Cache blocking of distributed-memory parallel matrix power kernels

Lacey, Dane; Alappat, Christie; Lange, Florian; Hager, Georg; Fehske, Holger; Wellein, Gerhard

DOI
10.1177/10943420251319332

Publication date
2025

Document Version
Final published version

Published in
International Journal of High Performance Computing Applications

Citation (APA)
Lacey, D., Alappat, C., Lange, F., Hager, G., Fehske, H., & Wellein, G. (2025). Cache blocking of distributed-memory parallel matrix power kernels. *International Journal of High Performance Computing Applications*, *39*(3), 385-404. https://doi.org/10.1177/10943420251319332

Research Paper

International Journal of
HIGH PERFORMANCE
COMPUTING APPLICATIONS

The International Journal of High
Performance Computing Applications
2025, Vol. 0(0) 1–20

DOI: 10.1177/10943420251319332
journals.sagepub.com/home/hpc
Sage

Cache blocking of distributed-memory parallel matrix power kernels

Dane Lacey[1], Christie Alappat[1], Florian Lange[1],
Georg Hager[1], Holger Fehske[1,2] and Gerhard Wellein[1,3,4]

Abstract

Sparse matrix-vector products (SpMVs) are a bottleneck in many scientific codes. Due to the heavy strain on the main memory interface from loading the sparse matrix and the possibly irregular memory access pattern, SpMV typically exhibits low arithmetic intensity. Repeating these products multiple times with the same matrix is required in many algorithms. This so-called matrix power kernel (MPK) provides an opportunity for data reuse since the same matrix data is loaded from main memory multiple times, an opportunity that has only recently been exploited successfully with the Recursive Algebraic Coloring Engine (RACE). Using RACE, one considers a graph based formulation of the SpMV and employs a level-based implementation of SpMV for the reuse of relevant matrix data. However, the underlying data dependencies have restricted the use of this concept to shared memory parallelization and thus to single compute nodes. Enabling cache blocking for distributed-memory parallelization of MPK is challenging due to the need for explicit communication and synchronization of data in neighboring levels. In this work, we propose and implement a flexible method that interleaves the cache-blocking capabilities of RACE with an MPI communication scheme that fulfills all data dependencies among processes. Compared to a "traditional" distributed-memory parallel MPK, our new distributed level-blocked MPK yields substantial speed-ups on modern Intel and AMD architectures across a wide range of sparse matrices from various scientific applications. Finally, we address a modern quantum physics problem to demonstrate the applicability of our method, achieving a speed-up of up to 4× on 832 cores of an Intel Sapphire Rapids cluster.

Keywords

Distributed-memory algorithms, sparse matrices, cache blocking, performance

1. Introduction and related work

Parallel solvers for linear systems or eigenvalue problems involving large sparse matrices have been widely used for decades in traditional research fields using high-performance computing (HPC) such as quantum physics, quantum chemistry, and engineering. In recent years, new applications relying on powerful and efficient sparse matrix solvers have been developed, ranging from social graph analysis as shown by Simpson et al. (2018) to spectral clustering in the context of learning algorithms, as shown by Luxburg (2004); McQueen et al. (2016). Typically these solvers use iterative subspace methods, which may include advanced preconditioning techniques and rely on an efficient parallel implementation of the sparse-matrix vector (SpMV) kernel $y \leftarrow Ax$, where A is a sparse matrix and x, y are dense vectors. Scalable and efficient SpMV implementations have thus been an active field of investigation for a long time, as shown by Vuduc and Demmel (2003) and more recently by Gao et al. (2024). Its low computational intensity makes SpMV strongly memory bound on all modern compute devices, and much research focuses on efficient sparse matrix data layouts or matrix bandwidth reduction to improve access locality in the dense vectors involved in the SpMV. Kreutzer et al. (2014) showed that

[1]Erlangen National High Performance Computing Center (NHR@FAU), Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany
[2]Institute of Physics, University of Greifswald, Greifswald, Germany
[3]Department of Computer Science, Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany
[4]Delft Institute of Applied Mathematics, Delft University of Technology, Netherlands

Corresponding author:
Dane Lacey, Erlangen National High Performance Computing Center (NHR@FAU), Friedrich-Alexander-Universität Erlangen-Nürnberg, Martensstraße 1, Erlangen 91058, Germany.
Email: dane.c.lacey@fau.de

this was particularly relevant on GPGPUs and wide-SIMD many-core CPUs.

However, these efforts do not exploit the data reuse opportunities presented by successive SpMV invocations with the same matrix.

Certain algorithms can be reformulated to group SpMV invocations with the same matrix together, as shown by Demmel et al. (2008) for CA-Krylov and by Loe et al. (2020) for preconditioners based on matrix polynomials. These back-to-back SpMVs constitute what we call the traditional *Matrix Power Kernel* (MPK) implementation. This kernel computes all vectors $y_p \leftarrow A^p x$ for each power $p = 1, \ldots, p_m$, where the sparse (and necessarily square) matrix A is loaded from main memory each time SpMV is called. This scenario presents an immense opportunity for raising the computational intensity through *cache blocking*, keeping relevant matrix data in the cache across successive SpMV invocations.

In recent years, the top CPU and GPGPU manufacturers have been rapidly increasing the cache sizes on their server-grade chips. Shown in Table 1 is a selection of top-of-the-line CPU and GPGPU models from Intel, AMD, and Nvidia, and their respective aggregate cache sizes (sum of all cache levels, rounded to the nearest MiB[1]) over the last several years. These advancements in hardware capabilities have only broadened the opportunities for cache blocking.

The Recursive Algebraic Coloring Engine (RACE), as introduced by Alappat et al. (2020a), can be used to construct an efficient, cache-blocked shared-memory MPK by taking advantage of the level-based formulation of SpMV. Alappat et al. (2022) describe the resulting *Level-Blocked Matrix Power Kernel* (LB-MPK), with applications of LB-MPK to contemporary sparse iterative solvers shown by Alappat et al. (2023). While successful, this work is restricted to shared-memory compute nodes. No concept or implementation to parallelize RACE for distributed-memory parallel systems using the Message Passing Interface (MPI) has been proposed until now. Satisfying the data dependencies of the level-based formulation among parallel processes by message passing is a non-trivial task.

The main contribution of this work is an MPI adaptation of LB-MPK. Other works on distributed-memory parallel MPK, such as those developed by Yamazaki et al. (2014a,b), are focused on reducing the MPI communication overhead. At the time of writing, there is surprisingly little work found in the direction of cache-blocking techniques for the distributed-memory parallel MPK. There exists an analysis of a similar diamond tiling strategy by Vatai et al. (2020), but it is purely theoretical. The closest work is likely from Mohiyuddin et al. (2009), but there are clear differences between this approach and ours. Besides being MPI-only whereas ours is a hybrid (MPI + OpenMP) approach, their MPK requires redundant computations and/or indirect accesses to matrix elements with bookkeeping to fulfill data dependencies. We will revisit this comparison in Section 5.

When compared to the traditional “back-to-back” SpMV implementation of MPK, our novel distributed level-blocked MPK algorithm shows speed-ups of up to 2.7× across various architectures for a wide variety of matrices from the SuiteSparse Matrix Collection by Davis and Hu (2011).

Note that there will only be notable benefits of cache blocking if the working set does not fit into the aggregate caches of all CPUs involved. With in-cache datasets, communication and synchronization overheads would likely dominate the runtime.

2. Overview and contributions

The *Distributed Level-Blocked Matrix Power Kernel* (DLB-MPK) algorithm extends the LB-MPK algorithm to the distributed-memory setting with MPI. Our implementation is efficient in that it does not increase the MPI overhead when compared to the traditional MPK implementation, and it does not require any redundant computations.

Table 1. Cache size trends for Intel, AMD, and Nvidia devices.[2,3]

Company	Year	Model	Type	Aggregate Cache
Intel	2019 Q1	Cascade Lake - 8280	CPU	68 MiB
	2021 Q4	Ice Lake - 8380	CPU	102 MiB
	2023 Q1	Sapphire Rapids - 8480	CPU	221 MiB
	2023 Q1	Ponte Vecchio - MAX 1550	GPGPU	472 MiB
AMD	2019 Q2	Zen 2 - 7742	CPU	294 MiB
	2022 Q1	Zen 3 - 7773X	CPU	804 MiB
	2023 Q2	Zen 4 - 9684X	CPU	1254 MiB
	2023 Q4	Aqua Vanjaram - MI300X	GPGPU	277 MiB
Nvidia	2018 Q1	Volta - V100 SXM3	GPGPU	16 MiB
	2020 Q1	Ampere - A100 SXM4	GPGPU	60 MiB
	2023 Q1	Hopper - H100 SXM5	GPGPU	83 MiB
	2023 Q2	Grace Superchip	CPU	333 MiB

This paper is organized as follows. In Section 3, we begin with a brief summary of shared-memory SpMV and MPK. Then, by exploring the graph-matrix correspondence, we are able to understand the broad strokes of how RACE performs LB-MPK. In order to generalize LB-MPK, we must first understand distributed-memory parallel SpMV and MPK without cache blocking, which we explore in Section 4. We close this section with a motivation of our method by comparing it against the distributed-memory parallel MPK implemented by Mohiyuddin et al. (2009). Section 5 details our DLB-MPK method and implementation. In Section 6 we investigate the relevant hardware characteristics of three modern multicore CPU systems and their influence on the performance of DLB-MPK. Performance predictions based on the roofline model by Williams et al. (2009) are derived, and we investigate the influence of various parameters with RACE in the distributed-memory setting. We close the section with a strong scaling analysis of DLB-MPK. In Section 7 we examine the weak scaling characteristics of DLB-MPK used in Chebyshev time propagation, which has applications in quantum physics.

In this work, we make the following contributions:

- We extend the level-based concepts in RACE to the distributed-memory setting.
- We detail the trapezoidal-like tiling strategy which enables our DLB-MPK to fulfill the data dependencies inherent in repeated SpMV invocations, and present an efficient implementation of the DLB-MPK.
- For a wide array of sparse matrices, we present a performance and scaling benefit summary of DLB-MPK on three modern CPUs from Intel and AMD.
- We investigate the weak scaling behavior of DLB-MPK when applied to the Chebyshev method for the time evolution of quantum states for the Anderson model of localization, and show the favorable scaling qualities as compared to the traditional MPK.

3. RACE applied to the matrix power kernel

For a given square sparse matrix A and dense vector x, MPK computes all vectors $y_p \leftarrow A^p x$ for each power $p = 1, \ldots, p_m$, and stores the result into p_m dense vectors. As mentioned before, this is traditionally implemented as a series of back-to-back SpMVs, using the output vector from the previous iteration as the input vector x; for example, at the kth SpMV invocation, $y_k \leftarrow Ax$ where $x = y_{k-1}$. SpMV is the central kernel of MPK, whose traditional implementation will be limited by the same bottleneck as SpMV. For matrices A that do not fit into the cache on modern CPUs (so-called "memory-resident" matrices), the limiting performance bottleneck is the main memory load bandwidth.

The key observation when cache blocking the MPK is that we can compute $A^p x$ on a subset of rows without waiting for the entire $A^{p-1}x$ computation to finish first for all rows of A. The only data dependency for "promoting" a row v from $A^{p-1}x$ to $A^p x$, that is, executing the pth SpMV operation on it, is that the rows that correspond to the column indices of the non-zero elements in row v have already been promoted to $A^{p-1}x$. When using LB-MPK, cache blocking is achieved by detecting the dependencies between successive SpMV invocations using the level-based SpMV formation within RACE. The degree to which this fact can be exploited strongly depends on the sparsity pattern of the matrix. To understand the cache-blocking scheme in the shared memory setting, we describe this level-based formulation here.

Given a matrix A, there exists a correspondence with a graph $G\,(V, E)$. The set of vertices V represents the rows of A, and the set of edges E represents the non-zero elements. If row v in A has a non-zero element at column j, then there exists a corresponding edge from vertex j to vertex v in $G\,(V, E)$. In order to make the correspondence more immediate, we use $G(A)$ to denote the graph which has A as its adjacency matrix. For this work, the values of the non-zero entries of the corresponding matrix A are not considered in the graph. An example of such a correspondence is given by the sparse matrix representing a modified 5pt stencil in Figure 1(b) and the associated graph in Figure 1(a). If vertex v is in the set of "neighbors" of u,

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

then v is said to be "distance 1" from u. We say that a vertex q is "distance k" from a vertex u when q is in the "kth neighborhood" of u, where we recursively define

$$N^k(u) = N^{k-1}(N(u)).$$

RACE will start a Breadth-First Search (BFS) at some "root vertex," typically at row index 0. In the next step, all vertices that have an edge connected to this root vertex (i.e., its neighbors) are collected into a structure that we call a "level." In general, for a graph $G\,(V, E)$, we can define the ith level as:

$$L(i) = \begin{cases} \text{root vertex if } i = 0, \\ u \in N(L(i-1)) \text{ if } i = 1, \\ u \in N(L(i-1)) \cap \overline{N(L(i-2))} \cap \overline{L(i-2)} \text{ else.} \end{cases}$$

At each successive step in the search, all vertices in the current level are scanned, and all neighbors of these vertices that have not yet been touched are collected into the next level. The process continues until the graph is fully traversed,

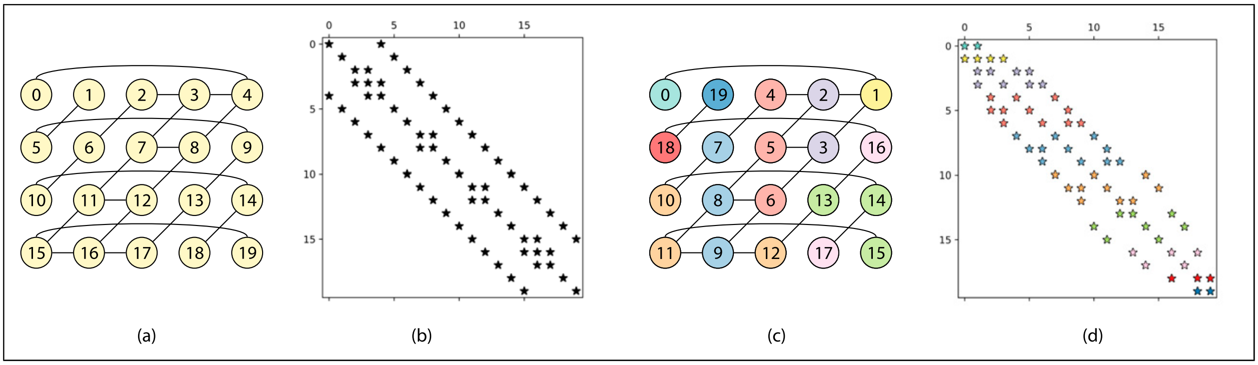


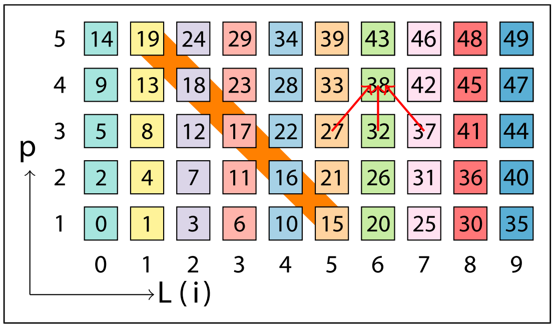
Figure 1. Graph (a) and sparsity pattern (b) of the matrix associated with a modified five-point stencil. Graph (c) shows the permuted graph and (d) the sparsity pattern of the matrix after applying Breadth First Search (BFS) reordering. The vertices (rows) of the graph (matrix) that belong to a level are represented with the same color.

at which point every vertex is collected into a mutually exclusive level.[4] Once the graph is fully traversed and each vertex is assigned to a level, RACE then symmetrically permutes the matrix A (rows and columns) based on the levels collected. The symmetric permutation, referred to as "BFS reordering," improves the temporal locality on the RHS x-vector and avoids irregular accesses to matrix elements.[5] An example of this reordering is given for our 5pt stencil matrix in Figure 1(d) and the associated graph in Figure 1(c).

We can visualize the dependencies and traversal order of LB-MPK with an "*Lp* diagram" given in Figure 2. The x-axis is the index of the level L, and the y-axis is the power p in $A^p x$. An important property of levels is that neighbors of $L(i)$ are contained in $\{L(i-1), L(i), L(i+1)\}$. This means in order to compute Ax on L(i), x has to be known only on $\{L(i-1), L(i), L(i+1)\}$. More generally, to compute $A^p x = AA^{p-1}x$ on the vertices of $L(i)$, $A^{p-1}x$ computations on the vertices of $L(i-1)$, $L(i)$, and $L(i+1)$ must have already been completed. One particular example is featured in Figure 2 for the computation of $A^4 x$ at $L(6)$, where the dependencies lie on $p = 3$ at $\{L(5), L(6), L(7)\}$.

One way to ensure these dependencies are fulfilled at any point in time is to traverse the *Lp* diagram such that each diagonal defined by $i + p :=$ const carries out computations in a "bottom-right to top-left" fashion for increasing values of "const" (i.e., $i + p = 1$, $i + p = 2$, …). This execution order is given by the numbered boxes in Figure 2, and emphasized by the highlighted diagonal for $i + p = 6$.

With the aid of the *Lp* diagram, the idea behind level-based cache blocking can now be briefly introduced. As LB-MPK diagonally traverses the levels as described above, levels (and therefore matrix entries) are reused after $p_m + 1$ execution steps (after the wind-up phase on the left end, and before the wind-down phase at the right end of the *Lp* diagram). If all the non-zero matrix entries associated with these $p_m + 1$ levels accessed between two computations of the same $L(i)$ can be held in cache, then all matrix data for the following computation with $L(i)$ will be accessed from cache (with the exception of $p = 1$, which has a compulsory cache miss and must come from main memory). As an explicit example, see the level $L(5)$ which is used in the 15th step in the execution of LB-MPK. If all the matrix data corresponding to the six levels $L(1)$–$L(6)$ can be held in cache, then the vertices of $L(5)$ are reused in the 21st step in the execution of LB-MPK when computing $p = 2$.



p										
5	14	19	24	29	34	39	43	46	48	49
4	9	13	18	23	28	33	38	42	45	47
3	5	8	12	17	22	27	32	37	41	44
2	2	4	7	11	16	21	26	31	36	40
1	0	1	3	6	10	15	20	25	30	35
L (i)	0	1	2	3	4	5	6	7	8	9

Figure 2. *Lp* diagram with 10 levels ($L(0)$, …, $L(9)$) and a maximum power of $p_m = 5$. Level colors are the same as in Figure 1(c). Each node in the *Lp* diagram is numbered according to the execution order. For $p = 4$ and level $L(6)$, the explicit dependencies to levels at $p = 3$ are indicated with red arrows. The nodes highlighted in orange fulfill $i + p = 6$ ("diagonal").

4. Challenges in the distributed-memory setting

Distributing MPK for level-based cache blocking across multiple MPI processes is not as easy as just executing LB-MPK locally on each MPI process. To understand this non-triviality, we first investigate the dependencies that arise from the "traditional" distributed-memory parallel MPK (TRAD). Just as in the shared memory setting, a distributed-memory parallel MPK is traditionally constructed from back-to-back SpMV invocations.

In the distributed-memory setting, the matrix is partitioned among the available MPI processes. The conventional approach, which we use, often employs row-based partitioning wherein both matrix and vector entries corresponding to a subset of rows are physically assigned to individual MPI processes. Figure 3(a) illustrates distributing the matrix A from Figure 1(b) across two MPI processes. The dotted lines represent the MPI "boundary," that is, where the data is physically disjointed. The corresponding graph of the matrix ($G(A)$) in the distributed-memory setting is shown in Figure 3(b). The crux of the problem lies in the distributed nature of the x-vector. During SpMV computations on a given MPI process, there may be non-zero matrix elements that do not have their corresponding RHS x-vector elements for the dot product locally on the process, necessitating their retrieval from remote MPI processes. For instance, in Figure 3(a), row 8 belonging to the first MPI process contains a non-zero element at column index 12. While the non-zero elements with column indices 4, 7, and 8 in row 8 can be multiplied with the local x-vector for the corresponding dot product, the non-zero element at column index 12 lacks the requisite data on this MPI process's x-vector, which then must be fetched from the second MPI process. Similarly, on the second MPI process, row 12 necessitates x-vector data corresponding to row index 8, which resides on the first MPI process.

Transferring remote elements on-demand is feasible but would result in significant performance overhead due to the high latency of MPI communications. Consequently, a common strategy involves bulk transfer of all required remote elements before executing SpMV operations. These elements are then stored consecutively, typically at the end of the x-vector, forming what is commonly known as the "halo region/buffer." Figure 3(c) illustrates this halo region and the process of populating it with remote elements. Algorithm 1 presents pseudocode for the traditional distributed-memory parallel MPK computing $A^{p_m}x$. Here, we assume a matrix A has already been partitioned row-wise and distributed to each of the n processes so that A_i resides on process i. The algorithm utilizes two subroutines: the `haloComm` routine that populates the halo region, and the sparse matrix-vector product `SpMV`. The local vector size is the local number of rows $N_{r,i}$, plus the number of remote elements the process i needs to receive into its halo buffer $N_{h,i}$. The "MPI overhead" O_{MPI} is understood to be the ratio of these halo rows on each x-vector across all the n MPI processes to the total number of rows N_r $(=\sum_{i=0}^{n} N_{i,r})$, that is

$$O_{\text{MPI}} := \frac{\sum_{i=0}^{n} N_{h,i}}{N_r}. \tag{1}$$

Figure 4(a) illustrates the distributed TRAD MPK approach and shows the required halo communication. The number on each vertex represents the execution order for computing SpMV on the particular vertex x_i. The TRAD approach necessitates a complete SpMV operation to be carried out before initiating the subsequent halo communication routine. This poses a challenge to cache blocking, particularly when dealing with large in-memory matrices, as the cache may not be able to accommodate all matrix elements loaded during the entire SpMV computation. In Section 3, we have seen that caching can be realized by the LB-MPK approach on shared memory. This necessitates that all the p_m SpMV computations required to raise the local matrix A_i to power p_m be carried out consecutively in one kernel. This requirement renders the basic halo

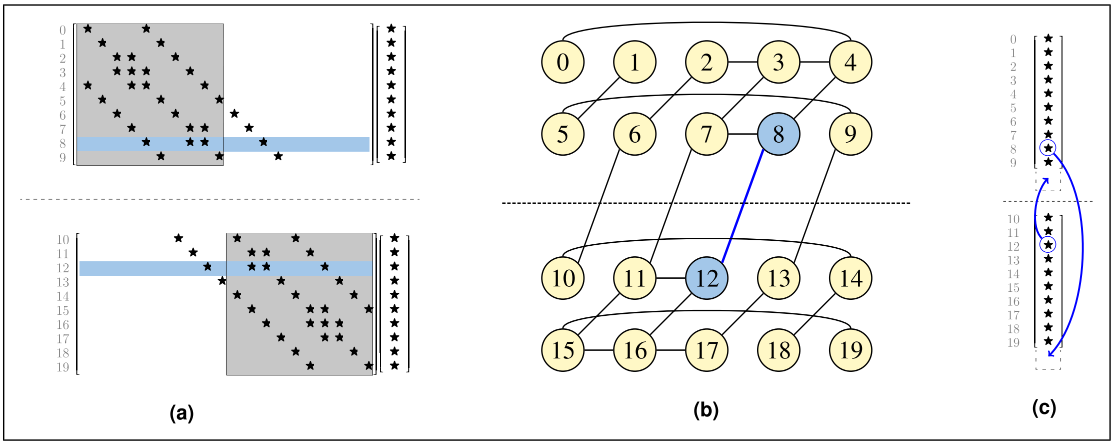


Figure 3. The global matrix A from Figure 1(b) and some RHS vector x are partitioned in a row-wise manner over two MPI processes in (a). The gray boxed-out regions show, on each MPI process, which elements are "local" (inside the gray region). The edge corresponding to the remote data dependency, that is, the edge crossing the MPI boundary, is highlighted in blue in (b). The rows at global indices 8 and 12 are highlighted as examples of rows that contain remote data dependencies for the SpMV. To fulfill these data dependencies, another MPI process must supply the appropriate "halo elements." Shown in (c) is the process of data exchange on the x-vector for our two example rows, where incoming halo elements are received into an appropriately resized buffer.

communication scheme explained above inadequate for distributed-memory parallelization since only the halos necessary for a single SpMV are communicated in this step. However, for each halo element, we now require the values of all $p_m - 1$ powers, that is, $A^p x$ for all p in the range $[0, p_m - 1]$. Complicating matters further, these values (for $p \geq 1$) are not yet available at this stage because SpMV computations have not been performed.

Algorithm 1 Traditional Distributed MPK

```
Input:
  double $x[N_{r,i} + N_{h,i}]$;
  sparseMatrix $A_i$
  int $p_m$
Output:
  double $y[N_{r,i} + N_{h,i}, p_m]$

  $y[:, 0] \leftarrow x$;
  for $p \leftarrow 1, \ldots, p_m$ do
      $y[:, p-1] \leftarrow$ haloComm($y[:, p-1]$);
      $y[:, p] \leftarrow$ SpMV($y[:, p-1], A_i[:, :]$);
  end for
```

For example, consider Figure 3: employing a `haloComm` routine ensures that the first MPI process gains access to x values corresponding to all rows that require remote elements (in this case, rows 6, 7, 8 and 9). However, when conducting LB-MPK with $p_m = 2$, the first MPI process requires Ax values (as opposed to x) at, for example, the 12th row when computing A^2x for row 8, but the Ax value at the 12th row has yet to be computed by the other process.

One potential solution to this problem, as explained by Mohiyuddin et al. (2009), is known as communication-avoiding MPK (CA-MPK). In this approach, all the necessary values of halo elements, $A^p x$ for all p in the range $[1, p_m - 1]$, are computed locally on each MPI process. To achieve this, each MPI process conducts additional SpMVs on the halo elements. However, as discussed in Section 3, computing $A^p x$ by an SpMV operation necessitates updating its neighbors to the $A^{p-1}x$ value, which in turn requires updating its neighbors to $A^{p-2}x$, and so on until it reaches the input vector $A^0 x = x$. Consequently, to raise boundary halos B to the $p_m - 1$ power, all its distance-$(p_m - 1)$ neighbors must also be updated. Given that these neighbors often reside on different MPI processes, remote elements must be brought into the current MPI process, thereby requiring additional halo elements. Figure 4(b) illustrates the additional halos required by the CA-MPK approach on a 1D tri-diagonal stencil example. Additional SpMVs take place within the halo buffer, that is, vertices that are "external" to the process-local data. In our example, these redundant SpMVs occur at execution stages 12, 15, and 16. To compute $A^p x$, CA-MPK requires $p - 1$ groups of these external vertices. In general, the halos are organized based on their distance from the boundary B, where E_k represents the set of external vertices that are at a distance of k from B. The boundary halo elements $B = E_0$ are elevated to power $p_m - 1$, while the remaining halo elements E_k are elevated to power $p_m - 1 - k$ to fulfill the dependencies.

To facilitate cache blocking, a diagonal-style execution order, similar to that in LB-MPK (see Section 3), can be employed. The name "communication-avoiding" stems from the ability of the CA-MPK approach to overlap communication and computations. In Figure 4(b) the purely local part (outlined in blue boundary) can be overlapped with the communication of the remote elements. Although the CA-MPK approach enables cache blocking, the overheads resulting from additional halo communication and SpMV computations escalate with the power p_m and the number of MPI processes n. It is important to note that these extra SpMV computations on halos are redundant, as the MPI process possessing the element locally also conducts SpMVs on these elements.

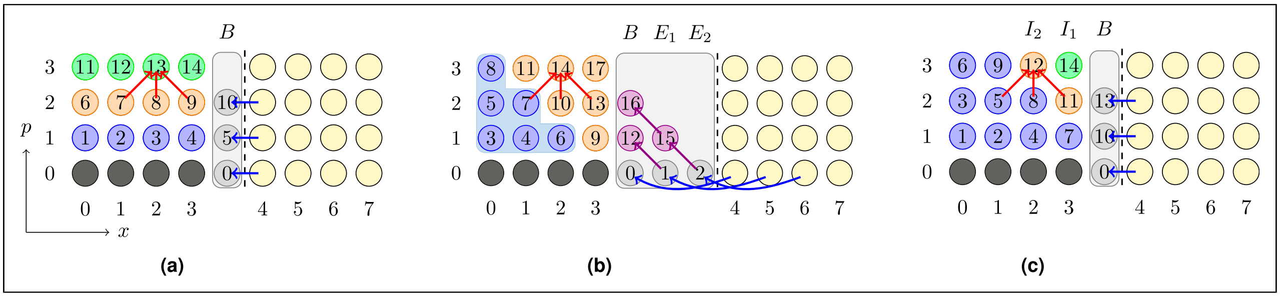


Figure 4. Comparison of three MPK implementations for the computation of A^3x on a 1D tri-diagonal stencil matrix, distributed across two MPI processes, where the execution order is written in each node. The traditional MPK implementation of back-to-back SpMVs is shown in (a) the "Communication Avoiding" MPK with redundant SpMVs in (b) and our implementation of DLB-MPK is shown in (c). Each dot represents an index of $A^p x$ for the respective power p. In each of the three diagrams, the dashed vertical line denotes the MPI boundary, the process-local x-vector data is shown in black on the bottom layer and is assumed to already be present, and the halo buffer is shown in gray. The x-axis represents the index of the RHS x-vector. The red arrows indicate that the data dependencies are the same in each MPK version, regardless of the execution order.

Particularly with irregular sparse matrices, these overheads can be substantial and may lead to limited speedups as shown in Yamazaki et al. (2014a).

One way to eliminate redundant computations involves a fine-grained synchronization mechanism, wherein the other process transmits the Ax value of halo elements once computations are completed, and the other process waits to receive this data. However, this entails significant synchronization overhead and the transmission of small MPI messages, ultimately resulting in substantial performance degradation due to the high latency of MPI communications. In the following section, we will introduce a savvy new approach to mitigate these performance pitfalls.

5. DLB-MPK methodology

The DLB-MPK approach enables cache blocking while mitigating the drawbacks associated with CA-MPK, namely the need for additional communication and computations. DLB-MPK achieves this by utilizing the same halo communication routine as in the traditional approach (TRAD), but with a reordering of computation and communication to facilitate cache blocking.

In our algorithm, following the initial halo communication, LB-MPK is executed on the local vertices. However, not all local vertices can be elevated to power p_m immediately due to dependencies with the halo elements in B, which contain only the input value x. Internal vertices that are distance-1 neighbors to B can only be promoted to Ax ($p = 1$), while their neighbors can only be promoted up to A^2x, and so forth. In general, internal vertices at a distance of k from the boundary B, denoted as I_k, can only be elevated up to A^kx. This implies that, at this stage of DLB-MPK, computations are incomplete on internal vertices I_k where $1 \leq k < p_m$. The final step of the DLB-MPK method is an iterative process ensuring the completion of SpMV computations on the incomplete internal vertices. The iterative post-cache-blocking computation phase begins with synchronization followed by a call to the halo communication routine to update halo boundaries B with the next power value (Ax in the first iteration). This enables all incomplete internal vertices I_k to perform SpMVs, advancing their power computations by one step. This remainder phase is repeated for a total of $p_m - 1$ times to ensure all internal vertices reach power p_m. Figure 4(c) illustrates the DLB-MPK approach using a 1D tri-diagonal stencil example.

As shown in Figure 4, DLB-MPK requires the same halos as TRAD while benefiting from cache-blocking advantages similar to CA-MPK due to its diagonal-style execution; refer to Section 3 for details. Figure 5 quantifies the advantages of reduced halo elements and zero redundant computations for DLB-MPK for an irregular sparse matrix (`Serena`), which is partitioned row-wise over 10 and 15 MPI processes, respectively. In order to minimize communication and optimize load balance, METIS by Karypis and Kumar (1998) was chosen as the global partitioner. The left subfigure in Figure 5 shows the relative number of halo elements incurred by CA-MPK in addition to what is caused by DLB-MPK accumulated over all MPI processes, while the right subfigure shows the relative number of required redundant computations for CA-MPK subject to the same global partitioning. Despite the banded sparsity pattern of the matrix, the halo elements required for CA-MPK grow significantly with the power p and the number of MPI processes.

The implementation of DLB-MPK can be straightforwardly derived from the execution order illustrated in Figure 4(c) for a 1D tri-diagonal example. However, when

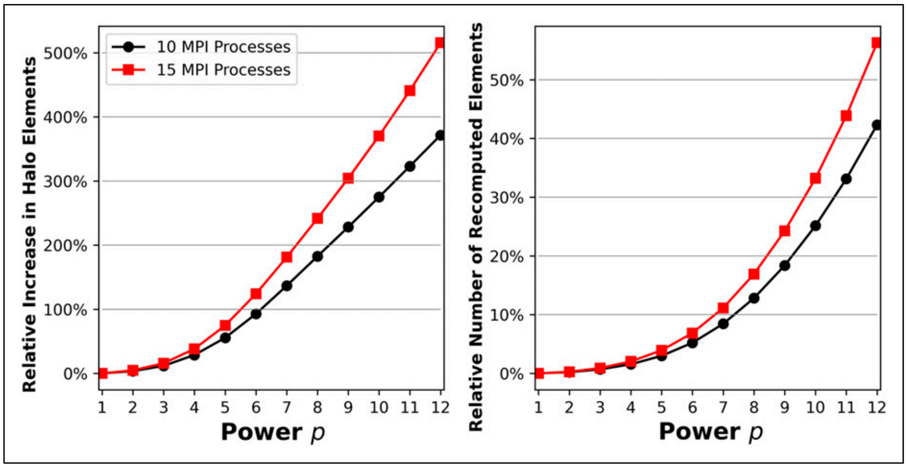


Figure 5. Overheads in CA-MPK associated with the `Serena` matrix, partitioned over 10 and 15 MPI processes for powers $p \in \{1, 2, \ldots, 12\}$. *Left*: additional halo elements relative to the total number of rows N_r. *Right*: recomputed elements relative to the total number of non-zero elements N_{nz}.

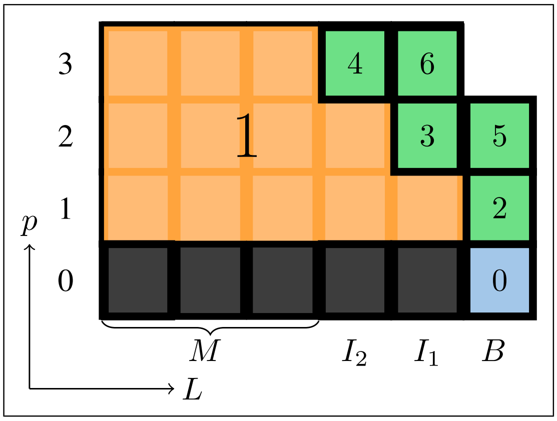


Figure 6. An adapted *Lp* diagram for DLB-MPK executing A_i^3x on some MPI process *i*. The numbers indicate the order of execution of DLB-MPK. The colors of the boxes indicate the phase of DLB-MPK in which they are executed. The blue box corresponds to the first phase, the orange to the LB-MPK phase, and the green to the iterative third phase.

dealing with a general sparse matrix, the internal boundary vertices I_k for $k < p_m$ may not be ordered consecutively. Therefore, an efficient implementation will require gathering these boundary vertices and reordering the matrix during preprocessing to ensure that these vertices (rows in the matrix) appear consecutively. All vertices which are a distance of p_m or greater from the boundary, i.e. all vertices in I_k for $k \geq p_m$, are collected into a single main "bulk structure" M, which is large in practice.

The algorithm is separated into three distinct phases: (i) execute the initial halo communication, (ii) use the cache blocking capabilities of RACE to fully promote all levels in M to p_m, and each I_k level to k, and (iii) iteratively finish remaining computations and communications. We use a modified *Lp* diagram in Figure 6 as an example of DLB-MPK executing A_i^3x on some MPI process i. The color of the box indicates in which phase it is executed. The blue box corresponds to the first phase, the orange to the LB-MPK phase, and the green to the iterative third phase. As previously mentioned, $B = I_0$ is the halo buffer, while I_1 and I_2 are all vertices that are distances 1 and 2, respectively, away from the MPI boundary. Instead of labeling individual levels, Figure 6 represents the main bulk structure by M. It is here that RACE can safely perform cache blocking.

A benefit of DLB-MPK is that we can use the same MPI routines as for TRAD. Hence, can easily integrate our algorithm into external libraries with existing SpMV and halo communication routines. This is shown in Algorithm 2, which gives a high-level overview of DLB-MPK. The callback functions `haloComm` and `SpMV` are both provided by the user. The structure I contains the first $p_m - 1$ levels of A_i, where again $I[0] = B$ contains the boundary vertices. The initial halo exchange takes place in the first phase, highlighted in blue. The cache-blocking second phase is executed during `localLBMPK`, highlighted in orange. Finally, the iterative third phase, represented by the nested for-loops, is highlighted in green.

The percentage of vertices that fall outside of the bulk structure is considered as the "local overhead" $O_{\text{DLB-MPK},i}$ of DLB-MPK. While not an "overhead" per se, it is a useful quantity for our investigation as it expresses the efficiency of cache blocking. With M_i denoting the bulk structure level on MPI process i, we can define this overhead as

$$O_{\text{DLB}-\text{MPK},i} := 1 - \frac{|M_i|}{N_{i,r}}. \tag{2}$$

To have a single number which represents the "global overhead" from cache blocking, we collect the local overheads in Equation (2) from each of the n processes and normalize them over the total number of rows, yielding

$$O_{\text{DLB}-\text{MPK}} := \frac{\sum_{i=0}^{n} (N_{i,r} \cdot O_{\text{DLB}-\text{MPK},i})}{N_r}. \tag{3}$$

6. Results

In this section, we investigate the performance and scaling characteristics of DLB-MPK and how they compare to TRAD in a variety of scenarios on a selection of modern multicore CPUs. To gain a deeper understanding of the performance of our level-based cache-blocked MPK, we first establish a theoretical roofline-based upper performance prediction for the SpMV kernel, the main kernel used in MPK.

It is well known that SpMV (and by extension traditional MPK) is usually a memory-bound kernel on modern hardware for sparse matrices from science and engineering, as described by Kreutzer et al. (2014). According to the roofline model, in the memory bound regime with the CRS matrix storage format[5] using 8 bytes for the matrix values, 4 bytes for the column indices, and 4 bytes for the row pointer, performance is limited by

$$P = \frac{b_s}{6\ \text{B} + 14\ \text{B}/N_{nzr}}, \tag{4}$$

where b_s denotes the saturated main memory bandwidth, and $N_{nzr} = N_{nz}/N_r$ denotes the average number of non-zero elements per row.

6.1. Experimental setup

The relevant hardware and software environment used for the measurements is explained in the following.

6.1.1. Hardware. In this work, all experiments were conducted on dual-socket nodes of either Intel Ice Lake (ICL), Intel Sapphire Rapids (SPR), or AMD Epyc Zen3 (MIL).

Table 2. Single-socket hardware Configurations.

Architecture	ICL	SPR	MIL
Chip model	Xeon Platinum 8360Y	Xeon Platinum 8470	AMD EPYC 7763
Microarchitecture	Sunny Cove	Golden Cove	Zen 3
Cores	36	52	64
ccNUMA domains	2	4	4
Max. SIMD width	512 bits	512 bits	256 bits
L1D cache capacity	36 × 48 KiB	52 × 48 KiB	64 × 32 KiB
L2 cache capacity	36 × 1.25 MiB	52 × 2 MiB	64 × 512 KiB
L3 cache capacity	54 MiB	105 MiB	8 × 32 MiB
L3 load bandwidth	452 GB/s	826 GB/s	2642 GB/s
Mem. Configuration	8 ch. DDR4-3200	8 ch. DDR5-4400	8 ch. DDR4-3200
Mem. Load bandwidth	180 GB/s	241 GB/s	179 GB/s

Table 2 details the important aspects of each architecture. ICL and SPR are both capable of performing AVX-512 instructions, while MIL supports only up to AVX-2. Sub-NUMA Clustering with the maximum possible number of ccNUMA domains was enabled on the Intel systems (two on ICL and four on SPR), and NPS = 4 was set on MIL.

In order to reflect practical use case scenarios, turbo mode was enabled for all experiments.

All of these machines have three levels of cache: private, inclusive L1 and L2, and a victim-style L3 cache. This means that we can consider the sum of L2 and L3 cache as the total size for which we can use RACE to cache block. RACE excels when blocking for outer level caches, as shown by Alappat et al. (2022).

Since the achievable bandwidth of the hardware plays a vital role in determining the performance of SpMV-like kernels (see Equation (4)), we investigate the bandwidths on each of the machines in Figure 7. The load-only kernel from `likwid-bench` by Treibig et al. (2010) is used here to determine the bandwidth as it reflects the predominant behavior of the SpMV kernel. The most striking contrast between the three plots in Figure 7 is the difference in the scale of y-axes. ICL has about half the L3 bandwidth of SPR, and only about a sixth of the L3 bandwidth of MIL. In terms of main memory bandwidth, however, ICL narrowly surpasses MIL, while SPR beats both by at least 30%.

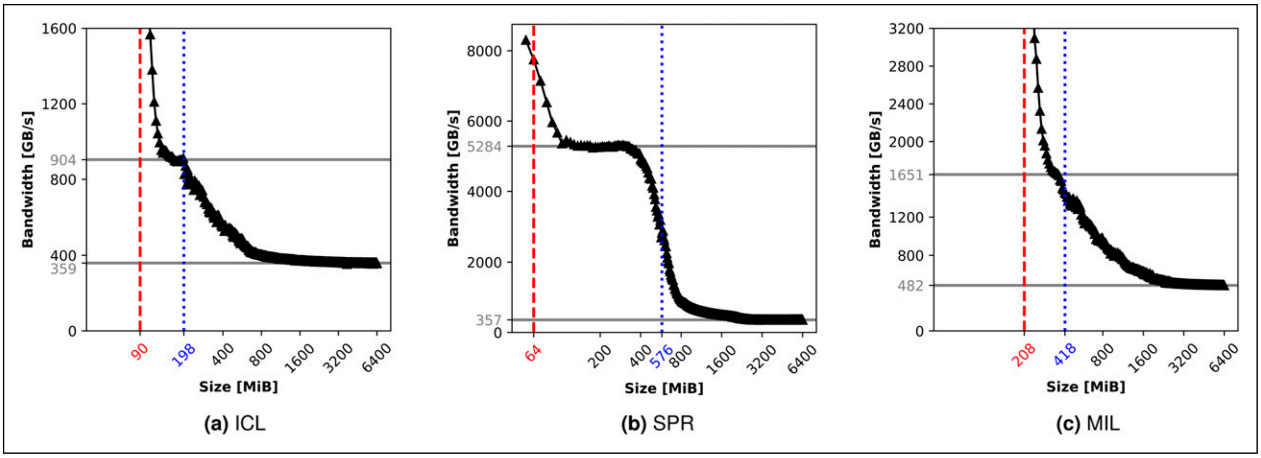
The cache “plateaus” at which we estimate the L3 bandwidths vary in behavior. Both ICL and SPR display a gradual degradation of bandwidth after the data set exceeds a cache size, which is due to Intel’s “dynamic replacement policy.” It has been shown by Alappat et al. (2020b) that this policy makes intelligent use of the cache for data sets that exceed the cache size. AMD’s cache replacement policy is different and leads to faster bandwidth degradation as can be seen in Figure 7(c). The wide plateau in Figure 7(c) can be explained by MIL having the largest ratio between L2 + L3 and L2 sizes out of the three architectures considered, which is due to its massive L3 cache. Figure 7 indicates that we should expect strong residual caching effects for matrices up to about 800 MiB on ICL, and up to about 2400 MiB for SPR and MIL.

6.1.2. Software. Table 4 lists the matrices used for benchmarking with their number of rows N_r, number of non-zero elements N_{nz}, average number of non-zero elements per row $N_{nzr} = N_{nz}/N_r$, and the size of the matrix data in CRS format. The total size of a matrix is $(4N_r + 12N_{nz})$ B. Here, matrix sizes are rounded to the nearest whole number in MiB.

Our selection of benchmark matrices is commonly used in the literature for performance investigations. They show the performance of DLB-MPK compared to TRAD across a wide variety of sparsity patterns while keeping data sets generally large enough to not be completely cache resident (thus, eliminating the need for cache blocking). Most matrices are freely available from the Suite Sparse matrix collection, with the exception of the `Lynx` matrices, which come from a finite-volume code for Cardiac Arrhythmia simulations over unstructured meshes as described by Langguth et al. (2015, 2019).

Measures had to be taken against the patch for the “Downfall” security bug as explained by Moghimi (2023), incurring a penalty for gather instructions on the architectures under consideration. The latest LLVM-based Intel compiler was required, with special compilation flags to avoid the expensive gather instructions. In Table 3, these flags are given under “Downfall fix.” To ensure vectorization of the SpMV kernel, `#pragma omp simd simdlen(VECLEN) reduction(+:sum)` is used on the innermost SpMV loop, where `VECLEN` is the maximum SIMD width on the respective hardware (see Table 2), and `sum` is our accumulator for the SpMV. On the Intel architectures, the flags `-xCORE-AVX512` and `-qopt-zmm-usage=high` shown in Table 3 were also required so that the compiler would generate instructions using the 512-bit wide `zmm` registers.

The same affinity is used for benchmarking on each architecture. Each MPI process is pinned to one ccNUMA



(a) ICL **(b)** SPR **(c)** MIL

Figure 7. Full-node measured load bandwidths in GB/s (*y*-axis) versus data set size for the CPUs under consideration. The higher solid horizontal line represents the estimated L3 cache bandwidths and the lower one represents the estimated bandwidth from main memory. The dashed red line marks the overall L2 cache size for the entire node, while the dotted blue line represents the aggregate L2 + L3 cache size for the entire node. The widest SIMD registers are used on each machine for the load instructions.

Table 3. Software Configurations and compiler flags.

Architecture	ICL	SPR	MIL
OS	AlmaLinux 8.8	AlmaLinux 8.8	RHEL 8.8
MPI library version	Intel MPI 2021.10	Intel MPI 2021.10	Intel MPI 2023.03
Compiler	icx 2023.2.0	icx 2023.2.0	icx 2023.0.3
Flags			
Opt. level	-Ofast	-Ofast	-Ofast
Arch	-xhost	-xhost	-Mar = core-avx2 -mtune = core-avx2
Downfall fix	-Xclang -target-feature -Xclang + prefer-no-gather	-Xclang -target-feature -Xclang + prefer-no-gather	-Xclang -target-feature -Xclang + prefer-no-gather
Force AVX512	-xCORE-AVX512 -Qopt-zmm-usage = high	-xCORE-AVX512 -Qopt-zmm-usage = high	
Misc	-std = c++14 -fopenmp	-std = c++14 -fopenmp	-std = c++14 -fopenmp

domain, process $i + 1$ is mapped physically as close as possible to process i, and OpenMP threads are also pinned compactly to the physical cores. Simultaneous Multithreading (SMT) was disabled across all the systems.

While not a primary focus of this work, RACE allows users to specify a maximum recursion stage s_m which enables the breaking down of "bulky" levels for increasing cache blocking efficiency. This maximum recursion stage is set to $s_m = 50$ for all matrices except `Lynx1151`, where it is set to $s_m = 80$.

We aim to understand the performance gained from cache blocking, not from improved data accesses on the RHS x-vector through the local symmetric BFS permutations (see Section 3). In an effort to not conflate the two, TRAD is executed with and without local symmetric BFS permutations and the representative performance metric is taken as the maximum of the two. Similarly, we take the maximum performance of DLB-MPK with and without recursion as the representative performance metric.

All numerical results are validated against Intel's Math Kernel Library.[6] Benchmarks are repeated several times, and the median performance is taken as the representative performance metric. Error bars are excluded from our plots as run-to-run deviations are less than 5%.

6.2. Parameter study

RACE provides tuning parameters to optimize performance for the specific hardware under consideration. In this section, we perform a parameter study on ICL with the matrix

Table 4. Benchmark matrices.

Matrix	N_r	N_{nz}	N_{nzr}	CRS Size [MiB]
inline_1	503,712	36,816,342	73.0	423
Emilia_923	923,136	41,005,206	44.4	473
Ldoor	952,203	46,522,475	48.8	536
af_shell10	1,508,065	52,672,325	34.9	609
Hook_1498	1,498,023	60,917,445	40.6	703
Geo_1438	1,437,960	63,156,690	43.9	728
Serena	1,391,349	64,531,701	46.3	744
bone010	986,703	71,666,325	72.6	824
audikw_1	943,695	77,651,847	82.2	892
Channel-500x100	4,802,000	85,362,744	17.7	995
Long_Coup_dt0	1,470,152	87,088,992	59.2	1002
dielFilterV3real	1,102,824	89,306,020	80.9	1026
nlpkkt120	3,542,400	96,845,792	27.3	1122
ML_Geer	1,504,002	110,879,972	73.7	1275
Lynx68	6,811,350	111,560,826	16.3	1303
Flan_1565	1,564,794	117,406,044	75.0	1350
Cube_Coup_dt0	2,164,760	127,206,144	58.7	1464
Bump_2911	2,911,419	127,729,899	43.9	1473
van_Stokes_4M	4,382,246	131,577,616	30.0	1523
Queen_4147	4,147,110	329,499,284	79.5	3787
nlpkkt200	16,240,000	448,225,632	27.6	5191
nlpkkt240	27,993,600	774,472,352	27.6	8970
Lynx649	64,950,632	978,866,282	15.0	11,450
Lynx1151	115,187,228	1,934,489,424	16.8	22,578

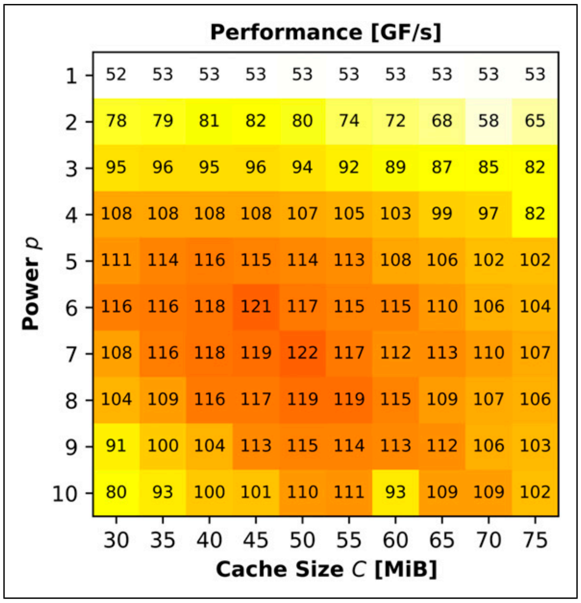


Figure 8. Parameter study with ML_Geer on one ICL node, scanning $p \in \{1, 2, \ldots, 10\}$ and $C \in \{30, 35, \ldots, 75\}$.

ML_Geer to better understand the influence of these parameters on the performance of DLB-MPK. This will also serve as an example of how one could perform such an investigation.

We focus here only on the parameters p and C since we have fixed the recursion depth s_m as described before. In Figure 8, we scan various powers p and cache sizes C when performing DLB-MPK on ML_Geer. We use METIS as the global partitioner, pinning one MPI process to each of the four ccNUMA domains compactly. We observe performance increasing until a local maximum at $p = 7$ and $C = 50$, after which performance degrades for higher values of p and C. The exact reasons for this behavior are described by Alappat et al. (2022), but we summarize the main points here. As the value of p increases, the number of levels to be kept in the cache also increases. Once the cumulative size of all levels needed for blocking is larger than the total cache size, measures must be taken to reduce the size of the levels so that they can still all fit into the cache. This incurs higher synchronization costs in RACE since all threads must be synchronized after computing each of the smaller and more numerous levels. A C which is smaller than the total cache size also generates smaller levels, again increasing synchronization overheads. But a C which is larger than the total cache size will provoke more cache misses.

From Table 2, we know that one ccNUMA domain on ICL has 49 MiB L2 + L3 aggregate cache, so we would expect an optimal value for the parameter C to be around this range. The optimal C does not always correspond directly with the amount of available cache per process, due to a safety factor internal to RACE. A user of DLB-MPK would manually tune these two parameters to achieve the best possible performance for their use case[7]. Notice that the DLB-MPK performance for $p = 1$ stays roughly constant as cache size grows. This corroborates our claim from Section 3 that computing $y \leftarrow A^p x$ for $p = 1$ can not make use of cache blocking.

6.3. Performance results summary

In this section, we give a concise high-level single-node performance summary of DLB-MPK and TRAD on our benchmark matrices.

Figure 9 shows the node-level performance of DLB-MPK (red, right bars) as compared to TRAD (blue, left bars) for optimally tuned parameters C and p. The matrices are ordered according to their size, and the vertical dashed line indicates the L2 + L3 aggregate cache size of the architecture. On MIL some matrices fit in the cache, that is, lie to the left of the dashed vertical line. In this regime, DLB-MPK has no benefit compared to TRAD since the matrices already fit in the cache, and cache blocking is pointless. The behavior is very similar with cache-resident matrices on ICL and SPR, although for this work, we chose large in-memory matrices to elucidate the situations in which DLB-MPK is advantageous to use.

The short black line in or above each TRAD bar is the memory-bound roofline performance limit of SpMV for the given matrix and hardware computed using Equation (4). As TRAD performs back-to-back SpMVs, ideally one would expect the performance of TRAD for large in-memory matrices to be below the roofline limit. However, in many cases, close to the cache boundary (just to the right of the dashed vertical line), TRAD's performance exceeds the roofline limit by a small margin. This is due to the residual caching effects as also observed in Figure 7 for the load-only benchmark. As predicted for SPR and MIL, TRAD exhibits these residual caching effects until the matrix size is up to 2400 MiB, that is, until `van_stokes_4M`, after which point the performance of TRAD is almost always lower than the upper roofline bound.

In general, towards the right of the dashed vertical line (the in-memory matrices), DLB-MPK has a significant advantage over TRAD. The performance of DLB-MPK is much higher than the roofline prediction and TRAD, due to cache blocking resulting in lower main memory traffic. We observe an average (maximum) speedup of 1.6× (2.5×), 1.7× (2.4×), and 1.6× (2.7×) for large in-memory datasets on ICL, SPR, and MIL, respectively. The numbers annotated above DLB-MPK bars show the optimal power value tuned in the range of $p \in \{1, 2, ..., 12\}$. On ICL and MIL, both TRAD and DLB-MPK perform poorly with the `Lynx1151` matrix. This is due to our benchmark requiring multiple copies of internal data structures, which is not uncommon in real applications. Combined with the size of `Lynx1151`, this leads to frequent remote ccNUMA domain accesses. This problem does not occur on SPR, as this architecture has roughly twice the memory per ccNUMA domain versus ICL, and roughly four times that of MIL.

As shown by Alappat et al. (2022), the preprocessing costs associated with RACE to collect the level structure are typically equivalent to 5 to 50 SpMVs (increasing with the recursion stage s_m). The preprocessing costs associated with the introduction of MPI are minimal since the only additional steps are the identification and collection of the boundary vertices. As this is equivalent to each MPI process scanning its local rows once, this overhead is equivalent to roughly one additional SpMV.

6.4. Strong scaling

It is frequently more important to understand the scaling characteristics of performance rather than taking a snapshot for a single parameter configuration and input. We now investigate how the performance of DLB-MPK grows with an increasing number of ccNUMA domains. The experiment is conducted on eight nodes of SPR. As in the previous section, the power p is tuned in the range $p \in \{1, 2, ..., 12\}$.

Figure 10(a) shows the performance of TRAD versus DLB-MPK for both $p = 4$ and $p = 6$ on `Lynx1151`. The reason that the performance for both A^4x and A^6x is shown is that both powers are optimal for `Lynx1151`, depending on the scale one considers. For a single SPR node, $p = 4$ performs better than $p = 6$. However, once more cache becoming available when scaling to multiple nodes, $p = 6$ performs better.

Figures 10(b) and 10(c) show on the right y-axis how the two overheads introduced in Sections 4 and 5 – MPI overhead (O_{MPI} from Equation (1)) and DLB-MPK overhead ($O_{\text{DLB-MPK}}$ from Equation (3)) – scale for `Lynx1151` with a growing number of processes. On the left y-axis, we show parallel efficiency for strong scaling $\varepsilon_{\text{strong}} := T_1/(nT_n)$ for DLB-MPK, where T_1 is the time required by DLB-MPK for a single process and is T_n the time required by n processes. Since we have a fixed workload, we can choose $T_n = 1/P_n$, where P_n is the performance of DLB-MPK on n ccNUMA domains.

Since we are blocking for a higher power in Figure 10(c), it makes sense that $O_{\text{DLB-MPK}}$ will be higher than in Figure 10(b), since there will be fewer vertices contained in the bulk structure M as described in Section 5. MPI overhead will be the same for both $p = 4$ and $p = 6$ since

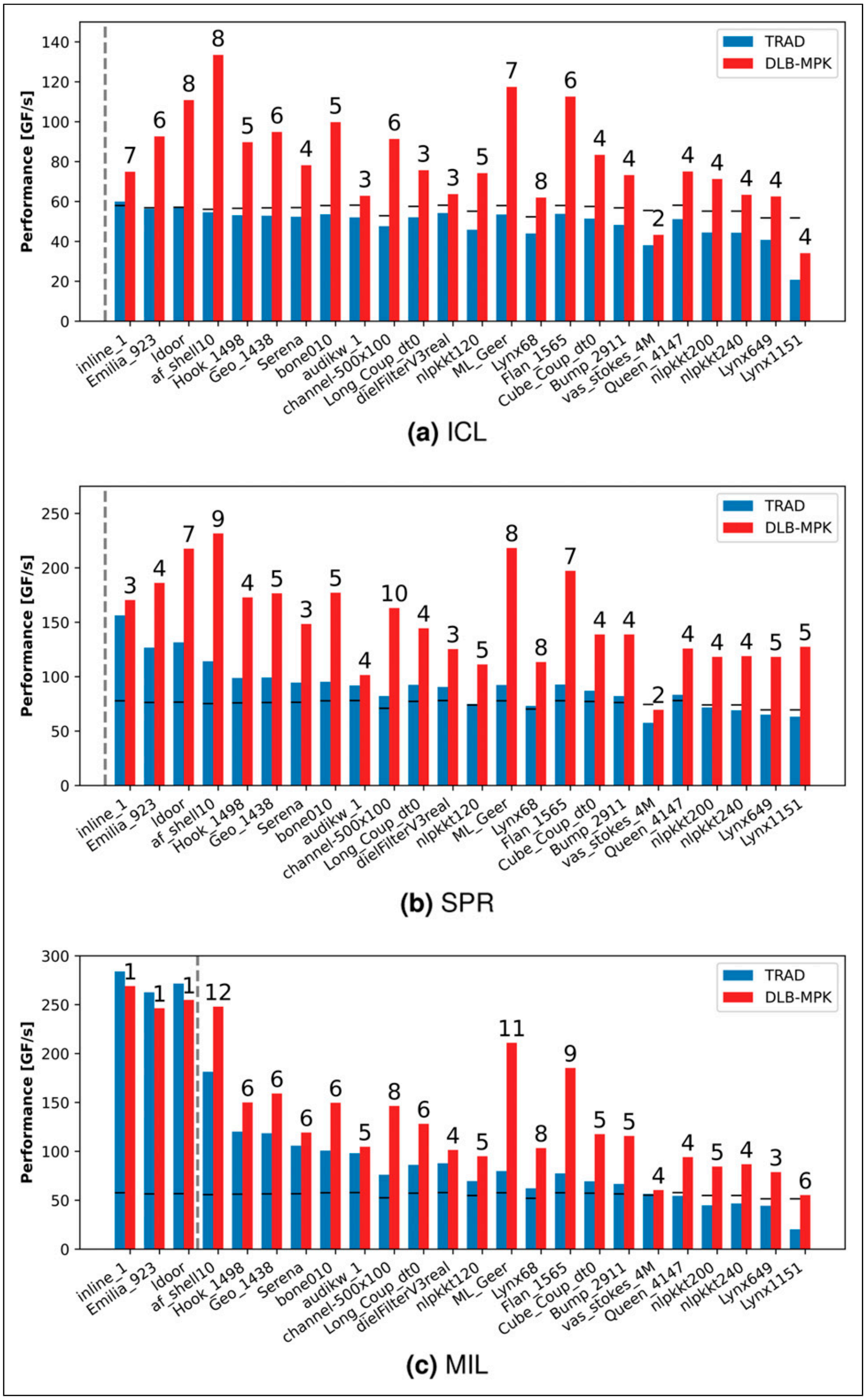
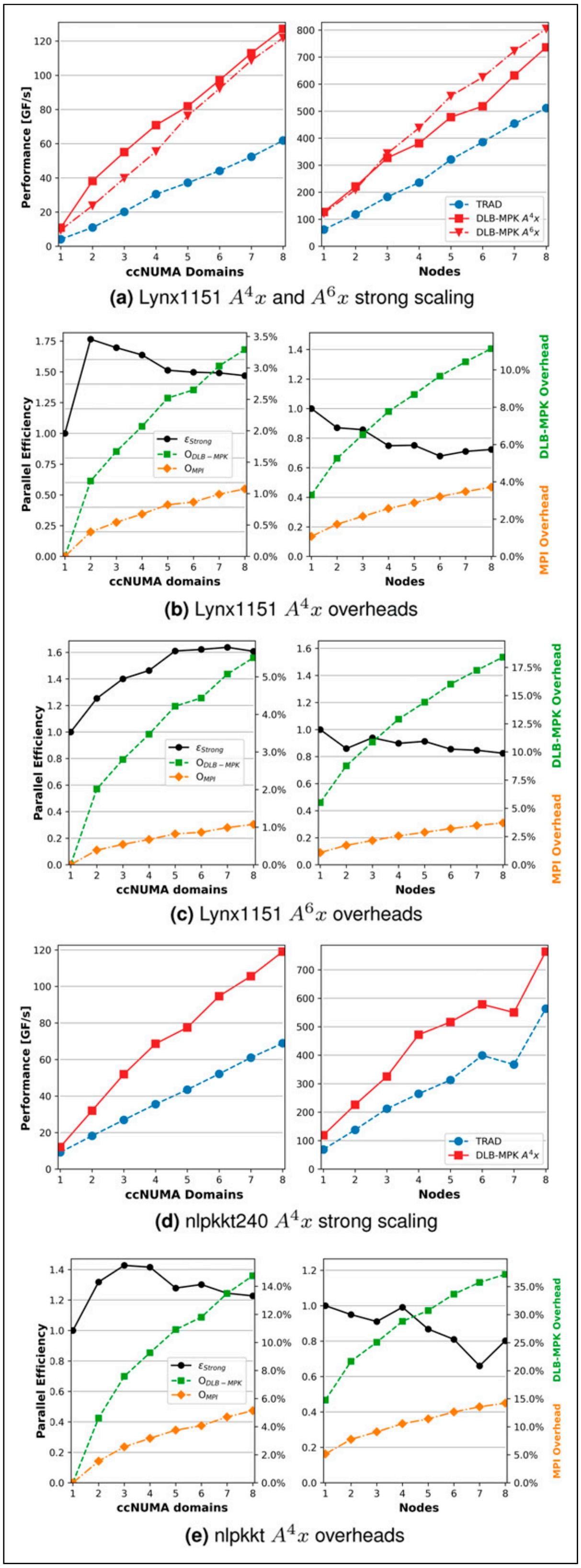


Figure 9. Node-level performance summary for benchmark matrices in Table 4, ordered by CRS size, on the CPUs under consideration. For each matrix, the numbers above the bars denote the optimal power p for which DLB-MPK was tuned. The horizontal black lines are the roofline predictions for TRAD according to Equation (4). The vertical dashed line represents the aggregate L2 + L3 cache size.



(a) Lynx1151 A^4x and A^6x strong scaling

(b) Lynx1151 A^4x overheads

(c) Lynx1151 A^6x overheads

(d) nlpkkt240 A^4x strong scaling

(e) nlpkkt A^4x overheads

Figure 10. Single- (left) and multi-node (right) strong scaling performance and overhead results for `Lynx1151` and `nlpkkt240` on SPR nodes.

O_{MPI} depends only on the matrix structure and number of MPI processes.

We see $\varepsilon_{\mathrm{strong}} \geq 1$ in the intra-node regime in the left subfigure in Figure 10(b) where we normalize $\varepsilon_{\mathrm{strong}}$ against the time taken by DLB-MPK on one ccNUMA domain. The sharp increase in $\varepsilon_{\mathrm{strong}}$ from one to two processes is due to the additional cache available with the second ccNUMA domain. As the number of processes increases, we gain access to more cache, yet the MPI costs grow as we communicate with other processes that are physically farther away. Alternatively, in the right subfigure in Figure 10(b) $\varepsilon_{\mathrm{strong}} \leq 1$ for the inter-node regime, where we normalize $\varepsilon_{\mathrm{strong}}$ against the time taken by DLB-MPK on one entire node. We see the impact of MPI on a larger scale here, as inter-node communication latency is much higher than within a single node. Parallel efficiency reaches a higher maximum with $p = 4$ for the intra-node case as shown in Figure 10(b), but is sustained for larger MPI process for the inter-node case with $p = 6$ as shown in Figure 10(c).

Figures 10(d) and 10(e) show how the performance and overheads of DLB-MPK scale for `nlpkkt240`. Although the maximum performance attained is roughly the same as for `Lynx1151` on all eight nodes, `nlpkkt240` exhibits different scaling behavior. There are two reasons for the strange scaling behavior of `nlpkkt240`.

First, the matrix structure is much “worse,” i.e., the sparsity pattern is not banded, and there are many non-zero elements that are far from the diagonal. This will not only increase DLB-MPK overhead as there are fewer levels (i.e., fewer vertices inside the bulk structure M), but it will also increase the MPI overhead as there are more halo elements on each process. The second reason is that we recognize residual caching effects after around 4–5 nodes by the sharp jumps in the performance of both TRAD and DLB-MPK. From Table 2, we can compute that if `Lynx1151` is partitioned roughly equally across eight nodes, about 2.8 GiB of matrix data lies on each node. Since this is above 2400 MiB, we will not see any residual caching effects. But if we partition `nlpkkt240` in the same manner, only about 1.1 GiB of matrix data will reside on each node.

This is not uncommon and poses a difficulty when performing scaling studies with DLB-MPK. Most matrices from Suite Sparse are simply not large enough to fully take advantage of DLB-MPK.

7. Application: Chebyshev time propagation

A common application that can benefit from the DLB-MPK is the Chebyshev method for the time evolution of quantum states as shown by Tal-Ezer and Kosloff (1984); Fehske et al. (2009). In this section, we demonstrate the advantage of cache blocking in the context of this application and investigate the weak scaling characteristics of DLB-MPK.

Given a Hamiltonian $\widehat{H}$ and an initial state $|\psi(0)\rangle$, the goal is to solve $|\psi(\tau)\rangle = e^{-i\tau\widehat{H}}|\psi(0)\rangle$ for some target time τ. This can be achieved by splitting the exponential into multiple small time steps $\delta\tau$ and approximating each as a polynomial in $\widehat{H}$. Using an expansion in Chebyshev polynomials and keeping the first $\mathcal{M}+1$ terms leads to the following approximation for a single time step:

$$\begin{aligned} |\psi(\tau+\delta\tau)\rangle &= e^{-i\delta\tau\widehat{H}}|\psi(\tau)\rangle \\ &\approx J_0(\delta\tau)|v_0\rangle + 2\sum_{k=1}^{\mathcal{M}}(-i)^k J_k(\delta\tau)|v_k\rangle, \end{aligned} \tag{5}$$

where $J_k(\delta\tau)$ is the Bessel function of the first kind of order k. The states $|v_k\rangle$ are calculated recursively using the relations

$$|v_{k+1}\rangle = 2\widehat{H}|v_k\rangle - |v_{k-1}\rangle, \tag{6}$$

$$|v_0\rangle = |\psi(\tau)\rangle, \quad |v_1\rangle = \widehat{H}|\psi(\tau)\rangle, \tag{7}$$

which primarily amounts to a sequence of $\mathcal{M}$ SpMVs when $\widehat{H}$ is given as a sparse matrix. Since these SpMVs are the computational hot spot of the algorithm, the Chebyshev time-propagation method can potentially be sped up significantly by using the DLB-MPK.

We demonstrate the Chebyshev time propagation for the `Anderson` matrix. Physically, it represents a single-particle Hamiltonian for electrons in a disordered medium

$$\widehat{H} = \frac{W}{2}\sum_{\boldsymbol{r}} w_{\boldsymbol{r}}|\boldsymbol{r}\rangle\langle\boldsymbol{r}| - t\sum_{\langle\boldsymbol{r},\boldsymbol{r}'\rangle}|\boldsymbol{r}\rangle\langle\boldsymbol{r}'|, \tag{8}$$

where the states $|\boldsymbol{r}\rangle$ with $\boldsymbol{r} = (x, y, z) \in \mathbb{Z}^3$ correspond to sites in a cubic lattice, and the second summation is over nearest-neighbor pairs. The parameter W determines the strength of the disorder potential. Here, we assume an uncorrelated random potential, with $w_{\boldsymbol{r}}$ drawn uniformly from the interval $[-1, 1]$. Equation (8) is a paradigmatic model for the metal-insulator transition due to Anderson localization as described by Anderson (1958): while the system is a conductor for small W, it becomes an insulator above some critical value W_c. For $W > W_c$, the eigenstates of $\widehat{H}$ are localized, that is, they are restricted to a finite region outside of which their weight decreases exponentially. As a consequence, an initially local state, for example, a Gaussian wave packet

$$|\psi(0)\rangle \propto \sum_{\boldsymbol{r}} e^{-\frac{r^2}{2\sigma^2}+i\boldsymbol{k}_0\boldsymbol{r}}|\boldsymbol{r}\rangle \tag{9}$$

of width σ, does not diffuse and instead remains localized indefinitely. Moreover, it was recently shown that the density distribution $\rho(\boldsymbol{r}, \tau) = |\langle\boldsymbol{r}|\psi(\tau)\rangle|^2$ at long times τ is insensitive to the initial momentum $\boldsymbol{k}_0$ of the wave packet, so that the center of mass of the wave packet must return to its origin.

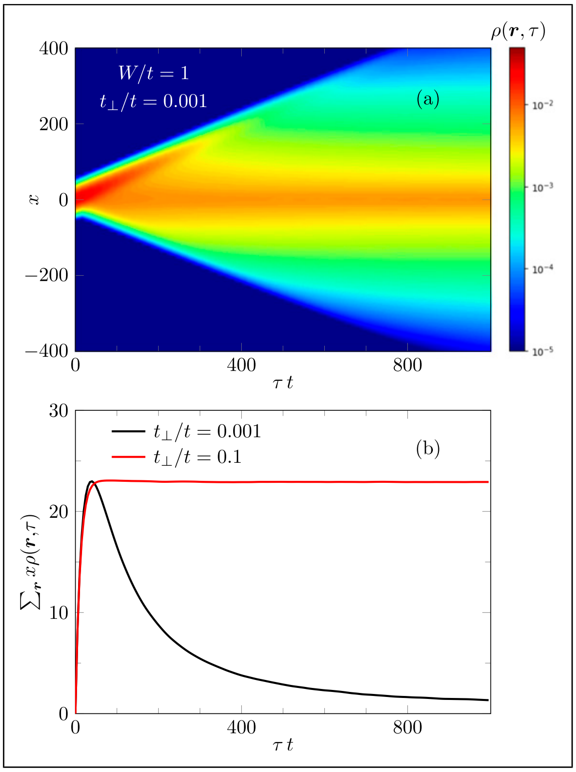


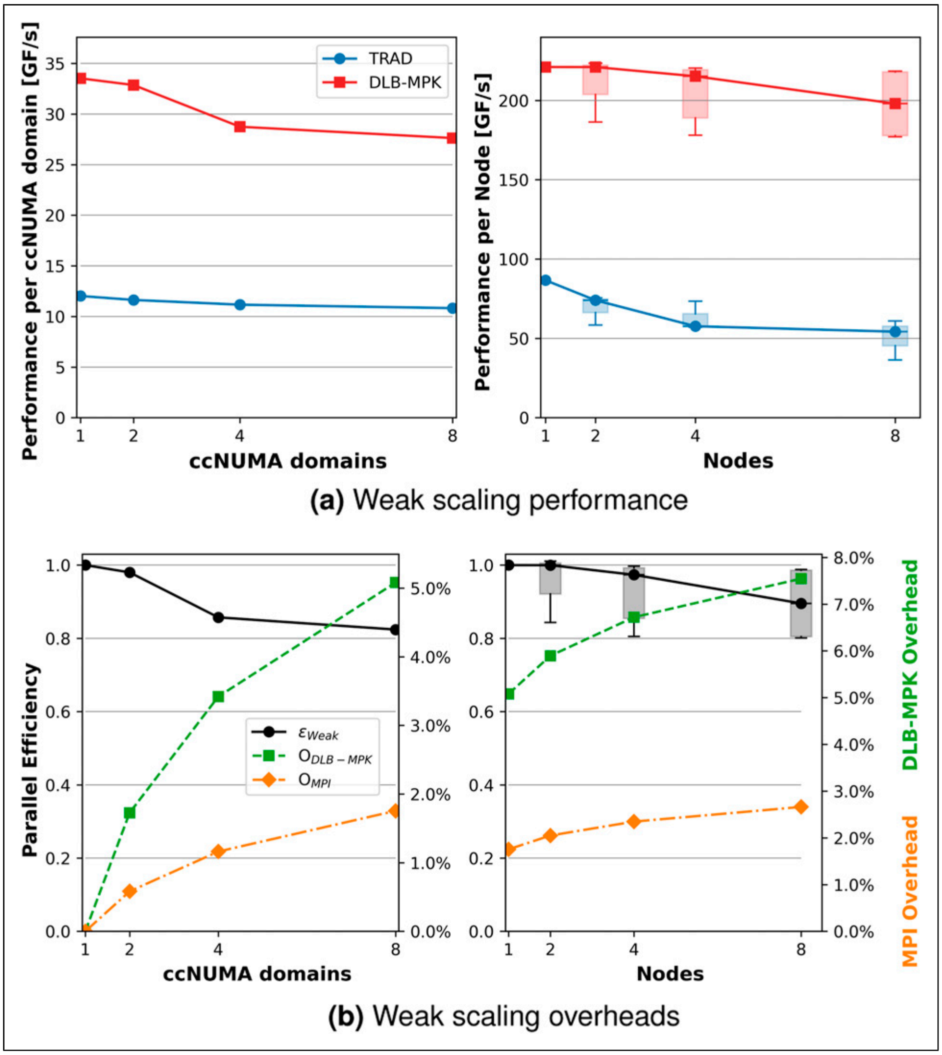
Figure 11. Time evolution of a wave packet (Equation (9)) with width σ = 20 and momentum $\boldsymbol{k}_0 = \pi/2\boldsymbol{e}_x$. Panel (a) shows the time-dependent density distribution in the localized regime with parameters $t_\perp/t = 0.001$ and $W/t = 1$. The center-of-mass motion is displayed in panel (b), which also includes data for a delocalized system with $t_\perp/t = 0.1$. We used a finite rectangular system with dimensions $L_y = L_z = 100$ and $L_x = 3000$ for the simulations, and averaged the results over 50 runs with different random potentials $w_{\boldsymbol{r}}$.

Prat et al. (2019); Janarek et al. (2020, 2022) have numerically investigated this "quantum boomerang effect" for various models using the Chebyshev time-propagation method. Here, we consider a variant of the Anderson model in which the hopping parameter t along the y and z axis is replaced by $t_\perp < t$, that is, a system of weakly coupled chains. By tuning $t_\perp$, a localization transition can be induced at fixed disorder W, as shown by Zambetaki et al. (1997). Figure 11 displays results for the time evolution of a wave packet moving in the x direction with $\boldsymbol{k}_0 = \pi/2\boldsymbol{e}_x$. As expected, the center of mass approaches $x \approx 0$ for long times in the localized system with small $t_\perp/t = 0.001$, while it remains at a finite displacement in the delocalized one with $t_\perp/t = 0.1$. In addition, the density distribution $\rho(\boldsymbol{r}, \tau)$ for $t_\perp/t = 0.001$ becomes stationary at long times τ, that is, the wave packet stops spreading, which is another signature of Anderson localization.

To demonstrate the effectiveness of DLB-MPK on a real-world application, we now perform a weak scaling study on the above-described Chebyshev time propagation method

Table 5. Anderson matrix Configurations.

# ccNUMA Domains	(Lx, Ly, Lz)	N_r	N_{nz}	N_{nzr}	CRS Size [MiB]
1	(160, 160, 160)	4,096,000	28,518,400	7.0	342
2	(320, 160, 160)	8,192,000	57,088,000	7.0	685
4	(320, 320, 160)	16,384,000	114,278,400	7.0	1370
8	(320, 320, 320)	32,768,000	228,761,600	7.0	2743
16	(640, 320, 320)	65,536,000	457,728,000	7.0	5488
32	(640, 640, 320)	131,072,000	915,865,600	7.0	10,981
64	(640, 640, 640)	262,144,000	1,832,550,400	7.0	21,972



(a) Weak scaling performance

(b) Weak scaling overheads

Figure 12. Weak scaling investigation using the Anderson matrices in Table 5 on a single (left) and multiple (right) SPR node(s). Boxplots are given when performance fluctuates by more than 5%.

on SPR nodes. The Anderson matrix is generated using the ScaMaC matrix generator.[8]

Previous state-of-the-art implementations of the Chebyshev time propagation method perform back-to-back SpMVs to compute $|v_{k+1}\rangle$ for successive time steps. However, these SpMVs can be accelerated by cache blocking using DLB-MPK. In order to be well outside of the residual caching effects on SPR, the study is constructed so that we always have about 342 MiB of matrix data per ccNUMA domain, i.e. 2743 MiB per node. Compared with our observations in Figure 7, this will be far outside of the cache. Specifically, we double the number of lattice sites in a selected direction (x, y, or z) in order to double the number of rows in the matrix. The `Anderson` matrix configurations used can be seen in Table 5.

As previously mentioned, Equation (6) represents a series of $\mathcal{M}$ SpMVs with the same matrix data $\widehat{H}$. Since $\mathcal{M}$ is on the order of 100-1000s, we must choose a factor $p_m < \mathcal{M}$ by which we block the matrix data in the cache.[9]

In the same manner described in Section 6.2, we first tune p_m and the cache size C to obtain optimal node-wide performance. After such an investigation, DLB-MPK yields the highest performance on SPR at $p_m = 8$, $C = 35$ MiB. Note that C is much smaller than the available L2 + L3 aggregate cache given for SPR in Table 4. This is expected since there are other data structures in the application that will also occupy space in the cache. We define parallel efficiency in the weak scaling case as $\varepsilon_{\text{weak}} := T_1/T_n$. Since our workload now increases with the number of processes, we choose $T_1 = 1/P_1$ and $T_n = n/P_n$, where P_n is still the performance of DLB-MPK on n ccNUMA domains.

Figure 12 shows the weak scaling performance per MPI process and the overheads of DLB-MPK applied to the Chebyshev time propagation method using various sizes of the `Anderson` matrix. We double the innermost spatial dimension last to respect layer conditions for cache blocking. In the single-node regime, we took the median of five executions of both TRAD and DLB-MPK as the representative performance, yet the fluctuations were less than 5%. For the multi-node regime, we included the box-and-whisker plots in addition to the median for both TRAD and DLB-MPK, since we noticed higher performance fluctuations when scaling to multiple nodes. The tip and tail of the whiskers are the minimum and maximum performance observed, and the boxes denote the interquartile range of the five executions. Our selected affinity is the same as described in Section 6.

Algorithm 2 Distributed Level-Blocked MPK

```
Input:
  double $x[N_{r,i} + N_{h,i}]$
  commFuncType haloComm
  spmvFuncType SpMV
  levelPointer $I$
  sparseMatrix $A_i$
  int $p_m$
Output:
  double $y[N_{r,i} + N_{h,i}, p_m]$

  $y[:,0] \leftarrow x$;
  $y[:,0] \leftarrow$ haloComm($y[:,0]$);
  $[x, y] \leftarrow$ localLBMPK($x, y$, SpMV);
  for $p \leftarrow 1, \ldots, p_m - 1$ do
      $y[:,p] \leftarrow$ haloComm($y[:,p]$);
      for $k \leftarrow 1, \ldots, p_m - p$ do
          $y[I[k], p+1] \leftarrow$ SpMV($y[I[k], p], A_i[I[k], :]$);
      end for
  end for
```

DLB-MPK maintains a speed-up of about 2.8× as compared against TRAD for one and two ccNUMA domains. When moving from 2 to 4, and then to 8 ccNUMA domains, speed-up drops to about 2.5×. In the multi-node regime (i.e., past eight ccNUMA domains) we maintain a speed-up of 2× to 3.3× for the worst performing DLB-MPK executions, and 2.5× to 4× for the best.

8. Summary

We have motivated and developed a novel cache-blocked MPI-parallel matrix power kernel based on the level-blocking capabilities of RACE. The resulting algorithm extends the ideas developed by Alappat et al. (2022) by first organizing local vertices on each MPI process by their distance k from the halo buffer into levels I_k, and then interleaving a local cache blocking MPK with communication steps to fulfill data dependencies. Our algorithm, DLB-MPK, has been shown to be efficient in that it does not increase MPI overhead when compared to the traditional MPK implementation. This is because these collections of vertices I_k grow inwards, keeping the number of halo elements constant while slightly reducing the efficiency of cache blocking. Furthermore, DLB-MPK has the advantage that it uses the same computation and halo communication routines as a traditional distributed-memory parallel MPK. Therefore, it can be easily integrated into existing libraries and can be used as a drop-in replacement for traditional distributed-memory parallel MPK.

We used the roofline model to explain expected performance behavior using key metrics extracted from our selection of test hardware platforms. After that, we gave an example of how one may tune DLB-MPK for optimal performance. To evaluate the performance of DLB-MPK, we first gave a snapshot summary of the optimally tuned performance as compared to the traditional MPK on modern multicore CPUs. We observed a node-wide average (maximum) speedup of 1.6× (2.5×), 1.7× (2.4×), and 1.6× (2.7×) for large in-memory datasets on ICL, SPR, and MIL, respectively

Then, strong scaling characteristics of DLB-MPK were studied, where we observed the influence of caches and communication on performance. Finally, DLB-MPK was integrated into an application using a Chebyshev method for the time evolution of quantum states for the Anderson model of localization. This enabled us to perform weak scaling investigations on up to eight Sapphire Rapids nodes, in which we observed a speed-up of up to 4× when compared to the traditional MPK implementation. Future work will be directed towards the integration of GPGPU support for DLB-MPK.

Acknowledgements

The authors gratefully acknowledge the computing time provided to them on the high-performance computer Noctua two at the NHR Center PC2. These are funded by the German Federal Ministry of Education and Research and the state governments participating on the basis of the resolutions of the GWK for the national high-performance computing at universities (https://www.nhr-verein.de/unsere-partner). We would also like to thank Johannes Langguth at Simula Research Laboratory for the thought provoking discussions, as well as his assistance in generating the Lynx matrices for our performance investigations. H.F. and G.W. acknowledge the hospitality at Los Alamos National Laboratory and the paper-related discussions with A. Saxena and A. R. Bishop.

Declaration of conflicting interests

The author(s) declared no potential conflicts of interest with respect to the research, authorship, and/or publication of this article.

Funding

The author(s) disclosed receipt of the following financial support for the research, authorship, and/or publication of this article: This work was supported by the Bundesministerium für Bildung und Forschung 16ME0708.

ORCID iDs

Dane Lacey https://orcid.org/0009-0008-5896-9646
Christie Alappat https://orcid.org/0000-0003-4548-8727
Florian Lange https://orcid.org/0000-0003-3389-1711
Georg Hager https://orcid.org/0000-0002-8723-2781

Notes

1. We conform to the standard of describing quantities as powers of two, and performance metrics as powers of 10, for example, 1 MiB = 2^{20} B, 1 M flop/s = 10^6 flop/s.
2. https://www.techpowerup.com/cpu-specs/.
3. https://www.techpowerup.com/gpu-specs/.
4. RACE internally handles non-symmetric matrices as symmetric, filling in non-symmetric entries to aid the collection of vertices into levels. These filled-in elements do not appear on the actual matrix, and it is therefore sufficient to discuss only symmetric matrices.
5. This technique is very similar to the well-known Reverse Cuthill-McKee reordering. The main difference is how the root vertex is selected.
6. We are not bound to any particular matrix format, but choose CRS for its ubiquity in the literature.
7. In practice, one would choose a value close to the cache size of the hardware for C. As we have seen from Figure 8, this is close to the optimal value. However, the optimal p value might need some tuning but in many practical cases it is determined by the application that we run. For example in case of s-step GMRES method, it is typically the value of s.
8. https://software.intel.com/en-us/mkl.
9. https://alvbit.bitbucket.io/scamac_docs/index.html.

References

Alappat C, Basermann A, Bishop AR, et al. (2020a) A recursive algebraic coloring technique for hardware-efficient symmetric sparse matrix-vector multiplication. *ACM Trans. Parallel Comput* 7(3): 1–37. DOI: 10.1145/3399732.

Alappat CL, Hofmann J, Hager G, et al. (2020b) Understanding HPC benchmark performance on Intel broadwell and cascade lake processors. In: Sadayappan P, Chamberlain BL, Juckeland G, et al. (eds) *High Performance Computing*. Cham: Springer International Publishing, pp. 412–433. ISBN 978-3-030-50743-5.

Alappat C, Hager G, Schenk O, et al. (2022) Level-based blocking for sparse matrices: sparse matrix-power-vector multiplication. *IEEE Transactions on Parallel and Distributed Systems* 34(2): 581–597. DOI: 10.1109/TPDS.2022.3223512.

Alappat C, Thies J, Hager G, et al. (2023) Algebraic temporal blocking for sparse iterative solvers on multi-core CPUs. *The International Journal of High Performance Computing Applications*. 2024;0(0). DOI: 10.1177/10943420241283828.

Anderson PW (1958) Absence of diffusion in certain random lattices. *Physical Review A* 109: 1492–1505. DOI: 10.1103/PhysRev.109.1492.

Davis TA and Hu Y (2011) The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software* 38(1): 1–25. DOI: 10.1145/2049662.2049663.

Demmel J, Hoemmen M, Mohiyuddin M, et al. (2008) Avoiding communication in sparse matrix computations. In: *2008 IEEE International Symposium on Parallel and Distributed Processing*, pp. 1–12. DOI: 10.1109/IPDPS.2008.4536305.

Fehske H, Schleede J, Schubert G, et al. (2009) Numerical approaches to time evolution of complex quantum systems. *Physics Letters A* 373(25): 2182–2188. DOI: 10.1016/j.physleta.2009.04.022.

Gao J, Liu B, Ji W, et al. (2024) A systematic literature survey of sparse matrix-vector multiplication. Preprint: arXiv: 2404.06047.

Janarek J, Delande D, Cherroret N, et al. (2020) Quantum boomerang effect for interacting particles. *Physical Review A* 102: 013303. DOI: 10.1103/PhysRevA.102.013303.

Janarek J, Grémaud B, Zakrzewski J, et al. (2022) Quantum boomerang effect in systems without time-reversal symmetry. *Physical Review B: Condensed Matter* 105: L180202. DOI: 10.1103/PhysRevB.105.L180202.

Karypis G and Kumar V (1998) METIS: a software package for partitioning unstructured graphs, partitioning meshes, and computing fill-reducing orderings of sparse matrices.

Kreutzer M, Hager G, Wellein G, et al. (2014) 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(5): C401–C423. DOI: 10.1137/130930352.

Langguth J, Sourouri M, Lines GT, et al. (2015) Scalable heterogeneous CPU-GPU computations for unstructured tetrahedral meshes. *IEEE Micro* 35(4): 6–15. DOI: 10.1109/MM.2015.70.

Langguth J, Arevalo H, Hustad KG, et al. (2019) Towards detailed real-time simulations of cardiac arrhythmia. In: *2019 Computing in Cardiology (CinC)*, p. 1. DOI: 10.22489/CinC.2019.301.

Loe JA, Thornquist HK and Boman EG (2020) Polynomial preconditioned gmres in trilinos: practical considerations for high-performance computing. In: *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, pp. 35–45. DOI: 10.1137/1.9781611976137.4.

Luxburg U (2004) A tutorial on spectral clustering. *Statistics and Computing* 17: 395–416. DOI: 10.1007/s11222-007-9033-z.

McQueen J, Meilă M, VanderPlas J, et al. (2016) Megaman: scalable manifold learning in Python. *Journal of Machine Learning Research* 17(148): 1–5.

Moghimi D (2023) Downfall: exploiting speculative data gathering. In: *32th USENIX Security Symposium (USENIX Security 2023)*.

Mohiyuddin M, Hoemmen M, Demmel J, et al. (2009) Minimizing communication in sparse matrix solvers. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, SC '09*. New York, NY, USA: Association for Computing Machinery, pp. 1–12. ISBN 9781605587448. DOI: 10.1145/1654059.1654096.

Prat T, Delande D and Cherroret N (2019) Quantum boomeranglike effect of wave packets in random media. *Physical Review A* 99: 023629. DOI: 10.1103/PhysRevA.99.023629.

Simpson T, Pasadakis D, Kourounis D, et al. (2018) Balanced graph partition refinement using the graph p-laplacian. In: *Proceedings of the Platform for Advanced Scientific Computing Conference, PASC '18*. New York, NY, USA: Association for Computing Machinery, 1–11. ISBN 9781450358910. DOI: 10.1145/3218176.3218232.

Tal-Ezer H and Kosloff R (1984) An accurate and efficient scheme for propagating the time dependent Schrödinger equation. *The Journal of Chemical Physics* 81(9): 3967–3971. DOI: 10.1063/1.448136.

Treibig J, Hager G and Wellein G (2010) LIKWID: a lightweight performance-oriented tool suite for x86 multicore environments. In: *2010 39th International Conference on Parallel Processing Workshops*, 207–216. DOI: 10.1109/ICPPW.2010.38.

Vatai E, Singhal U and Suda R (2020) Diamond matrix powers kernels. In: *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region, HPCAsia2020*. New York, NY, USA: Association for Computing Machinery, 102–113. ISBN 9781450372367. DOI: 10.1145/3368474.3368494.

Vuduc RW and Demmel JW (2003) *Automatic performance tuning of sparse matrix kernels*. PhD Thesis, University of California, Berkeley. AAI3121741.

Williams S, Waterman A and Patterson D (2009) Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM* 52(4): 65–76. DOI: 10.1145/1498765.1498785.

Yamazaki I, Anzt H, Tomov S, et al. (2014a) Improving the performance of CA-GMRES on multicores with multiple GPUs. In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, 382–391. DOI: 10.1109/IPDPS.2014.48.

Yamazaki I, Rajamanickam S, Boman EG, et al. (2014b) Domain decomposition preconditioners for communication-avoiding Krylov methods on a hybrid CPU/GPU cluster. In: *SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 933–944. DOI: 10.1109/SC.2014.81.

Zambetaki I, Li Q, Economou EN, et al. (1997) Localization in weakly coupled planes and weakly coupled wires. *Physical Review B: Condensed Matter* 56: 12221–12231. DOI: 10.1103/PhysRevB.56.12221.

Author biographies

Dane Lacey received a bachelor's degree in Applied Mathematics from the University of Utah and a master's degree in Computational and Applied Mathematics from Friedrich-Alexander-Universität Erlangen-Nürnberg. He is currently pursuing a Ph.D. under the supervision of Prof. Gerhard Wellein. His research focuses on hardware-aware building blocks for sparse linear algebra, with an emphasis on performance engineering for distributed-memory algorithms and iterative linear solvers.

Christie Alappat holds a master's degree with honors from the Bavarian Graduate School of Computational Engineering, Friedrich-Alexander-Universität Erlangen-Nürnberg. He is currently in the final stages of completing his doctoral studies under the supervision of Professor Gerhard Wellein while simultaneously being employed as a math algorithm engineer at Intel. His research interests encompass performance engineering, sparse matrix and graph algorithms, iterative linear solvers, and eigenvalue computation. He is the author of the RACE open-source software framework, which facilitates the acceleration of computationally demanding sparse linear algebra operations on modern compute devices.

Florian Lange holds a master’s degree and a Ph.D. in Theoretical Physics from the University of Greifswald. He is currently working at Erlangen National High Performance Computing Center (NHR@FAU) as a domain expert for physics in the Training & Support Division. His research interests are low-dimensional condensed matter systems, tensor network methods, and quantum computing.

Georg Hager holds a Ph.D. and a Habilitation degree in Computational Physics from the University of Greifswald. He heads the Training and Support Division at Erlangen National High Performance Computing Center (NHR@ FAU) and is an associate lecturer at the Institute of Physics of the University of Greifswald. Recent research includes architecture-specific optimization strategies for current microprocessors, performance engineering of scientific codes on chip and system levels, and structure formation in large-scale parallel codes.

Holger Fehske received a Ph.D. in physics from the University of Leipzig, and a Habilitation degree and Venia Legendi in theoretical physics from the University of Bayreuth. In 2002, he became a full professor at the University of Greifswald. There the held the chair for Complex Quantum Systems and worked in the fields of solid-state theory, quantum statistical physics, light-matter interaction, quantum informatics, plasma physics, and computational physics. He is co-author of more than 400 scientific publications. Dr. Fehske is a member of the steering committee the Erlangen National High-Performance Computing Center.

Gerhard Wellein is a Professor for High Performance Computing at the Department for Computer Science of the Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) and holds a Ph.D. in theoretical physics from the University of Bayreuth. From 2015 to 2017, he was also a guest lecturer at the Faculty of Informatics at the Università della Svizzera italiana (USI) in Lugano. Since 2024, he has been a Visiting Professor for HPC at the Delft Institute of Applied Mathematics at the Delft University of Technology. He is the director of the Erlangen National Center for High Performance Computing (NHR@FAU) and a member of the board of directors of the German NHR Alliance, which coordinates the national HPC Tier-2 infrastructure at German universities.