBIG, G. HAGER, AND G. WELLEIN, *Likwid: A lightweight performance-oriented tool suite for x86 multicore environments*, (2010).

[22] 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
___

1: Choose starting node(s) = $\{n\}$
2: $marked\_all$ = false
3: $N = nrows(graph)$
4: $distFromRoot[1..N] = -1$
5: $curr\_children.push\_back(n)$;
6: $currLvl = 0$
7: **while** $!marked\_all$ **do**
8:   $marked\_all$ = true
9:   $nxt\_children = \{\}$
10:   **for** $i = 1 : size(curr\_children)$ **do**
11:     **if** $distFromRoot[curr\_children[i]] == -1$ **then**
12:       $distFromRoot[curr\_children[i]] = currLvl$
13:       **for** j in $graph[curr\_children[i]].children$ **do**
14:         **if** $distFromRoot[j] == -1$ **then**
15:           $nxt\_children.push\_back(j)$
16:         **end if**
17:       **end for**
18:     **end if**
19:   **end for**
20:   $curr\_children = nxt\_children$
21:   $currLvl = currLvl + 1$
22: **end while**

632

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:    $T\_size = \text{update}(T\_ptr)$ %T_size contains non-zeros in each *level group*
6:    $mean\_r = \text{sum}(T\_size[0 : num\_sweep : len]) \:/\: nthreads$
7:    $mean\_b = \text{sum}(T\_size[1 : num\_sweep : len]) \:/\: nthreads$
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:    $var = \text{dot\_product}(diff, diff)$
11:    $absRankIdx = \text{sortIdx}(\text{abs}(diff))$ % sortIdx returns permutation after
12:                         % sorting from bigger to larger
13:    $rankIdx = \text{sortIdx}(diff)$
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:        $\text{shift}(T\_ptr, acquireIdx, currRank)$ % 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:        $\text{shift}(T\_ptr, currRank, giveIdx)$
33:      **end if**
34:      **if** $!fail$ **then**
35:        $newVar = \text{calculate\_variance}(T\_ptr)$ % 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**