A Novel Cache-Blocked MPI-parallel Matrix Power Kernel:

Application to Finite-Volume Methods in Cardiac Arrhythmia Simulations

Master-Arbeit

zur Erlangung des Grades

Master of Science (M.Sc.)

im Studiengang Computational and Applied Mathematics

am Department Mathematik der
Friedrich-Alexander-Universität Erlangen-Nürnberg

vorgelegt am 09.03.23

von **Dane Lacey**

Betreuer: Prof. Dr. Gerhard Wellein
und Prof. Dr. Michael Stingl

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG
NATURWISSENSCHAFTLICHE
FAKULTÄT

Department
MATHEMATIK

Acknowledgements

I would like to take this opportunity to thank those who have helped encourage and support me throughout my writing of this thesis, and my time here in Germany.

Thank you to all the NHR@FAU staff who've been so welcoming to me first as a student research assistant in April 2022, then as a master's thesis student in October 2022. Particularly, I would like to thank the following people. Christie Alappat, whose work this thesis is centered around. Christie spent several hours a week patiently helping me, even though he is writing his own doctoral dissertation. I am very grateful for the mentorship, and for the many helpful debugging sessions. Prof. Dr. Gerhard Wellein, thank you for introducing me to the wide world of high performance computing through your lecture "Programming Techniques for Supercomputers". I also want to say thank you for all the letters of recommendation you've written for me over the past year, and the opportunities they have afforded me. Dr. Georg Hager, thank you for the opportunity to volunteer at ISC High Performance both remotely in 2021 and in-person in 2022. The tutorials and courses that you helped arrange were important in my development as a student research assistant. Thank you to my office mate Gonzalo Pinzon Waltero for the camaraderie and commiseration. Best of luck to you with your thesis! Also to Rasa Mabande and Kerstin Brandl for helping me navigate the bureaucracies that come with employment in Germany. Christie, Prof. Wellein, and Dr. Hager all provided insights and feedback on this thesis, for which I am grateful.

There are others outside of NHR@FAU that I would like to thank as well. Prof. Johannes Langguth at Simula Research Laboratory, thank you for the time you spent with me answering questions and helping me understand LYNX. Thank you to the FAU Scholarship Committee for the financial support through the Abschlussstipendium der FAU Wintersemester 2022/23 scholarship. To the many influential teachers that I've been lucky enough to meet over the years, thank you. From Salt Lake Community College, I would like to thank Kyle Costello for igniting my passion for science, and Robert Morelli for the introduction to rigorous mathematics, formal logic, and LaTeX. From The University of Utah, thank you to both Bronson Lim and Don Tucker for their influential Analysis lectures.

To my dear wife Rylee, thank you so much for your support and patience throughout this entire process. Between the Covid-19 pandemic and relocating to a different country in order for me to study, there were some stressful days we've shared over the past two and a half years. But also many, many good days. I deeply appreciate and value your friendship, love, and devotion.

To my mom and dad, Kimberly and Charles Lacey, thank you so much for your continued support of my education and for always believing in me. Being able to catch up with you both on the phone is something I looked forward to every week, and I'll always warmly remember your visits.

Contents

1 Introduction

It can be surprising how often the same patterns show up in numerical algorithms, regardless of the domain or application from which they originated. These patterns – which we refer to as "kernels" – can be considered the building blocks of said numerical algorithms. The reason is that algorithms inherit their performance characteristics from the kernels from which they are comprised [13]. Having efficient implementations of kernels is a high priority for many people, and is an active area of research.

One such common pattern is multiplying a matrix A with a vector x, i.e. $y \leftarrow Ax$. Applications in engineering, such as solving systems of differential equations with the Finite Element Method (FEM), generate and make use of matrices with hundreds of millions to billions of rows. These matrices typically consist of mostly zeros with only a handful of non-zero elements per row, so we call these matrices "sparse". The vector x tends to have mostly non-zero elements, and so we call it "dense". We call this kernel, the Sparse Matrix Vector Multiplication kernel (SpMV). Although we greatly exploit the sparsity of these matrices, they are usually still too large to "fit" into cache. This leads to a serious problem, as we now need to "load" this large sparse matrix from main memory for every single SpMV. If our numerical algorithm needs many SpMV iterations, which is often the case, this becomes a serious problem.

The Matrix Power Kernel (MPK), a kernel that computes $y \leftarrow A^p x$ for some power p is traditionally implemented as a series of p many repeated SpMV invocations (TRAD). We focus only on square matrices, that is $A \in \mathbb{R}^{n \times n}$. Although, we cannot represent all numbers in $\mathbb{R}$ on a computer, which is necessarily finite. So, the matrix A should be understood as being representable by floating point numbers when in the computing context. Just like SpMV, the utility of MPK can be seen in many diverse numerical algorithms. One typical application of this kernel is in Krylov subspace methods, which are used to find approximate solutions to high-dimensional problems in linear algebra. Since they make use of the linear subspace spanned by the first r power of A, efficiently computing $y \leftarrow A^p x$ for all (or applicable) $1 \leq p \leq r$ is of central importance [13], [27]. Other typical examples are exponential time integration [29] and polynomial preconditioning [20].

The Recursive Algebraic Coloring Engine (RACE), a library originally written for graph coloring and the efficient computation of SpMV for symmetric matrices [1], can also be used to efficiently compute MPK with cache blocking in the shared memory setting with an algorithm called "Level-Blocked Matrix Power Kernel" (LB-MPK), [2]. The contribution of this thesis is the efficient – yet flexible – implementation of LB-MPK in the distributed memory setting, based on the cache-blocking capabilities of RACE. We call this final algorithm the "Distributed Level-Blocked Matrix Power Kernel" (DLB-MPK).

For related works, we refer the reader to the following paper on "diamond blocking" for the MPK in the distributed setting [32]. In the shared memory setting, the problem of redundant computations for cache-blocking in the MPK is first addressed in [22]. Numerous references to other cache-blocking MPK works can be found in [2].

2 Background

In order to discuss the contributions of this work, there is background material to cover first. Namely, the parallel execution model we are working with, the kernels we will be focusing on, the basics of performance modeling with the SpMV kernel, and some fundamental concepts in numerical linear algebra.

Taking the computational point of view: The term "kernel" can mean very different things, even within the same field. Here, we use the term kernel to mean a small building block of an algorithm.

As mentioned before, kernels are at a sweet spot in understanding the performance of an algorithm, as full algorithms typically inherit the performance properties of the kernels they are comprised of. Where to distinguish separations in this complexity hierarchy is somewhat subjective. The way our kernels are defined here is done in a manner that best elucidates the way they affect performance.

Remark. The term "vector" and "array" are used somewhat interchangeably in this thesis. Typically, we say "vector" when discussing in the mathematical/theoretical context, and "array" when discussing something regarding an implementation.

2.1 Parallel Execution Model

We need an agreed-upon context in which we are working. Throughout this thesis, we will distinguish between two broad contexts in which computations take place.

The shared-memory parallel computer model, which we refer to as the shared memory context, shared memory setting, or just "shared setting". It is in this context that the Recursive Algebraic Coloring Engine was originally designed to operate.

A number of workers, or "processors", all work concurrently together on a shared region of data, or "address space", which is accessible to all processors. For illustrative purposes, and clarity, this shared address space will just be the main memory. Additionally, each processor has its own, private, "fast to access" address space called its "cache". There exist "shared" cache, but in an effort to keep our parallel execution model simple, this will not be discussed until Section 4. As a first step to establishing our parallel execution model, we make the following simplifying assumptions for the shared setting:

- We do not distinguish between different cache levels or types.
- Data in the cache is only a copy of what is present in main memory.
- In our case, we are exclusively focusing on cache-based multicore Central Processing Unit (CPU) based computer architectures. So for us, the workers/processors are just the cores of the CPU.

- The time it takes for a processor to access cache is "short", and the time to access main memory is "long". The details are not important for this thesis.

Remark. Main memory, or "primary memory", is typically implemented as Dynamic Random Access Memory (DRAM), while cache is typically implemented as Static Random Access Memory (SRAM). As is the usual case, we assume both are "volatile", i.e. data is lost when power is removed. This puts main memory and cache in contrast with a non-volatile "secondary-memory" storage device, such as a hard disk or solid-state drive.

This model is generally easier to conceptualize and program for. Work is typically parallelized with the use of in-code `#pragma` commands and details of work-distribution are taken care of implicitly, i.e. "under the hood". The most common API for these kinds of shared setting parallelizations, and what is used by RACE is OpenMP [24]. But there are inherent limitations on the size of problems you can efficiently solve with such a shared address space. Therefore, the shared setting is not our focus in this thesis, and only pertinent details are touched on. The interested reader should have a look at [1] and [2] for shared setting parallelizations, data dependency fulfillment, and optimizations for MPK that we will be taking for granted in this work.

Remark. Multicore CPUs have become omnipresent in the last decade. This architecture is found almost anywhere; from personal computers to smartphones and gaming consoles, to IT and industrial applications like virtual machines and databases. There are numerous interesting reasons for this trend, both commercial and technical, which are outside the scope of this work, see [12].

The distributed-memory parallel execution model, similarly just called "distributed context" or "distributed setting", can be conceptualized as a collection of shared setting computer models, connected by some "network".

Here, the entire address space is no longer visible to every processor but split up over multiple logically disjoint "processes" connected by a network, each "process" being its own shared memory context. To avoid confusion in the terminology between the CPU "processor" and the logically disjoint network "process", we will henceforth refer to the processes on the network as "MPI process". The Message Passing Interface (MPI) is the most common open library standard for distributed memory parallelization, and what we will be using for our work [21]. When referring to work being carried out within a specific MPI process, we will say this work is "MPI process local".

The benefit is that in the distributed memory context, we could theoretically scale our problem size to be as large as we want, and combine the computational resources (main memory, processors, etc.) from any number of machines. The downside is that we now need to concern ourselves with explicit communication of messages over a network.

These "over-network" communications have a very high cost when compared to the "under the hood" data movements that take place in the shared memory context. See [12] for more details.

Additionally, having to segment this work in an even manner and communicate data manually is – in general – more difficult to program for when compared to the shared memory setting. As we will see, one of the biggest advantages of DLB-MPK is that we do not require any *more* MPI communication than the traditional distributed MPK implementation.

We similarly make the following simplifying assumptions for the distributed setting:

- For consistency, the term "communication" will refer only to data movement over a network via MPI.
- Details about communication are abstracted to maintain an appropriate scope.
- Any MPI-process *can* communicate with any other, or none at all.
- We have no apriori knowledge of the network topology.

Remark. Technically, both our shared and distributed setting fall under the MIMD (Multiple Instructions, Multiple Data) classification of parallel computers [10], since there are many levels of parallelism on modern-day computing systems. But in an attempt to keep the two settings separate, we will not be using this terminology.

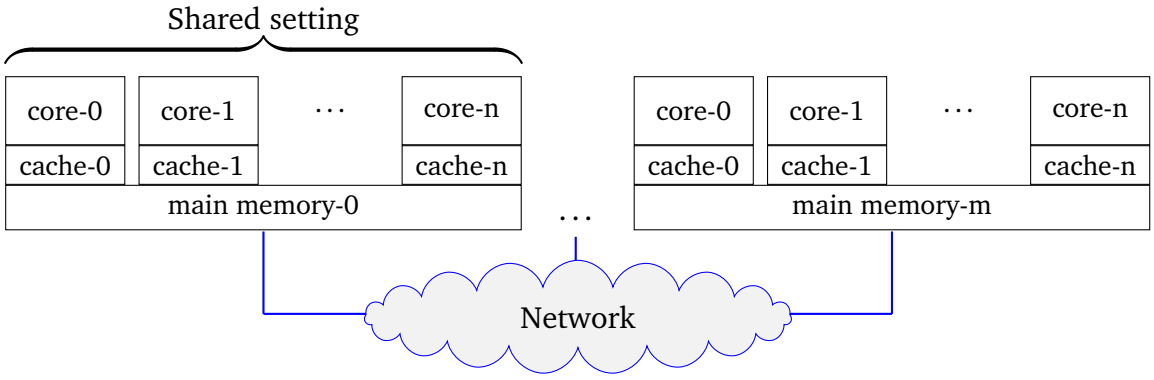


Figure 1: Distributed setting: m-many n-core CPUs connected by a network

Figure 1 further illustrates how our distributed setting is a collection of shared memory contexts (as we've defined them), connected by a network. For the remainder of this thesis, dark red is used to draw attention, while dark blue is used to signify communication over a network.

What is depicted in Figure 1 is a collection of Unified Memory Access (UMA) computers over a network, because this is the easiest to conceptualize. It should be noted that HPC applications utilizing DLB-MPK are probably better suited for the more advanced cache-coherent Non-Uniform Memory Access (ccNUMA) computers used in Section 4.

RACE, as well as the contributions of this thesis, are implemented in the C and C++ programming languages (specifically C++ 14). Those wishing for a deeper dive into

shared vs. distributed memory models, parallel computing, cache hierarchy, performance modeling, computer architecture, or anything else HPC are encouraged to look at [12].

The rough outline of the rest of the thesis is as follows: Beginning in the shared setting, we will continue the discussion on SpMV and MPK, paying special attention to sparsity and performance characteristics on modern computers. The useful notion of "permutations" is then discussed, in particular "symmetric permutations". We then continue the discussion of SpMV and MPK in the distributed setting, and how certain complications are simplified by taking the graph-theoretic point of view.

With the background material covered, we then start discussing the relevant parts of RACE: what it does, how it operates, and how to best make use of it. The derivation of the main contribution, DLB-MPK, comes afterwards. Essentially, DLB-MPK is centered around three main phases, after a pre-processing phase earlier in RACE. First, we aim to fulfill the supporting data dependencies. Then, after this set-up phase, we execute LB-MPK locally on each MPI process. Lastly, we need to "clean up" the remaining computations that were not done by LB-MPK.

In the Results section, we examine – through a few different lenses – the performance of DLB-MPK on a wide variety of matrices from the Suite Sparse Matrix Collection [9]. In Section 5, we give a showcase example of DLB-MPK in the Cardiac Arrhythmia Simulation code LYNX [18], which simulates the electrical activity in a patient's heart. Lastly, the thesis is summarized in Section 6, as well as the list of benchmark matrices used.

2.2 Sparse Matrix Vector Multiplication Kernel

As mentioned before, the multiplication of a matrix with a vector is a fundamental operation. Not only in mathematics, but also in many computational algorithms and domains. For a given vector $n \times 1$ vector x and $m \times n$ matrix A, the computation of the $m \times 1$ vector y by

$$y \leftarrow Ax$$

works by multiplying the transposed rows of A to the column vector x, and summing the corresponding entries.
Many statements made in this thesis also hold for rectangular matrices, but as already mentioned in Section 1, we restrict our view to the case where $m := n$. In theory, this is a restrictive yet necessary assumption, as square matrices are required for repeated matrix vector multiplications. In practice this is not a very strict assumption, as the most common matrices resulting from practical problems are square and symmetric (technically, "Hermitian") [26].

Example 2.1 (Matrix Vector Multiplication)**.** Even though this is a fundamental operation, and is likely known to the reader, it will be extremely helpful to have a concrete example of matrix vector multiplication to explain concepts in later sections. Take the following square matrix $A \in \mathbb{R}^{8\times 8}$ (which we will refer to in later sections as our "toy matrix") and dense vector $x \in \mathbb{R}^{8\times 1}$. The empty spaces denote 0 values. In the manner

of standard matrix vector multiplication, we move through the matrix row-wise, from row index 0 to 7 in this case, and multiply each row element with the corresponding right-hand side column vector x element. Both row and column indices are indicated in grey. Notice how the column index of the A element dictates to which element of x it is paired with for the product:

$$Ax = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{array}{c} \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{bmatrix} 1 & & & 1 & & & & \\ & 3 & & & 2 & & & \\ & 2 & & & & & & \\ & & & 4 & & 7 & & \\ & & & & & 2 & & \\ & & & & & 1 & & \\ & & 8 & & & & 2 & \\ & & & 9 & & & & 3 \end{bmatrix} \end{array} * \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{bmatrix} 1 \\ 0 \\ 6 \\ 4 \\ 4 \\ 3 \\ 2 \\ 0 \end{bmatrix} \rightarrow \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array} \begin{bmatrix} 5 \\ 8 \\ 0 \\ 37 \\ 6 \\ 3 \\ 52 \\ 36 \end{bmatrix} = y$$

This operation can also be thought of as a "dot product" between the rows of A, and the vector x. We show an explicit example for row index 3.

$$0(1) + 0(0) + 0(6) + 4(4) + 0(4) + 7(3) + 0(2) + 0(0) \rightarrow 37$$

There is no standard definition of when a matrix is "sparse" because it depends on the application from which the matrix is generated. Typically, when there are "many more" zeros than non-zeros, we say the matrix is sparse. A good, common-sense definition is "...a matrix can be termed sparse whenever special techniques can be utilized to take advantage of the large number of zero elements and their locations." [26]. In addition to just the number of zeros versus non-zeros, this non-zero pattern can have a structure to it (as is often the case with real-world applications). This introduces another layer of ambiguity in the definition of sparsity. For example, would you say the matrix in Example 2.1 is sparse? Is it structured or unstructured? It appears that there are quite a few zeros, and there may be some rough structure to it, but that distinction is subjective.

Sparsity opens a wide range of opportunities for optimizations in storage and processing. In the 1960s, engineers working with electrical networks were the first to realize that the sparsity present in their linear systems could be exploited to process much larger systems [26]. But there also exist serious challenges.

In the absence of any cache-blocking strategy, matrices need to be loaded from main memory every time they are processed in an algorithm. These blocking techniques are very straightforward with dense matrices, as one can just cut the matrix into tiles small enough to be able to fit into cache. For sparse matrices, these strategies will not work well. What is needed is an algebraic blocking technique that works on the "graph" of A (as discussed in Section 2.5). Even with such a blocking technique, algorithms that make use of sparse matrices very quickly saturate the main memory bandwidth. This is such a big problem because the speed at which CPUs can process data has been growing much faster than the speed at which this data can be retrieved from

main memory. This phenomenon has been termed the "DRAM gap" [12]. This is why, when dealing with sparse matrices, optimizations typically aim to minimize this data traffic to the main memory. Conversely, these same optimizations are senseless for algorithms that make use of dense matrices.

There are many formats with which one may store a sparse matrix in memory, all of which have their pros and cons. Coordinate format (COO) is the most intuitive, storing a sparse matrix as three arrays: the row indices, the column indices, and the values. Some formats, such as ELLPACK or Sliced-ELLPACK [23] allow one to make use of the wide SIMD [10] registers on one's CPU, and pack multiple data elements to be processed in a single instruction. ELLPACK is especially important for numerical algorithms using Graphics Processing Units (GPUs). The SELL-C-σ [17] format builds on this idea of wide register usage with two tunable parameters, C and σ. The former parameter denotes the "chunk height" which should correspond to a multiple of the SIMD width of the target register, and the latter is a sorting scope that reduces the overhead of the format as compared to Sliced-ELLPACK. This format also enables more consistent code across homogeneous hardware, by the use of these two parameters. One very popular format, and the format we've chosen as the focus of this work, is Compressed Row Storage (CRS) [5]. This format was chosen, as it has well-known corresponding kernels, is cache friendly, and has been shown to perform well on cache-based multicore CPUs, [33], [5], [11]. More details of how SpMV performs with the CRS format, especially in comparison to SELL-C-σ format, can be seen in [17].

As with all sparse storage formats, with CRS we make use of the fact that most elements of the matrix are zero, and only store the non-zero entries. We scan the matrix, row-by-row, and store these non-zeros values in the `val[]` array. The corresponding column indices are put in the `col[]` array. Finally, we need some way to keep track of where the rows end. This data is stored in the `rowPtr[]` array and is the index of the element in the `val[]` (or `col[]`) array which lies on the following row. Throughout this thesis, we will assume that values are stored as 8 byte double precision floating point numbers, and column and row pointer data are stored as 4 byte integers. In practice, there are many choices for storing the values (i.e. 2 byte half precision, 4 byte single precision, etc.). The choice of datatype depends on the level of precision required by the application. For storing column and row pointer data, 8 byte "long" integers are also a common choice.

Example 2.2. We can represent the matrix A of Example 2.1 in CRS with three arrays:

```
val = [1,1,3,2,2,4,7,2,1,8,2,9,3]
col = [0,3,1,4,1,3,5,5,5,2,6,3,7]
 rowPtr = [0,2,4,5,7,8,9,11,13]
```

Yet, the COO format represents A with the arrays:

```
val = [1,1,3,2,2,4,7,2,1,8,2,9,3]
col = [0,3,1,4,1,3,5,5,5,2,6,3,7]
row = [0,0,1,1,2,3,3,4,5,6,6,7,7]
```

Clearly, COO requires more storage space when compared to CRS. This larger storage overhead will only grow with the number of rows of matrix A. For more details about the creation of the CRS arrays, as well as many other formats, see [25].

Define N_r as the number of rows and N_{nz} as the number of non-zeros of matrix A. One implementation of SpMV making use of the CRS matrix storage format is given in Algorithm 1.

Algorithm 1: Sparse Matrix Vector Multiplication Kernel

```
Input : int sRow, eRow;                          // starting and ending row
        double in[N_r];                            // dense RHS input array
        double val[N_nz];                                    // CRS arrays
        int col[N_nz], rowPtr[N_r + 1];
Output: double out[sRow : eRow];
1 for row ← sRow to eRow do
2 |   double tmp ← 0;
3 |   for idx ← rowPtr[row] to rowPtr[row + 1] − 1 do
4 |   |   tmp ← tmp + val[idx]*in[col[idx]];
5 |   end
6 |   out[row] ← tmp;
7 end
```

We will need the flexibility of being able to choose the "start" and "end" rows later. But for our purposes now, think of `sRow` and `eRow` as 0 and N_r respectively.

2.3 Matrix Power Kernel

As stated in Section 1, the Matrix Power Kernel (MPK) is a kernel that computes $y \leftarrow A^p x$ for some power p. It is traditionally implemented as a series of p many repeated SpMV invocations (TRAD).

Depending on the use case, there are two typical implementations:

1. MPK returns as output the successive powers of $A^p x$ stored column-wise in a two dimensional array $y[N_r, p_m]$ where $y[:, p]$ stores $A^p x$ for $1 \leq p \leq p_m$.
2. MPK returns as output only the highest power $A^p x$ in a one-dimensional array $y[N_r]$.

We choose the former as our "Standard MPK" implementation. The pseudocode is given in Algorithm 2, where line 3 calls the routine defined in Algorithm 1 as SpMV.

Many of the performance characteristics of MPK are inherited from SpMV [1].

Algorithm 2: Matrix Power Kernel

Input : `double x[`N_r`];` `// dense RHS input array`
`double val[`N_{nz}`];` `// CRS arrays`
`int col[`N_{nz}`], rowPtr[`N_r+1`];`
`int` p_m`;` `// highest power to compute` $A^{p_m}x$
Output: `double y[`N_r, p_m`];` `// 2D results array,` $y[N_r,k]=A^k x$

```
1 y[:,0] ← x;
2 for p ← 1 to p_m do
3 |   y[:,p] ← SpMV(0, N_r, y[:,p−1], val, col, rowPtr);
4 end
```

2.4 Roofline Model

It is well known that SpMV performs relatively poorly on modern cache-based multicore processors. The performance limiting factor here is typically loading data from main memory, i.e. "accessing" the data in main memory. As mentioned in the previous section, the center of the issue is that CPUs have become incredibly fast and efficient, and in the case of SpMV, are starved for data to process. This quote from [12] states the problem well:

> "Since many applications in science and engineering consist of loop-based code that moves large amounts of data in and out of the CPU, on-chip resources tend to be underutilized and performance is limited only by the relatively slow data paths to memory or even disks."

The problem boils down to "bandwidth saturation" of the data lanes that run from the CPU registers to main memory (or to the cache, if that is where the data resides). By saturation, we mean that throughput is the same as bandwidth, and it is not physically possible to transfer more bytes per second than are already being transferred. In other words, the kernel exhibits a high "code balance" (or equivalently, a low "arithmetic intensity"). These two metrics are inversely related by $I[F/B] = B_C^{-1}[B/F]$. This problem is stated succinctly, by saying SpMV is "main memory-bound".

Being that SpMV is such a universally important kernel for many numerical applications, suffice it to say that there are very good reasons why data access optimization for memory-bound kernels is such an active area of research.

Even when we restrict our attention to matrices in CRS format, the details that come with understanding and accurately modeling the performance of data accesses from SpMV could easily fill a thesis on its own. Therefore, we only cover the basics and what is directly relevant to our work. Particularly, we focus on modeling by means of the (naive) Roofline Model [34]:

$$P[F/s] = \min\left(P_{peak}[F/s], \frac{b_s[B/s]}{B_C[B/F]}\right)$$

In words, the number of floating point operations per second (i.e. addition, subtraction, multiplication, but *not* division) is the minimum of two quantities: P_{peak} which is entirely determined by the hardware and b_s/B_C where b_s is the memory bandwidth (also determined by hardware) and B_C the code balance (depends entirely on the kernel).

This simple yet powerful model answers the question, "what kind of performance can we expect?" in the shared context, with "lightspeed" assumptions. On a single modern CPU, the peak performance can be calculated by:

$$P_{peak} = n_{cores} * n_{super}^{FP} * n_{FMA} * n_{SIMD} * f$$

where n_{cores} is the number of cores, n_{super}^{FP} is the "superscalarity" factor, n_{FMA} is the number of "fused multiply-add" execution ports, n_{SIMD} is the "SIMD factor", and f is the selected frequency in "cycles per second". For more details, see [12].

Remark. For this thesis, we understand performance in terms of floating point operations per second (Flops). In general, one can choose any metric for measuring performance, as long as it makes sense in the context of the application. Flops may not be a reliable metric in all settings, but for SpMV it works well.

The bandwidth to memory b_s can be theoretically estimated, based on a hardware data fact sheet for example. But due to many complicating CPU internal factors, it should just be determined empirically, for example employing a "Copy" or "Load" bandwidth benchmark [17].

We need to establish the "code balance" B_C for our kernel in Algorithm 1, assuming data comes from main memory. The code balance is a metric that, as the reader will notice by the units, describes how many bytes one must move from main memory per floating point operation. Assuming we are using arrays that contain 4 byte integers for `rowPtr[]` and `col[]`, and an array that contains 8 byte double precision floating point numbers for `val[]`. Let LD denote a "load" instruction from main memory, and ST a "store" instruction to main memory. Both LD and ST incur data traffic, 4 bytes for integers and 8 bytes from doubles, denoted "V". Let N_{nzr} denote the average number of non-zero elements per row of the matrix. The focus is at line 4 in our SpMV implementation for CRS format in Algorithm 1:

$$B_C[B/F] = \frac{(V_{LHS} + V_{RHS} + V_{mat} + V_{rowPtr})[B]}{2[F]} = \frac{12 + 8\alpha + 20/N_{nzr}}{2}\left[\frac{B}{F}\right]$$

where data volumes are :

$$V_{LHS} \leftarrow 1LD + 1ST = \frac{(1(8) + 1(8))[B]}{N_{nzr}} = \frac{16}{N_{nzr}}[B]$$

to update a single element of y on average (the reason for both the LD and ST instruction is due to a write-allocate transfer),

$$V_{RHS} \leftarrow 1LD * \alpha \leftarrow 1(8)\alpha[B] = 8\alpha[B]$$

to load a single element of x, and

$$V_{mat} \leftarrow 2LD = (1(8) + 1(4))[B] = 12[B]$$

to load the respective `val[]` and `col[]` elements. The `rowPtr[]` must also be loaded from memory, incurring a data traffic of

$$V_{rowPtr} \leftarrow 1LD = \frac{(1(4))[B]}{N_{nzr}} = \frac{4}{N_{nzr}}[B].$$

The 2 flops come from the single addition and single multiplication [17]. The reason for the parameter α is to capture the interplay between matrix structure and cache hierarchy. The minimum $\alpha = N_{nzr}^{-1}$ implies we only load our RHS vector x from main memory once, for the initial compulsive load. The rest of the loads would then come from cache. This is possible if our "matrix bandwidth" – the maximum distance from the leftmost column index to the rightmost column index in any given row of the matrix (not to be confused with network/memory bandwidth) – is small, and our cache is large enough. The B_C, in that case, would be $B_C = (6 + 12/N_{nzr})[B/F]$, and the limiting case would then be $B_C = 6[B/F]$ as $N_{nzr} \to \infty$, described in [1]. To further complicate the modeling of α, the access pattern on x is often irregular [2]. This problem is addressed in Section 2.5.

What the Roofline Model tells us to expect in performance for SpMV is:

$$P = \min\left(n_{cores} * n_{super}^{FP} * n_{FMA} * n_{SIMD} * f, \frac{2 * b_s}{12 + 8\alpha + 20/N_{nzr}}\right)$$

But since we've established that SpMV is main memory-bound, we are concerned only with the expression on the right, not the P_{peak}:

$$P = \frac{2 * b_s}{12 + 8\alpha + 20/N_{nzr}} = \frac{b_s}{6 + 4\alpha + 10/N_{nzr}} \tag{1}$$

We will return to this model in Section 4 after establishing a testbed machine in order to describe observed performance.

Remark. There is some research in a similar vein as the Roofline Model, which also encapsulates network performance in the modeling. Since our contributions are in the distributed setting, one would think that this "Ridgeline Model" makes more sense to use. But seeing as how network communication is not our bottleneck, we leave the Ridgeline Model pre-print citation here for interested readers [7].

2.5 Permutations and Correspondence with Graphs

As mentioned in Section 2.4, the access pattern on x for our SpMV implementation is often irregular. This means that – for a given row i of A, $0 \leq i \leq N_r$ – the elements $a_{i,j}$ in that row could have any column index $0 \leq j \leq N_r$.

Recall, the column index of a given element in A dictates to which element of x it pairs up with for the dot product. Therefore, it is the sparsity pattern of A which determines how scattered these x accesses are.

Before we move on to methods that improve the sparsity pattern of matrices, and what it even means to "improve" a sparsity pattern, we should understand why scattered accesses are such a bad thing.

The central reason is poor spatial locality. Think about main memory as one, long array. Data that is required by the CPU for a given calculation is pulled from main memory in little, contiguous bundles called "cache lines" (CL). Since the elements of x are all stored contiguously, if the CPU asks for the data element at $x[100]$ for a calculation, the operating system would grab all elements from $x[95]$ - $x[102]$ (or a similar range of elements containing $x[100]$) and put them into cache. We are assuming here that x is an array of 8 byte doubles, and a CL is 64 bytes. See Figure 2.

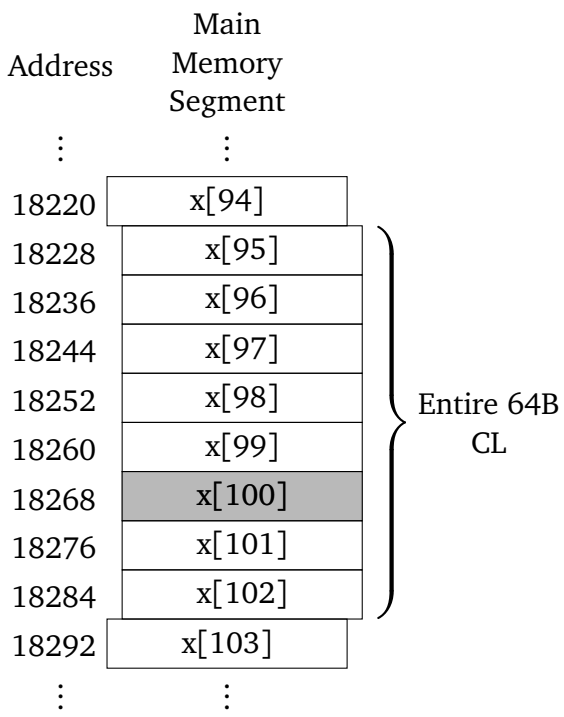


Figure 2: Loading a CL from main memory

This is a very nice thing for the operating system to do for us, because if one accesses an array at an index "100", typically there is a very good chance that the next index accessed will be "101". With the array element at index 101 already in cache, we can access it without paying the latency cost of having to request it all the way from main memory (and increase the strain on an already saturated data lane, in the case of SpMV). In the language of Section 2.4, needing to load our x elements from main memory every time increases the code balance, quantified by α. So, performance will take a big hit if the elements in x are accessed "far enough" away from one another, and it will make the cache essentially unusable [17].

This is extremely important for the performance of SpMV (and by extension MPK), as matrices with a very "bad" sparsity pattern will lead to these very scattered accesses of x.

How can we remedy this problem, and make a bad sparsity pattern "better"? We permute the matrix, that is, reorder the rows and columns in some clever way as to improve the sparsity pattern. This "improvement" is typically achieved by decreasing the bandwidth of the matrix, meaning the columns of A with non-zero elements get closer together. Hence, the elements of x that we access, based on the needs of a given row of A for the SpMV, become closer together. That is, we make better use of

spatial locality. To quote Saad [26], "...reordering rows and columns is one of the most important ingredients used in parallel implementations of both direct and iterative solution techniques." In the language of linear algebra, a "permutation" is a matrix multiplication by an "elementary matrix". An elementary matrix is one which differs from the identity matrix by a single "elementary row operation". While elementary row operations can achieve a few different things: row switching, row multiplication, or row addition; we will only need to make use of row switching.

Example 2.3. We take the toy matrix A from Example 2.1, and $E \in \{0,1\}^{8\times 8}$ to be the elementary matrix formed from interchanging row indices 0 and 5 of the identity matrix $I \in \{0,1\}^{8\times 8}$:

$$I = \begin{bmatrix} 1 & & & & & & & \\ & 1 & & & & & & \\ & & 1 & & & & & \\ & & & 1 & & & & \\ & & & & 1 & & & \\ & & & & & 1 & & \\ & & & & & & 1 & \\ & & & & & & & 1 \end{bmatrix} \quad E = \begin{bmatrix} & & & & & 1 & & \\ & 1 & & & & & & \\ & & 1 & & & & & \\ & & & 1 & & & & \\ & & & & 1 & & & \\ 1 & & & & & & & \\ & & & & & & 1 & \\ & & & & & & & 1 \end{bmatrix}$$

Here, we view the elementary matrix as a permutation vector P, where the interchanged rows in E are indicated by interchanged vector elements in P:

$$E = \begin{bmatrix} & & & & & 1 & & \\ & 1 & & & & & & \\ & & 1 & & & & & \\ & & & 1 & & & & \\ & & & & 1 & & & \\ 1 & & & & & & & \\ & & & & & & 1 & \\ & & & & & & & 1 \end{bmatrix} \Rightarrow \quad P = \begin{bmatrix} 5 \\ 1 \\ 2 \\ 3 \\ 4 \\ 0 \\ 6 \\ 7 \end{bmatrix}$$

Just as we can collect any number of elementary row operations into a single elementary matrix with successive left (or right) matrix multiplications, we can collect any number of permutation vectors by successively applying each vector to one another via

$$P_b[P_a][i] = P_b[P_a[i]] \text{ for each } i \in P_a \ ,$$

which we will just write $P_b[P_a]$. For some given $k \in \mathbb{N}$, we are able to write:

$$E_k E_{k-1} \cdots E_2 E_1 = \begin{bmatrix} & & & & & & 1 & \\ & & & 1 & & & & \\ & & & & & & & 1 \\ & & 1 & & & & & \\ & & & & 1 & & & \\ & 1 & & & & & & \\ 1 & & & & & & & \\ & & & & & 1 & & \end{bmatrix} \Rightarrow \quad P = P_k[P_{k-1}[\cdots P_2[P_1]]\cdots] = \begin{bmatrix} 6 \\ 3 \\ 7 \\ 2 \\ 4 \\ 1 \\ 0 \\ 5 \end{bmatrix}$$

One could ask, "How would this change MPK"? Before seeing any benefit of the technique of permutations, one may start to worry that this could significantly alter Algorithms 1 and 2, or lead to some other headache and decide not to bother with them at all. But worry not, nothing changes concerning our kernels, there are just a couple of bookkeeping details to keep track of. The first comes with the fact that, since the rows of our matrix are permuted, the LHS results vector $y[:,p]$ would also come out of the SpMV with the same permutation. We illustrate this idea in Example 2.4.

Example 2.4. Take the toy matrix A and RHS input vector from Example 2.1. Permute the rows of A by the left matrix multiplication EA, where E is the elementary matrix from Example 2.3. We proceed with two powers of MPK. We take a particular row to show exactly where the problem lies:

Step 1: SpMV: $Ay[:,0] \rightarrow y[:,1]$

$$\begin{array}{c} \\ 5 \\ 1 \\ 2 \\ 3 \\ 4 \\ 0 \\ 6 \\ 7 \end{array}\!\!\begin{array}{c} \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{bmatrix} & & & & 1 & & & \\ & 3 & & & 2 & & & \\ & 2 & & & & & & \\ & & & 4 & & 7 & & \\ & & & & & 2 & & \\ 1 & & & 1 & & & & \\ & & 8 & & & & 2 & \\ & & & 9 & & & & 3 \end{bmatrix} \end{array} * \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \\ 6 \\ 7 \end{array}\!\!\begin{bmatrix} 1 \\ 0 \\ 6 \\ 4 \\ 4 \\ 3 \\ 2 \\ 0 \end{bmatrix} = \begin{array}{c} \\ 5 \\ 1 \\ 2 \\ 3 \\ 4 \\ 0 \\ 6 \\ 7 \end{array}\!\!\begin{bmatrix} 4 \\ 8 \\ 0 \\ 37 \\ 6 \\ 5 \\ 52 \\ 36 \end{bmatrix} \rightarrow y[:,1]$$

Step 2: SpMV: $Ay[:,1] \rightarrow y[:,2]$

$$\begin{array}{c} \\ 5 \\ 1 \\ 2 \\ 3 \\ 4 \\ 0 \\ 6 \\ 7 \end{array}\!\!\begin{array}{c} \begin{array}{cccccccc} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array} \\ \begin{bmatrix} & & & & 1 & & & \\ & 3 & & & 2 & & & \\ & 2 & & & & & & \\ & & & 4 & & 7 & & \\ & & & & & 2 & & \\ 1 & & & 1 & & & & \\ & & 8 & & & & 2 & \\ & & & 9 & & & & 3 \end{bmatrix} \end{array} * \begin{array}{c} \\ 5 \\ 1 \\ 2 \\ 3 \\ 4 \\ 0 \\ 6 \\ 7 \end{array}\!\!\begin{bmatrix} 4 \\ 8 \\ 0 \\ 37 \\ 6 \\ 5 \\ 52 \\ 36 \end{bmatrix} = \begin{array}{c} \\ 5 \\ 1 \\ 2 \\ ? \\ ? \\ ? \\ 6 \\ 7 \end{array}\!\!\begin{bmatrix} 6 \\ 36 \\ 16 \\ \times \\ \times \\ \times \\ 104 \\ 441 \end{bmatrix} \rightarrow y[:,2]$$

The first SpMV can execute as expected, even though the resulting vector comes out row-permuted. But on the second SpMV iteration (and all iterations thereafter), the incoming RHS input vector, $y[:,1]$ is *still* row-permuted. This will cause a mismatch, as the columns of the elements in A need to match with the rows of the elements in x.

More generally, for all steps $p > 1$, the MPK computing

$$y[:,p] = Ay[:,p-1]$$

will be incorrect, with this strategy.

The obvious solution to this problem is to, on step p, un-permute $y[:, p-1]$ before assigning this results vector to the corresponding column of the 2D results vector $y[N_r, p_m]$. The required "inverse" permutation vector P^{-1} can be constructed by merely exchanging the elements of P with their indices:

$$P^{-1}[P[i]] = i \text{ for each } i \in P \ .$$

So this solution, implemented for the same 3 steps above would look like this:

Step 1: Initialize: $x \rightarrow y[:, 0]$, Permute rows of A via. $A \leftarrow EA$

Step 2: SpMV: $P^{-1}[Ay[:, 0]] \rightarrow y[:, 1]$

Step 3: SpMV: $P^{-1}[Ay[:, 1]] \rightarrow y[:, 2]$

But, at the implementation level, there is a cost in performance one must pay to un-permute an array. This cost will have to be paid every single iteration of MPK after the initialization phase. Hence this solution, while intuitive, incurs unnecessary overhead. It is possible to only pay this price of un-permutation a single time, regardless of the number of SpMV or even MPK invocations. With the example above, it is clear that the root of the issue is in the order of the columns of EA. With this understanding, the workaround to this overhead becomes clear.

What we need is the concept of a "symmetric permutation". When one permutes symmetrically, the rows *and* columns of A are interchanged, both according to the permutation vector P. This can be done at initialization, before the SpMV iterations of the MPK. Then the SpMVs can take place purely in the "permuted space".

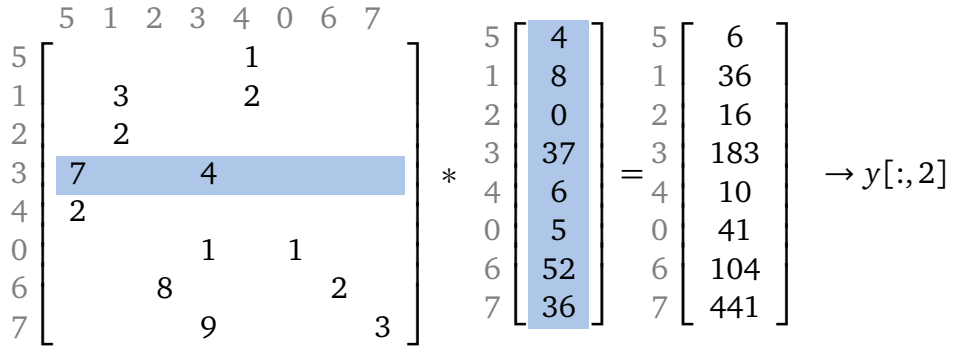
Example 2.5. The rows *and* the columns of the matrix, as well as the incoming RHS vector, are now permuted in the initialization phase by $A \leftarrow EAE$.

We will again focus on the same particular row from Example 2.4 to illustrate how the problem has been solved:

Step 1: SpMV: $Ay[:, 0] \rightarrow y[:, 1]$

$$\begin{array}{c} \\ 5 \\ 1 \\ 2 \\ 3 \\ 4 \\ 0 \\ 6 \\ 7 \end{array}\!\!\begin{array}{c} \begin{array}{cccccccc} 5 & 1 & 2 & 3 & 4 & 0 & 6 & 7 \end{array} \\ \begin{bmatrix} & & & & 1 & & & \\ & 3 & & & 2 & & & \\ & 2 & & & & & & \\ 7 & & & 4 & & & & \\ 2 & & & & & & & \\ & & & 1 & & 1 & & \\ & & 8 & & & & 2 & \\ & & & 9 & & & & 3 \end{bmatrix} \end{array} * \begin{array}{c} \\ 5 \\ 1 \\ 2 \\ 3 \\ 4 \\ 0 \\ 6 \\ 7 \end{array}\!\!\begin{bmatrix} 3 \\ 0 \\ 6 \\ 4 \\ 4 \\ 1 \\ 2 \\ 0 \end{bmatrix} = \begin{array}{c} \\ 5 \\ 1 \\ 2 \\ 3 \\ 4 \\ 0 \\ 6 \\ 7 \end{array}\!\!\begin{bmatrix} 4 \\ 8 \\ 0 \\ 37 \\ 6 \\ 5 \\ 52 \\ 36 \end{bmatrix} \rightarrow y[:, 1]$$

Step 2: SpMV: $Ay[:,1] \rightarrow y[:,2]$



Once all the required SpMVs have taken place, we then have the option to inversely permute the rows within the 2D results vector $y[N_r, p_m]$ to the "un-permuted space" one column at a time, or leave it in the permuted space, depending on the needs of the numerical library.

While this "optimization" may not seem like a sizable benefit now, consider the fact that MPK is just one of many kernels that a modern numerical library has at its disposal. When trying to solve a practical problem in science and engineering, one may use a solver from a numerical library that executes many kernels, after each of which we would have to pay the price of "un-permutation" if we didn't use symmetric permutation at initialization.

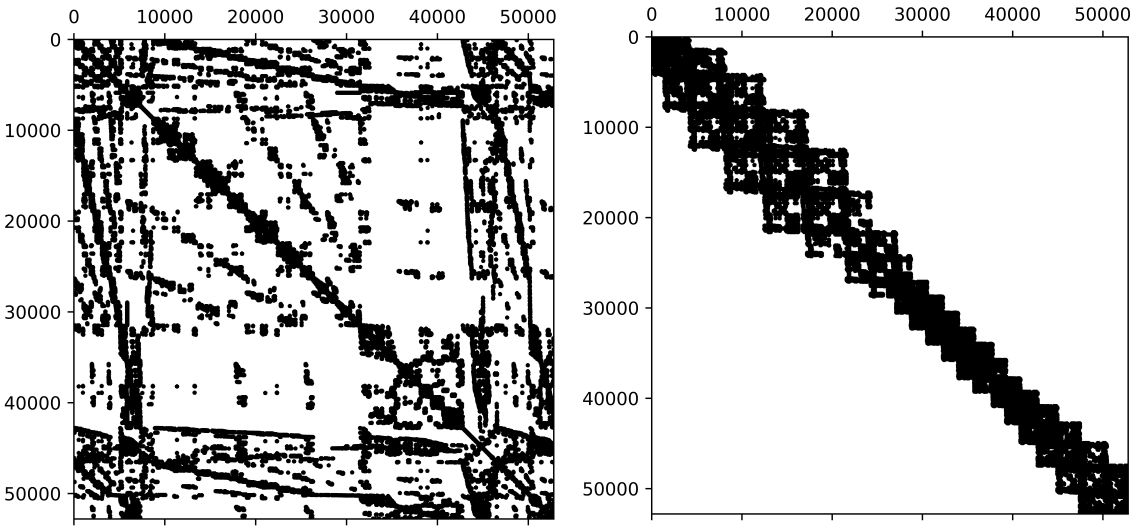
With symmetric permutation, we also gain the advantage of not having any extra steps in the kernel itself (e.g. if we only permuted the rows of the matrix A, we would need an inverse permutation step within the MPK kernel itself). That is, permutation of the rows and columns occurs only in the set-up phase of the host framework, and optionally after the main kernel(s) have been executed. Since the kernels themselves are not touched by symmetric permutation, we do not need to deal with permutations explicitly within the aforementioned kernels.

We return to the idea of permutations in Section 3.1. Without giving too much away now, we leave the reader with an example of how dramatically such a symmetric permutation scheme can reduce the bandwidth, and thus, improve the sparsity pattern of a matrix.

Here, we take a matrix `crankseg_1` from the Suite Sparse Matrix Collection [9] and perform a Breadth First Search (BFS) symmetric reordering of the rows and columns.

There exists a convenient correspondence between matrices and "graphs". Graphs are a useful and ubiquitous way to represent pairwise relations (edges) between objects (verities). This correspondence allows one to make use of linear algebra concepts in the context of graphs, and graph theoretic concepts in the context of linear algebra. We will make use of both directions of the correspondence throughout this work.

The definition of a graph varies from source to source, so we must define here exactly what we mean. The graph theoretic terminology is sourced from [6]. Instead of citing this source for every definition, we cite it now.



(a) Without Permutations (b) With Symmetric BFS Permutation

Figure 3: `crankseg_1` before and after permutation

Definition 2.1. Graph
A graph $G = (V, E, W)$ is a 3-tuple: an ordered set of vertices V, an ordered set of edges E, and an ordered set of weights W.

Definition 2.2. Vertices
A set of vertices

$$V = \{v_1, v_2, \ldots, v_n\}$$

is just a set of labels, representing distinct entities in a graph.

Given two vertices $u, v \in V$, if there exists an edge $e \in E$ that "connects" u and v, then we say vertices u and v are "adjacent". How this connecting edge e is represented in the set E depends on if we're discussing an "undirected" graph, or a "directed" graph:

- In the former case, there is no notion of the "direction" of the edge, and so e is represented as the (unordered) set $\{u, v\}$. Here, we say u and v are adjacent/connected.
- In the latter case, the edges are directed, and so e is instead represented as the (ordered) pair (u, v). We now must make the distinction in our verbiage, that v is adjacent/connected *to* u, but the adjacency doesn't necessarily go the other way

Definition 2.3 (Edges)**.** The set of edges $E \subseteq V \times V$ consists of ordered pairs of vertices. For vertices u and v, if $(u, v) \in E$, we say that v is adjacent to u.

Definition 2.4 (Weights)**.** The set of weights W is defined as

$$W = (w_k)$$

such that, if e is the k-th edge in E, then the corresponding weight w is the k-th entry in W.

Remark. For convenience, we allow loops, i.e. the case in which $u = v$. Notice that an undirected graph is just a special case of a directed graph, in which both $(u, v), (v, u) \in E \iff u, v \in V$.

More generally: for any matrix $A \in \mathbb{R}^{n \times n}$, there exists a correspondence with a weighted, directed graph $G = (V, E, W)$ such that:

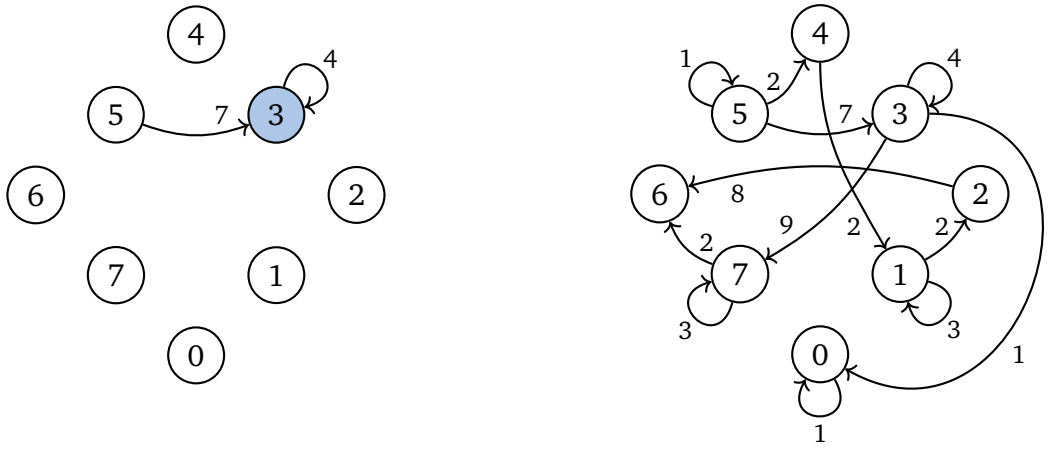
$$\begin{cases} A = (a_{i,j}) \in \mathbb{R}^{n \times n} \\ i, j \in \mathbb{N} \end{cases} \iff \begin{cases} G = (V, E, W) \\ i, j \in V = \{1, 2, \ldots, n\} \\ (i, j) \in E \iff a_{i,j} \neq 0 \\ W(i, j) = a_{i,j} \end{cases}$$

In this context, matrix A is typically called the "weighted adjacency matrix" of the graph.

One way in which we can construct the graph $G = (V, E, W)$ corresponding to the square matrix A is by the following steps:

1. For every row i of A, $1 \leq i \leq n$, declare a vertex of G labeled i
2. For each row i of A, scan the columns $1 \leq j \leq n$
3. If $a_{i,j} \neq 0$, connect vertex j *to* vertex i with corresponding weight $w = a_{i,j}$

Example 2.6. Following the above procedure, it can be said that $A \in \mathbb{R}^{8 \times 8}$ from Example 2.1 is the "weighted adjacency matrix" representing the graph in Figure 4. We illustrate row 3 as an explicit step.



(a) Only edges to vertex 3

(b) All edges included

Figure 4: Graph representation of our toy matrix

While not representative of a sparse matrix coming from a real application, this "toy graph" example gives us an instance to refer to later. Entire books are dedicated to exploring this correspondence, and many fruitful research projects are making use of software built around these ideas[1].

[1] https://graphblas.org/

2.6 SpMV and MPK in the Distributed Setting

Up to this point, we have been viewing SpMV and MPK through the lens of the shared memory setting. Wherein, we've been ignoring "data dependencies", i.e. when one worker (CPU core) is depending on the result of another worker, as is sometimes the case with OpenMP parallelizations. These are assumed to be taken care of. But data dependencies also exist in the distributed setting. When work is distributed across MPI-processes, it is frequently the case that one MPI process depends on the result of another. For the remainder of this thesis, the main theme will be fulfilling these distributed setting data dependencies.

What if now, we wanted to partition a matrix over two MPI processes? What additional complications arise, and what data dependencies need to be taken care of? We illustrate by example.

Example 2.7. Again, take the toy matrix A from Example 2.1. Assume A is segmented after row index 4 by some earlier routine. This segmentation should split the work evenly. The naive methods of splitting a matrix by rows or number of non-zeros are viable options, but more sophisticated matrix splitting (or "graph partitioning") methods exist, such as METIS (see Section 4.2). Having a fair balance of work between MPI processes is critical for performance, as performance in the distributed setting is often dictated by the slowest process. Hence, graph partitioning methods are an active area of research.

For our work, the details of the graph partitioning technique are not relevant. Suffice it to say, we segment our work somewhat evenly. Assume A now lies on two MPI-processes, as in Figure 5.

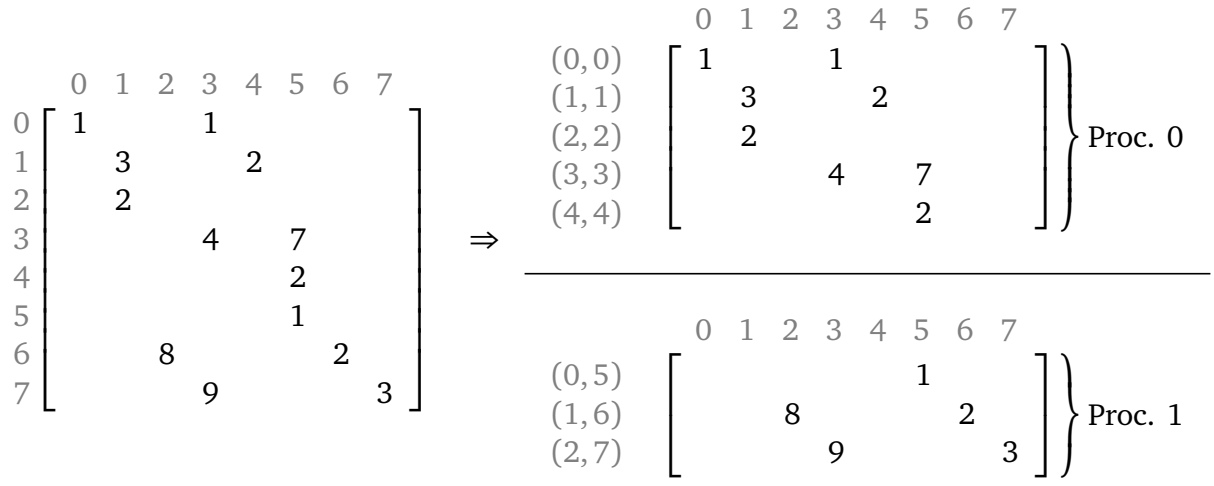


Figure 5: Global to local partitioning

It now makes sense to distinguish between "global" row indices, and process-"local" row indices. We associate each row with the pair:

(process-local row index, global row index)

so that each row index to have a unique identifier. This organization is useful if we try to do an SpMV with the distributed matrix A, and the process-local part of x. This, along with the column indices, is given in grey. We partition x, according to the same row index that we partition A. It is logical to not keep a full copy of the RHS vector x on every MPI process, since many of the entries will not be used on every process, and this incurs an unnecessary storage overhead. We can now, on each MPI process, compute a local $Ax \rightarrow y$ as in Figure 6.

$$\text{Proc. 0} \begin{cases} (0,0) \\ (1,1) \\ (2,2) \\ (3,3) \\ (4,4) \end{cases} \begin{bmatrix} 1 & & & 1 & & & & \\ & 3 & & & 2 & & & \\ & 2 & & & & & & \\ & & & 4 & & 7 & & \\ & & & & & 2 & & \end{bmatrix} * \begin{bmatrix} 1 \\ 0 \\ 6 \\ 4 \\ 4 \end{bmatrix} = \begin{bmatrix} 5 \\ 8 \\ 0 \\ \times \\ \times \end{bmatrix}$$

$$\text{Proc. 1} \begin{cases} (0,5) \\ (1,6) \\ (2,7) \end{cases} \begin{bmatrix} & & & & & 1 & & \\ & & 8 & & & & 2 & \\ & & & 9 & & & & 3 \end{bmatrix} * \begin{bmatrix} 3 \\ 2 \\ 0 \end{bmatrix} = \begin{bmatrix} \times \\ \times \\ \times \end{bmatrix}$$

Figure 6: Incorrect MPI-Parallel SpMV

Immediately we see there is a problem, since now the dimensions of the local matrix A and RHS vector x do not agree.

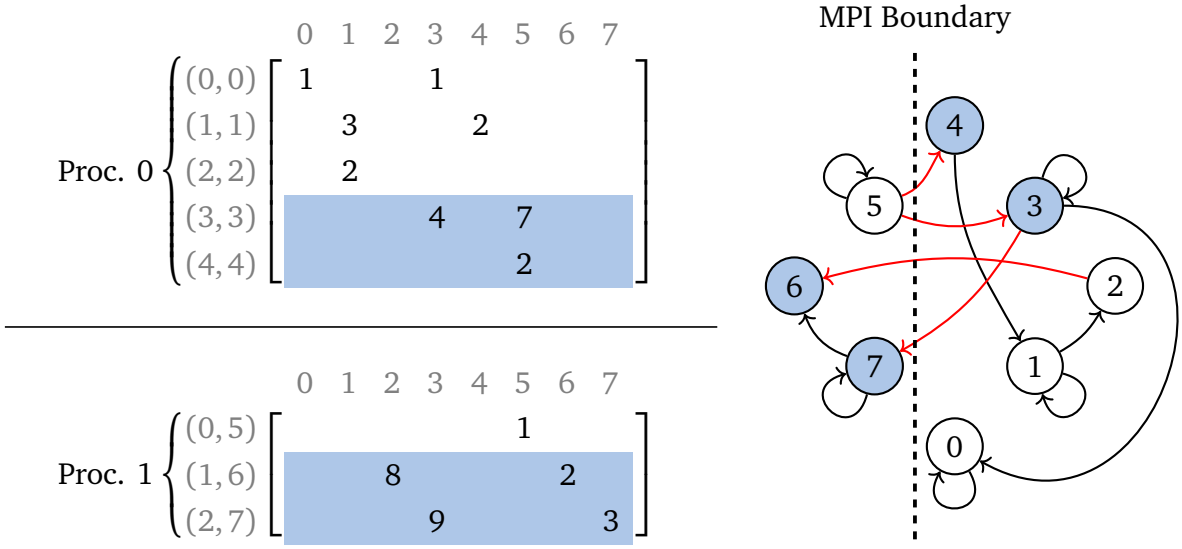
Even if we artificially expand the size of each local x, the data that is needed for SpMV is simply not present. In other words, there now exist data dependencies between the two processes. We cannot compute some rows of the result vector y on MPI process 0 without data from process 1, and vice versa. Specifically, if we're looking at the (local, global) index pair:

- Process 0 needs the element at index (0,5) from process 1's RHS x vector
- Process 1 needs the elements at indices (2,2) and (3,3) from process 0's RHS x vector

The key takeaway here is that these data dependencies are determined by the column indices, since the column index of a given element in A dictates to which element of x it pairs up with for the dot product.

The dependency becomes more clear when we look at the graph representing the global matrix A in Figure 7. The MPI boundary is illustrated as a dashed line, and the weights of the graph are removed.

The rows (vertices) causing trouble are the ones that are highlighted, and the edges that cross the MPI boundary are drawn in red. For us to fulfill these data dependencies,



(a) Segmented matrix

(b) Segmented graph

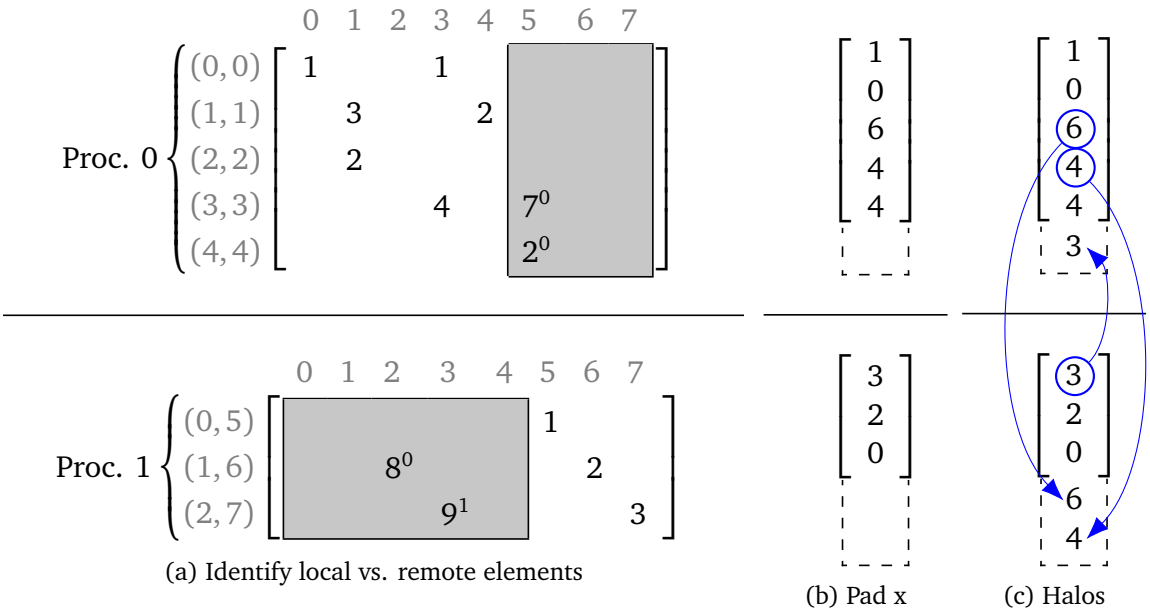
Figure 7: Identify vertices with data dependencies

we need explicit communication in the form of messages. One way to resolve the data dependencies is as follows, illustrated in Figure 8.

(a) Partition the elements of each MPI process-local A into local and remote elements (we need to be careful between "MPI process-local" and local in the sense of "local vs. remote"). Recall the "(process-local row index, global row index)" notation introduced earlier. To make this partition, take η to be the global row index of the first row on this MPI process and θ to be the last. Let e be an element of A, and col(e) its column index. All elements e where $\eta < \text{col(e)} \leq \theta$ are local elements, and all other elements are remote elements. Everything within the grey box is remote. The local elements of A always multiply with the local elements of x. Similarly, the remote elements of A always multiply with the halo elements of x. We need to keep track of which remote elements of A pair with which halo elements in x for the dot product. One way we can do this is to store all the incoming halos "at the bottom" of x, and assign to each remote element an index. This index describes to which halo it multiplies with. This index is generated in a "left to right" manner by scanning each column for halo elements, essentially "compressing" the column indices [16]. These compressed column indices are shown in the superscript. These two steps are illustrated in Figure 8a.

(b) Then, count how many distinct non-empty *columns* of remote elements exist (e.g. 1 on process 0, and 2 on process 1). The number of unique remote columns with at least one element will be the number of spaces by which we pad the process-local x vector. This "buffer padding" step is shown in Figure 8b.

(c) Let r be a remote element on some "needy" MPI process. The element r would



(a) Identify local vs. remote elements (b) Pad x (c) Halos

Figure 8: Halo Communication

like to multiply with the x vector at global row index col(r), but it is not present on this process. The MPI process which *does* contain the x vector with the global row index col(r) must send this element to the needy MPI process in order for the SpMV to execute properly. These required elements are collected to a (necessarily) contiguous buffer and sent to the needy MPI processes. See Figure 8c.

Remark. In this small example, only two processes communicate with one another. Keep in mind that, in general, any MPI process can receive from and send to any number of other MPI processes, including none at all.

Now that halo communication is completed and the data dependencies are fulfilled on each process, the SpMV can proceed as expected (at least for the local elements). As mentioned earlier, the remote elements of A are multiplied only with the remote elements of x, such that the "compressed column" indices in the superscript of the A elements match the index of the "padded" region of the x vector buffer. This is shown in Figure 9.

Compare the result with the shared memory context SpMV in Example 2.1. This process, of communicating remote elements to fulfill data dependencies, is often called "halo communication" or "halo exchange".

Remark. Elements are only sent *from* the local part of the process-local x vector (i.e. solid line part in Figure 8c), and received into the remote part of the process-local x vector (dashed line part in Figure 8c).

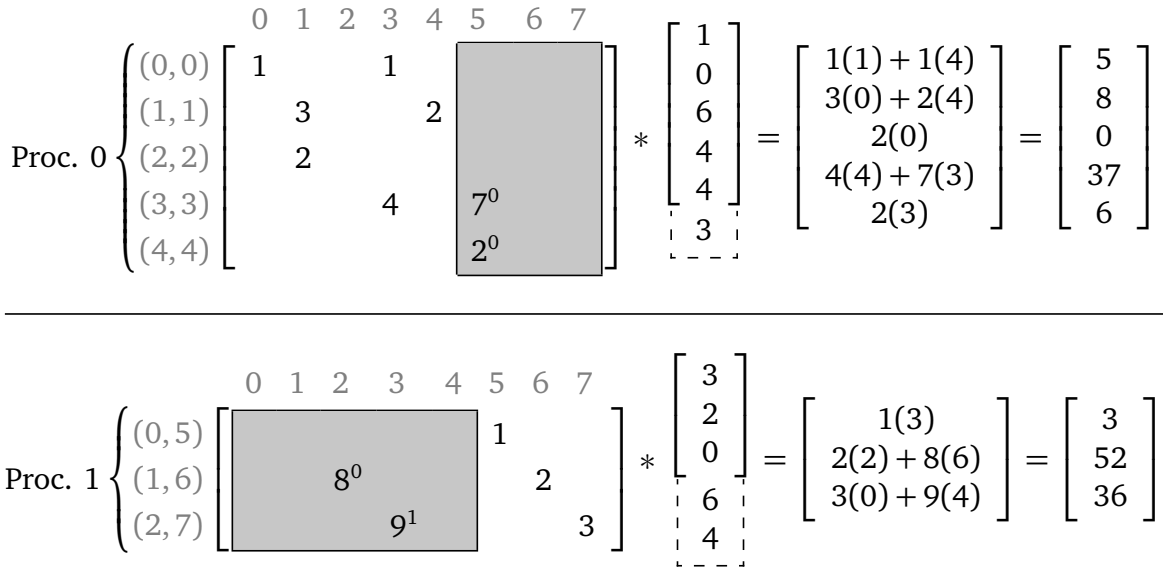


Figure 9: Correct MPI-Parallel SpMV

We formalize this communication process into pseudocode in Algorithm 3. Many details are left out of the communication routine, but it is useful to illustrate the process so that we can refer to it in later sections. Define N_p to be the number of padded rows for the remote part of the x vector, or order to accept the halo communication. First, notice the size of the input and output of this routine. The input vector x is now of size $N_r + N_p$, where N_p is the number of needed halo elements from other processes.

There are a few important arrays that we need for the pseudocode that we have yet to discuss. These are commonly collected into one structure, called a "context", which is constructed during the pre-processing phase of the framework from which this routine is called, i.e. before any communication or computation takes place. It is very useful for keeping data related to communication together, which is unique to this process. We will also follow this norm, as to simplify the function signature for later routines. These arrays are unpacked at the beginning of the routine for clarity.

The array `sendingRanks[]` is the process numbers ("ranks") which send at least one x-vector element to the process calling this routine, and `cSendCounts[]` is the cumulative sum of the number of elements from each of those processes. Similarly, `receivingRanks[]` is the array of ranks that receive a non-zero number of elements from this process, and `cRecvCounts[]` is the cumulative sum of the number of those elements.

The `commIdxs[]` array describes the actual indices of x to be communicated. It is a little more complicated than the previous arrays just described, in that it is an array of arrays. The index of an inner array within the outer array describes to which process elements are to be sent. The elements within an inner array are the actual elements of the x vector to be sent. While a bit convoluted, this gives us a way to keep track of *indices* of the elements in the local part of our x to be sent to each process. For

Algorithm 3: Halo Communication

```
Input : double x[N_r + N_p];                          // remote part empty
        context localCtxt;                            // context structure
        int perm[N_r];                                      // permutation
Output: double x[N_r + N_p];                         // remote part filled
 1 int sendingRanks[] ← localCtxt.sendingRanks;     // unpack context
 2 int cRecvCounts[] ← localCtxt.cRecvCounts;
 3 int receivingRanks[] ← localCtxt.receivingRanks;
 4 int cSendCounts[] ← localCtxt.cSendCounts;
 5 int commIdxs[] ← localCtxt.commIdxs;
 6 for fromRankIdx ← 0 to size(sendingRanks) do
 7 |   int fromRank ← sendingRanks[fromRankIdx];
 8 |   int inSize ← cSendCounts[toRank + 1] − cSendCounts[toRank];
 9 |   mpiRecv(x[N_r + cRecvCount[fromRank]], inSize, fromRank);
10 end
11 for toRankIdx ← 0 to size(receivingRanks) do
12 |   int toRank ← receivingRanks[toRankIdx];
13 |   int outSize ←
   |    cRecvCounts[fromRank + 1] − cRecvCounts[fromRank];
14 |   double toSendElems[outSize];                // contiguous buffer
15 |   for i ← 0 to outSize do
16 |   |   toSendElems[i] ← x[perm[commIdxs[toRank][i]]];
17 |   end
18 |   mpiSend(toSendElems, outSize, toRank);
19 end
```

illustration, here is an example of what this array may look like:

```
commIdxs = [[1,3,5,6,19], [], ..., [8,9,10]]
```

The `perm[]` array describes the symmetric permutation from Section 2.5. If no permutations are used, the identity permutation $\texttt{perm} = [0, 1, 2, \ldots, N_r]$ would be passed.

The purpose of the for-loop on line 16 is to pack these elements, from x, into a contiguous buffer. In order to send the data in a buffer, MPI requires that data be contiguous within the buffer. There exist other methods to pack elements into a contiguous buffer, but this was chosen for simplicity.

Finally, a couple of remarks on the `mpiRecv` and `mpiSend` functions. These are simplifications of the routines defined by the MPI standard [21], but we can at least discuss the arguments. For `mpiRecv`, the arguments are `*recvBuffer`, `*sizeOfRecvBufer`, and `*fromWhichRank`. Notice, we receive into the x vector, but only after the local elements N_r and then moved to the corresponding cumulative sum. For `mpiSend`, we must provide `*sendBuffer`, `*sizeOfSendBufer`, and `*toWhichRank`.

This is just one way to collect and communicate halo elements, as MPI is flexible in

how it enables users to implement communication routines. In practice, one would probably use non-blocking communication here, as to avoid problems with deadlocks. For more details on blocking vs. non-blocking communication, see [21].

Remark. In our implementation, we chose to shift columns that contain remote elements to the right. Having the columns which contain local elements come before the columns which contain remote elements makes many in-code decisions easier, and leads to cleaner code in the author's (limited) experience. This is merely a design choice, not a necessity. This is not depicted in Figures 8, 9, and 6 so as to not needlessly complicate things at this point in the thesis.

We now turn our attention to MPK. Just like in the shared setting, MPK is traditionally implemented as a sequence of p_m many back-to-back SpMVs in the distributed setting. Except now, each SpMV step requires a preliminary communication step (such as Algorithm 3), so each process can receive its needed halo elements, as described above.

Each iteration of the distributed MPK would look something like this:

1. Communicate the required halo elements.
2. Execute Algorithm 1 locally on each process.

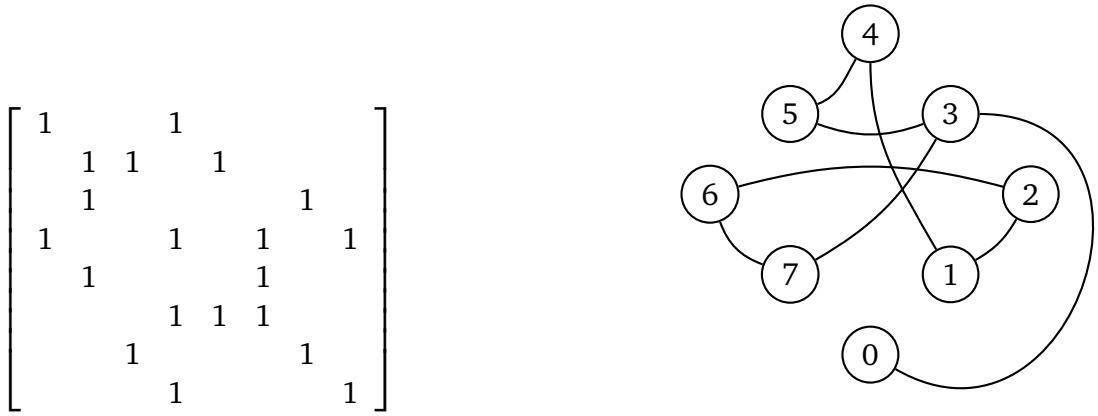
For continuity with previous sections and flexibility for later sections, we present the distributed MPK with the starting row `sRow` and ending row `eRow` options for the SpMV kernel, yet at their limits (i.e. 0 and N_r respectively) to avoid any complications at this point.

Algorithm 4: Distributed Matrix Power Kernel

Input : `double x`$[N_r + N_p]$; `// padded dense RHS input array`
`double val`$[N_{nz}]$;
`int col`$[N_{nz}]$, `rowPtr`$[N_r + 1]$;
`int` p_m;
`context ctext;`
`int perm`$[N_r]$;

Output: `double y`$[N_r + N_p, p_m]$; `// padded local 2D results array`

```
1 y[:, 0] ← x;
2 for p ← 1 to p_m do
3 |   y[:, p − 1] ← haloComm(y[:, p − 1], ctext, perm);
4 |   y[:, p] ← SpMV(0, N_r, y[:, p − 1], val, col, rowPtr);
5 end
```



(a) Forced symmetric matrix

(b) Graphs lose direction

Figure 10: RACE's view of matrices

3 A Distributed Level-Blocked MPK

The Recursive Algebraic Coloring Engine (RACE) is a library originally written for parallelizing kernels with "distance k" dependencies, such as the symmetric sparse matrix vector multiplication kernel (SymmSPMV). The way RACE accomplishes this is by the use of graph coloring (specifically, distance-k graph coloring) techniques. Most of the details of RACE are not discussed here, but the interested reader can see [1]. In this section, we will see how select functionality from RACE is applied to MPK, and how it has already been successful in the shared memory setting. Then, we will develop the main contribution of this work, specifically the extension of LB-MPK to the distributed setting.

3.1 The Recursive Algebraic Coloring Engine

The following description is restricted to symmetric matrices, and consequently, undirected graphs. Internally, RACE represents unsymmetric graphs by symmetric ones, so ultimately there is no difference. It only cares about the structure of the underlying graph. So for this section – and the remainder of this work – we refine what we mean by a "graph". Recall Definition 2.1 and 2.3 from Section 2.5. Our refinement of Definition 2.1, is that we essentially get rid of the W and just now have $G = (V, E)$, and for Definition 2.3 we replace ordered tuples with unordered sets (i.e. the directed edge $(u, v) \in E$ becomes the undirected edge $\{u, v\} \in E$) and we now say these vertices are simply "adjacent". Self-loops on the graph will also be disregarded for clarity. See Figure 10 for an illustration of how RACE would treat the toy matrix from Example 2.1.

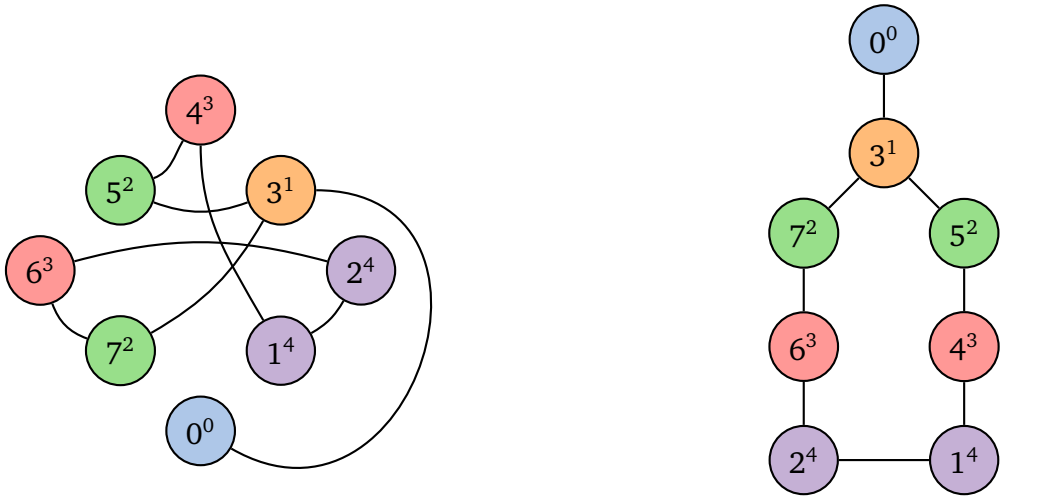
RACE is split up into three main stages: level construction, distance-k coloring, and load balancing. The reason RACE is fundamental to the efficient computation of MPK, and the reason why it is the basis for the contribution of this work is because of the first stage. Therefore, we will take time to focus on the role of level creation, and not

be going into detail regarding the latter two stages.

Level construction is a method by which we can logically segment the graph G corresponding to the matrix A, into structures called "levels". Starting from a "root vertex", we successively expand our view one vertex distance more every iteration. The process is what is known as a Breadth First Search (BFS). At each iteration, we collect all vertices that are the same distance from the root node into the same structure, which we call a "level". Levels consist only of vertices of the graph. This method leads to a natural algorithm for bandwidth reduction, at which point the discussions from Section 2.5 will become especially relevant. This is an instance of a "Level-set ordering", and is one of the more common reordering techniques, see [26].

We first illustrate the relevant parts of level construction, BFS, and BFS reordering with an example.

Example 3.1. Take the symmetric matrix A from Figure 10. Recall, this is just the matrix from Example 2.1 with unit entries and forced symmetry. The superscript of the vertex labels in the following graph will denote to which vertex the level belongs.



(a) Graph perspective of matrix A

(b) Tree perspective of matrix A

Figure 11: Level Construction Example

Step 1: Choose a "root" vertex, from which to start level creation. This root is the only member of the first level, $L(0)$. For simplicity, this root vertex is usually just chosen as vertex 0 as in Figure 11a.

Step 2: Select all vertices that "touch" (that is, are adjacent to) the root node, and assign them to the next level, $L(1)$. This corresponds to vertex 3 in Figure 11a.

Step 3: Iteratively select all nodes which are adjacent to the last level's vertices, and assign them to the next level. In Figure 11a, this step covers levels $L(2)$ - $L(4)$. The method of traversal is exactly BFS.

We denote the total number of levels obtained as N_ℓ. In the above Example Step 3:, we have $N_\ell = 5$.

Definition 3.1. Neighborhood
The neighborhood $N(u)$ of a vertex u is defined

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

In general [1], we can define the i-th level as:

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

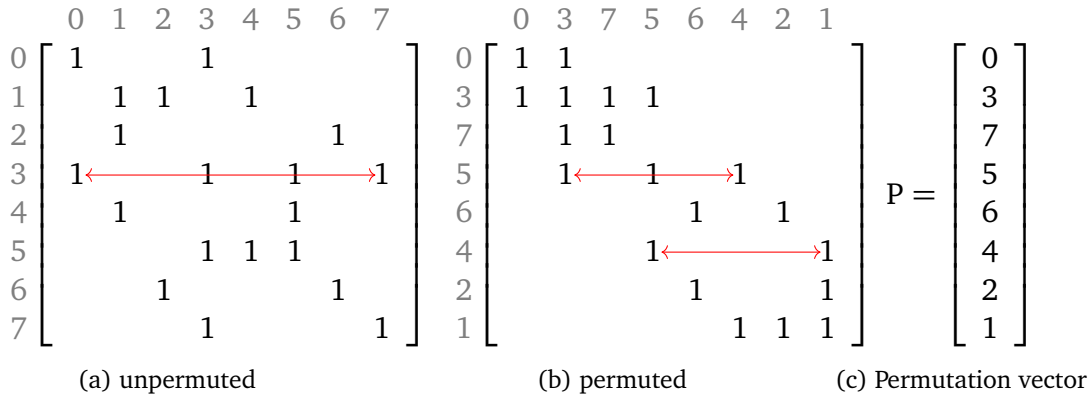
The reason for the term "levels" becomes much more clear when we re-organize our graph into this "tree" – more specifically a "rooted tree" – as in Figure 11b. This organization is not only more natural for the BFS traversal, but it is how RACE will internally represent the matrix.

We store the vertex indices of the entry points to each level of the permuted graph $G' = (V', E')$ into the array $\texttt{levelPtr}[N_\ell + 1]$, such that levels in G' can be identified as:

$$L(i) = \{u : u \in [\texttt{levelPtr}[i] : (\texttt{levelPtr}[i+1] - 1)] \text{ and } u \in V'\}. \tag{2}$$

RACE makes use of these levels by permuting the matrix such that each level $L(i)$ is stored consecutively, and before the vertices of $L(i+1)$. As discussed in Section 2.5, this (symmetric) permutation strategy reduces the bandwidth and allows better use of the RHS x vector by increasing spatial locality.

Take the toy matrix from Example 2.1 and force the symmetric structure and unit values as RACE would do. Then, we permute in exactly this BFS manner as shown in Figure 12.



(a) unpermuted (b) permuted (c) Permutation vector

Figure 12: BFS permutation

3.2 RACE Applied to the MPK

Recall, that MPK is traditionally implemented as a sequence of back-to-back SpMVs. Also, as stated before, for successive iterations of SpMV, the same sparse matrix needs to be loaded from main memory every iteration. There is immense opportunity for data transfer reduction and a lower code balance (i.e. a performance improvement, see Section 2.4) by keeping the maximum amount of relevant parts of the sparse matrix in cache [2].

The main idea is that we compute each iterative SpMV on only a subset of the total rows of the matrix. This is where being able to compute an SpMV on a slice of a vector is necessary, and we finally can make use of the flexibility that `sRow` and `eRow` afford us, as mentioned in Algorithms 1 and 2.

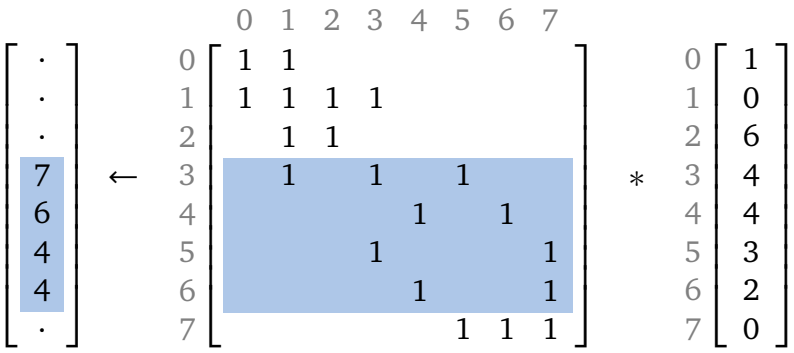
The way these rows are selected will follow shortly, but this subset of rows should be small enough so that all associated data structures can fit into cache and be reusable for the next SpMV. This would improve "temporal locality", as opposed to the improvement on spatial locality that permutations give us. Temporal locality is the ability to reuse data that we've already pulled from main memory and placed into cache. Modern-day cache systems typically use a "Least Recently Used (LRU)"-like eviction strategy. Assume for instance, that our cache is full of data. With this LRU-like strategy, some "age" mechanism is used to determine the "oldest" data that has been loaded from main memory and placed into cache. When new data is loaded from main memory and wants to be placed into our cache, this "oldest data" will be evicted to make room for the "new" data.

For our purposes, the data we wish to reuse would be elements of A (specifically the CRS arrays `val[]`, `col[]`, and `rowPtr[]`) that were required for the previous SpMV iteration, and are still residing in cache. This technique, used all over HPC, is called "blocking" [12].

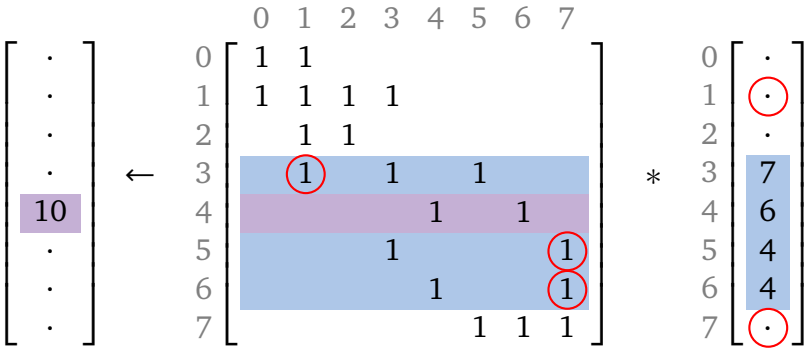
Before going into more detail about the specific blocking strategy RACE uses for LB-MPK in the shared context – and how we can then extend to the distributed context – we need to first pay special attention to the dependencies that exist between successive iterations of SpMVs in the shared context MPK.

Example 3.2. We illustrate with our forced symmetric, permuted toy matrix from Figure 12. We've reset the row and column indices for clarity. Assume we've already made the initialization assignment $\texttt{y}[:,0] \leftarrow x$. We show Algorithm 2 for computing Ax, then A^2x. Except now, we select a subsection of rows $\texttt{sRow} = 3$ to $\texttt{eRow} = 6$.

Step 1: $\texttt{y}[\texttt{sRow} : \texttt{eRow}, 1] \leftarrow A[\texttt{sRow} : \texttt{eRow}, :]\texttt{y}[:, 0]$



Step 2: $\texttt{y}[???, 2] \leftarrow A[\texttt{sRow} : \texttt{eRow}, :]\texttt{y}[\texttt{sRow} : \texttt{eRow}, 1]$



In Step 1, we can supply the SpMV with all rows of the $\texttt{y}[:, 0]$ vector as the incoming RHS vector. But after the first SpMV, the resulting LHS vector does not have all of its rows present, only sRow to eRow. By

$$\texttt{y}[\texttt{sRow} : \texttt{eRow}, 1] \leftarrow A[\texttt{sRow} : \texttt{eRow}, :]\texttt{y}[:, 0],$$

the $\texttt{y}[\texttt{sRow} : \texttt{eRow}, 1]$ vector will be the RHS vector input for the next SpMV iteration

$$\texttt{y}[???, 2] \leftarrow A[\texttt{sRow} : \texttt{eRow}, :]\texttt{y}[\texttt{sRow} : \texttt{eRow}, 1].$$

But, as shown by the highlighted red circles, this presents an issue for some matrix elements at rows 3, 5, and 6. This is because they would like to multiply with elements of the x vector which are not present, namely at indices 1 and 7. In other words, there now exist dependencies between successive SpMV iterations. The RHS input vector in Step 2 does not contain the row element corresponding to the column index of the "problem" matrix elements. Thus, the x vector elements at rows 3, 5, and 6 cannot be promoted in Step 2.

Define $\mathbf{I}$ to be the set of row indices of the matrix A for the initial SpMV invocation, i.e. sRow to eRow. As illustrated in Example 3.2, we calculate the next power on a (usually) "smaller" subset of rows. This smaller subset of rows is those in which all

dependencies are fulfilled from the previous SpMV iteration, which we refer to as **K**. The set **K** is defined as the rows in **I** such that the column indices of the non-zero elements in those rows are within the range of **I**. We also understand $\mathbf{K} = \mathbf{K}(\mathbf{I})$ as another way to express the dependency of **K** on **I**. With this notation, we can rewrite Step 2 of Example 3.2 as

$$\mathrm{y}[\mathbf{K}(\mathbf{I}), 2] \leftarrow A[\mathbf{I}, :]\mathrm{y}[\mathbf{I}, 1].$$

Obviously we don't want the next SpMV iteration to have far fewer rows than the previous SpMV iteration, because then our blocking strategy would not be very efficient. In fact, [2] quantifes the overhead of this approach as the ratio of the previous SpMV's number of rows to the following SpMV's number of rows.

We return to our choice of rows **I** for the initial SpMV. Our precise formulation of the set $\mathfrak{C}$ follows from [2]. With **I** defined as above, the set $\mathfrak{C}(\mathbf{I})$ is the column indices of all non-zero entries in the rows of **I**. That is, if $i \in \mathbf{I}$, then $j \in \mathfrak{C}(\mathbf{I}) \iff A_{i,j} \neq 0$.

Based on this notation, the requirement of **K** is that $\mathfrak{C}(\mathbf{K}) \subseteq \mathbf{I}$ [2]. Furthermore, the SpMV operation for a given row index $i \in \mathbf{I}$ can be written:

$$y_i = \sum_{j \in \mathfrak{C}(i)} A_{i,j} x_j \tag{3}$$

RACE will internally consider SpMV as a graph traversal problem. If $G = (V, E)$ is the adjacency graph representation of a sparse matrix A, then for every vertex $u \in V$:

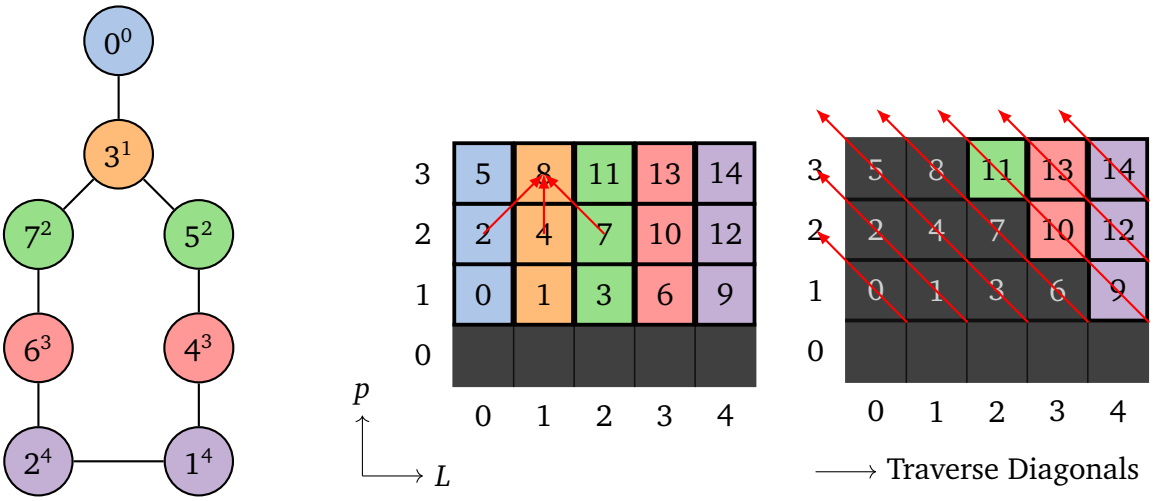
$$y_u = \sum_{v \in N(u)} A_{u,v} x_v \tag{4}$$

Remark. Notice the similarities to Equation 3. This establishes the correspondence from the row index i and column index $j = \mathfrak{C}(i)$ of A, and vertex u and neighboring vertex $v \in N(u)$ of G.

To better understand and visualize MPK dependencies between the successive SpMVs, we make use of the "Lp-diagram" developed in [2], and adapt it slightly to make our extension later more natural. Taking the point of view of "level computations" allows us to move a step up from the cumbersome "row-by-row" point of view, while not losing any generality.

The Lp-diagram in Figure 13 is created from the toy matrix A from Example 2.1. The x-axis represents the indices of the levels (as explained in Section 3.1), and the y-axis represents the stages of MPK from $1 \leq p \leq p_m$. The range of p from 0 to 3 indicates that our goal here is to calculate A^3x. If we take a particular box at coordinate $(L(i), p)$ in the diagram, this represents an SpMV on the vertices contained in level $L(i)$ (i.e. the rows from `sRow` = `levelPtr[i]` to `eRow` = `levelPtr[i+1]`), to compute from the power $p-1$ to p.

The benefit of the Lp-diagram is that now, dependencies can now be seen immediately for any stage in the MPK. To satisfy the dependencies of box at coordinate $(L(i), p)$, the



(a) Tree Graph of A (b) Dependency Pattern (c) Computation Pattern

Figure 13: Lp-Diagram Example

boxes at coordinates $(L(i-1), p-1)$, $(L(i), p-1)$, and $(L(i+1), p-1)$ must already be computed. The power $p = 0$ is already fulfilled for every level, as this corresponds to $A^0 y = x$, which already exists as the RHS input vector (i.e. x is already "calculated"). The dependencies that need to be fulfilled in order to compute the rows in the box at coordinate $(L(1), 3)$ are seen in red arrows in Figure 13b.

One way we can ensure that these dependencies are fulfilled at any moment in out calculation, is to traverse the Lp-diagram such that:

1. Each *diagonal*, defined by $i + p :=$ const, is traversed in a "bottom-right to top-left" fashion
2. The diagonals themselves are traversed from left to right. That is, start at the leftmost diagonal. Once the diagonal is finished being traversed, move to the diagonal immediately to the right until all diagonals are traversed.

This order of execution ensures that the boxes at levels $L(i-1)$, $L(i)$, and $L(i+1)$ are updated to $p-1$ before the box at level $L(i)$ is promoted to p. Hence, the boxes of Figures 13b and 13c are labeled in the order of execution.

Remark. It is important to note, that this is only one way to ensure dependency fulfillment. As we will see in Section 3.3, there are other ways to accomplish this.

The strategy that is used to select the levels to load into cache (i.e. the set of rows **I**, in the language of Section 3.2) is as follows [2]:

1. We first need to understand how often a given level is "reused", i.e. assuming a level $L(k)$ has just been touched, when will it be touched again (assuming the computation pattern in Figure 13c)? One can derive this distance to be $p_m + 1$ [2].

2. If all the matrix elements associated with these $p_m + 1$ levels fit into cache, then all data needed for SpMV on $L(k)$ (except the first, as we need to load elements from main memory at least once) will come from cache. This will lower the amount of data accessed from main memory. Recall from Section 2.4, this is exactly what is needed to improve SpMV performance, assuming the Roofline Model [34].

3. If all associated matrix elements from $L(k)$ do not fit into cache, then there are further optimizations that can still enable cache blocking (e.g. recursion on these "bulky" levels to generate new, smaller levels that *will* fit into cache). These are outside the scope of this work, see [2] for more details.

Lastly, we show the pseudocode for what this Level-Blocked MPK (LB-MPK) would look like, given the discussion above. We need one additional array, as compared to the traditional MPK in Algorithm 2. Recall the `levelPtr[]` from Equation 2, which holds the level data for our permuted graph. With this information about the levels, LB-MPK is as follows in Algorithm 5.

Algorithm 5: Level-Blocked Matrix Power Kernel

Input : `double x`$[N_r]$;
`double val`$[N_{nz}]$;
`int col`$[N_{nz}]$, `rowPtr`$[N_r+1]$;
`int levelPtr`$[N_\ell+1]$; `// level pointer array`
`int` p_m;
Output: `double y`$[N_r, p_m]$;

```
1  y[:,0] ← x;
2  int L_m ← size(levelPtr);                    // total number of levels
3  for d ← 1 to L_m + p_m − 1 do
4  |  int p_start ← max(1, d − (L_m − 1));      // account for wind up
5  |  int p_end ← min(d, p_m);                  // account for wind down
6  |  for p ← p_start to p_end do
7  |  |  int i ← (d − p);                       // execute along diagonals
8  |  |  int sL ← levelPtr[i];
9  |  |  int eL ← levelPtr[i + 1] − 1;
10 |  |  y[K(sL : eL), p] ← SpMV(sL, eL, y[:, p − 1], val, col, rowPtr) ;
11 |  end
12 end
```

Notice the lack of padding on `x` and `y`, since we are still in the shared memory context. Recall from Section 3.1 that `levelPtr[]` is made from the BFS permuted graph G'. So, we may also need to apply the inverse permutation vector P^{-1} to our results vector y after all SpMVs, if we want to recover the original ordering of the results vector. This is situation dependent, as one may want to leave y in the "permuted space". See Section 2.5 for details.

3.3 Extending LB-MPK to the Distributed Setting

The main idea of DLB-MPK is to shift the load of computation ahead in time – in order to better utilize the cache – while shifting the load of communication further back in time. Recall, DLB-MPK requires no more communication than TRAD, the traditional back-to-back SpMV implementation of MPK.

To extend Algorithm 5 to the distributed setting, first assume we are starting with a global square matrix, partitioned in some manner so that each process has some row-wise "slice" of it. We make an initial call to RACE locally on each process to create the local levels, permutation vectors, and initialize any internal data structures that RACE requires (as in Section 3.1), taking care of local vs. remote parts (as in Section 2.6). This initial step will be just called "RACE Pre-Processing", and not included in the final algorithm DLB-MPK.

The rest of the process is spread over three phases:

1. MPI Pre-Computations: execute the initial halo communication, and compute support of LB-MPK in the ring diagrams (to be explained shortly).
2. Local LB-MPK: use RACE to compute the inner parallelogram of the ring diagram with cache blocking.
3. MPI Post-Computations: finish computing the top corners of the ring diagram.

This needs to be done in a way that keeps all dependencies satisfied, which is the central issue addressed in this thesis.

Example 3.3. Say we wanted to start Algorithm 5 on three MPI processes. We'll begin with the Lp-diagrams from Section 3.2, and see where it leads us.

There are two points of view illustrated in Figure 14. In the left column, we have a kind of "distributed" Lp-diagram, in which the x-axis of the diagrams represents the level index $L(i)$, and the y-axis represents the power p. We vary the number of levels artificially across MPI processes (although, we will see a matrix that has exactly this level structure in Figure 15). In the right column, we have the "ring diagrams". These "rings" can be thought of as groups of vertices, such that all the vertices in a group lie the same "distance" away from the remote elements, with a (typically) "large" group of vertices called the "main ring". Exactly what "rings" are in this context will be explained in the coming sections, but for now just know that these two points of view are equivalent. The ring diagrams draw inspiration from the Lp-diagrams, but should not be confused with them. The x-axis of the diagrams in the right column represent the ring index R_i, and the y-axis represents the power p.

Let's focus on just the Lp-diagrams in the left column. We can immediately see a few differences between the Lp-diagrams in the shared memory context of Section 3.2, versus the distributed memory context shown in Figure 14. There are "buffer levels" (labeled by "B") in grey on the left of each diagram, the numbered step labels aren't

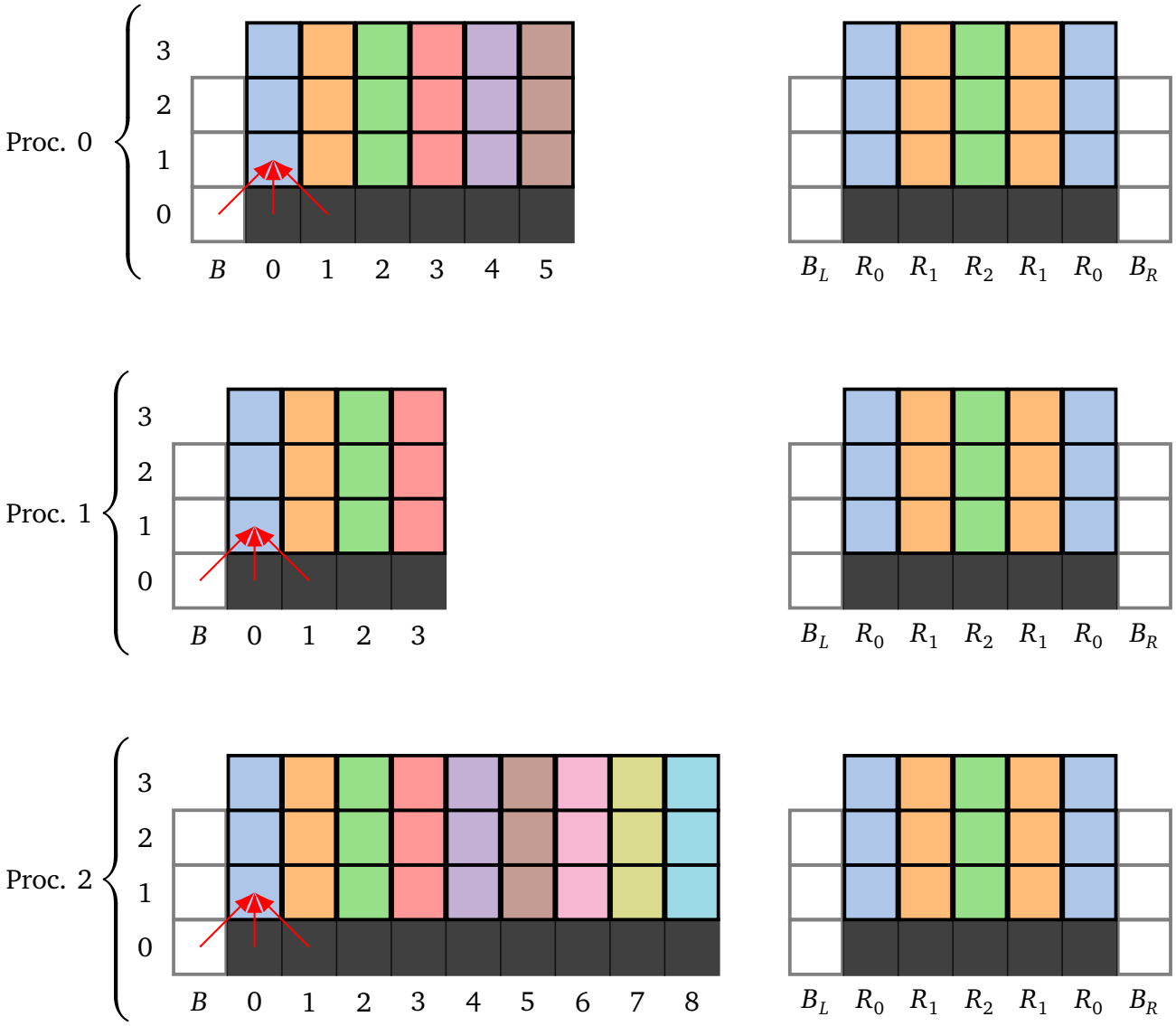


Figure 14: Distributed Lp-diagrams vs. Ring diagrams

present on the boxes, and the dependencies are not fulfilled to compute Ax for all levels.

The buffer levels that we see in Figure 14, are the Lp-diagram equivalent of the dashed "remote part" of the RHS x vectors from Section 2.6, i.e. the padded portion of x vectors which have been resized to accept incoming halo elements from other processes. In other words, the grey boxes represent the halo elements that must be collected in order to compute the next power for level 0. We should emphasize that, in the Lp-diagrams in the left column, the buffer is only depicted as being on the left because only the "level 0" vertices of the graph have edges which extend over the MPI boundary.

Recall from Section 2.6, that in order to satisfy the dependencies of the box at coordinate $(L(i), p)$, the boxes at coordinates $(L(i-1), p-1)$, $(L(i), p-1)$, and $(L(i+1), p-1)$ must already be computed. These buffer levels are key because as you can see, we can't even compute the first box for $p = 1$, $i = 0$ in the lower left corner. This is emphasized with the red arrows.

The DLB-MPK algorithm is given in Algorithm 6.

Algorithm 6: Distributed Level-Blocked Matrix Power Kernel

Input : `double x`$[N_r + N_p]$;
`int distFromRemotePtr`$[p_m + 1]$; `// boundary level array`
`commFuncType commFunc;`
`commFuncArgType commArgs;`
`spmvFuncType spmvFunc;`
`spmvFuncArgType spmvArgs;`
Output: `double` $y[N_r + N_p, p_m]$;

```
1 userInput ← commFunc, commArgs, spmvFunc, spmvArgs;
2 [x, y] ← mpiPreComp(x, y, distFromRemotePtr, userInput);
3 [x, y] ← localLBMPK(spmvFunc, spmvArgs);
4 [x, y] ← mpiPostComp(x, y, distFromRemotePtr, userInput);
```

The array `distFromRemotePtr[]` is a central data structure to the contributions of the thesis, and to the creation of the "rings" mentioned earlier. Recall, DLB-MPK is separated into three main stages, with a pre-processing step not given within Algorithm 6.

3.3.1 RACE Pre-Processing

We focus here on only the pre-processing steps of RACE that are directly relevant to our results. That will be: the registering of the SpMV function, communication function, and creation of the "`distFromRemotePtr[]`" array. As the name suggests, these pre-processing routines will be used somewhere earlier in the host framework. The `distFromRemotePtr[]` array represents the cumulative sum of the "level rows" in

the matrix at hand. A level row is a row-wise slice of our local BFS permuted matrix, such that all vertices in the slice are the same distance from the `root`. It is directly connected to our notion of "rings". As nice as the toy matrix from Example 2.1 has been for illustrating concepts thus far, it will be better for us to have something larger to help illustrate the creation of the `distFromRemotePtr[]` array as well as more notions to come.

When moving to the distributed setting, we select the nodes that lie on the MPI boundary, or "boundary nodes", as our *roots* for the BFS. In other words, all vertices of the graph with edges that extend over the MPI boundary are boundary nodes. They are identified programmatically as having corresponding rows that have connections to remote columns, i.e. an element with a column index larger than N_r, (due to the shifting of remote columns to the right of local columns as discussed in the remark at the end of Section 2.6). This corresponds to the boundary node being adjacent to at least one remote element (i.e the vertices which have data dependencies). The reason for choosing these vertices as our roots for the BFS is that it allows us to determine the distance of each vertex from the MPI boundary. This is, in a way, a generalization to our earlier notion of a "level". Executing BFS from multiple roots often gives way to a certain symmetry in the graph traversal, and it is exactly these generalized levels which we call "rings". The roots, or "distance-1" vertices from the remote elements, constitute our first "ring".

We first need to assume a power p_m in which to compute $A^{p_m}x$, as this determines when to "finish" the construction of `distFromRemotePtr[]`. The reason is that the power p_m dictates how many steps of BFS to traverse inwards from the roots. Then at each step, the newly "touched" vertices are collected into the next ring. As mentioned before, the *cumulative sum* of the number of these vertices is what makes up `distFromRemotePtr[]`. We continue this process for $p_m - 1$ many steps. After the last step, all remaining vertices that have not been touched yet are assigned to the "main ring". In other words, all vertices with a distance greater than $p_m - 1$ from the boundary nodes are "collapsed" into the main ring. Referring back to Figure 14, this main ring is R_2. The main ring will be very important for us, as is describes which vertices that LB-MPK can execute over. All vertices in `distFromRemotePtr[p_m - 1]` to `distFromRemotePtr[p_m]` reside in the "main ring".

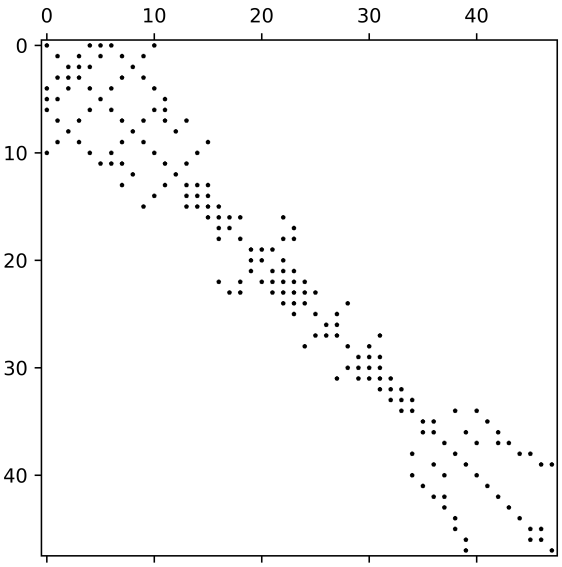


Figure 15: Larger toy matrix

Example 3.4. Let's construct the `distFromRemotePtr[]` using the new toy matrix.

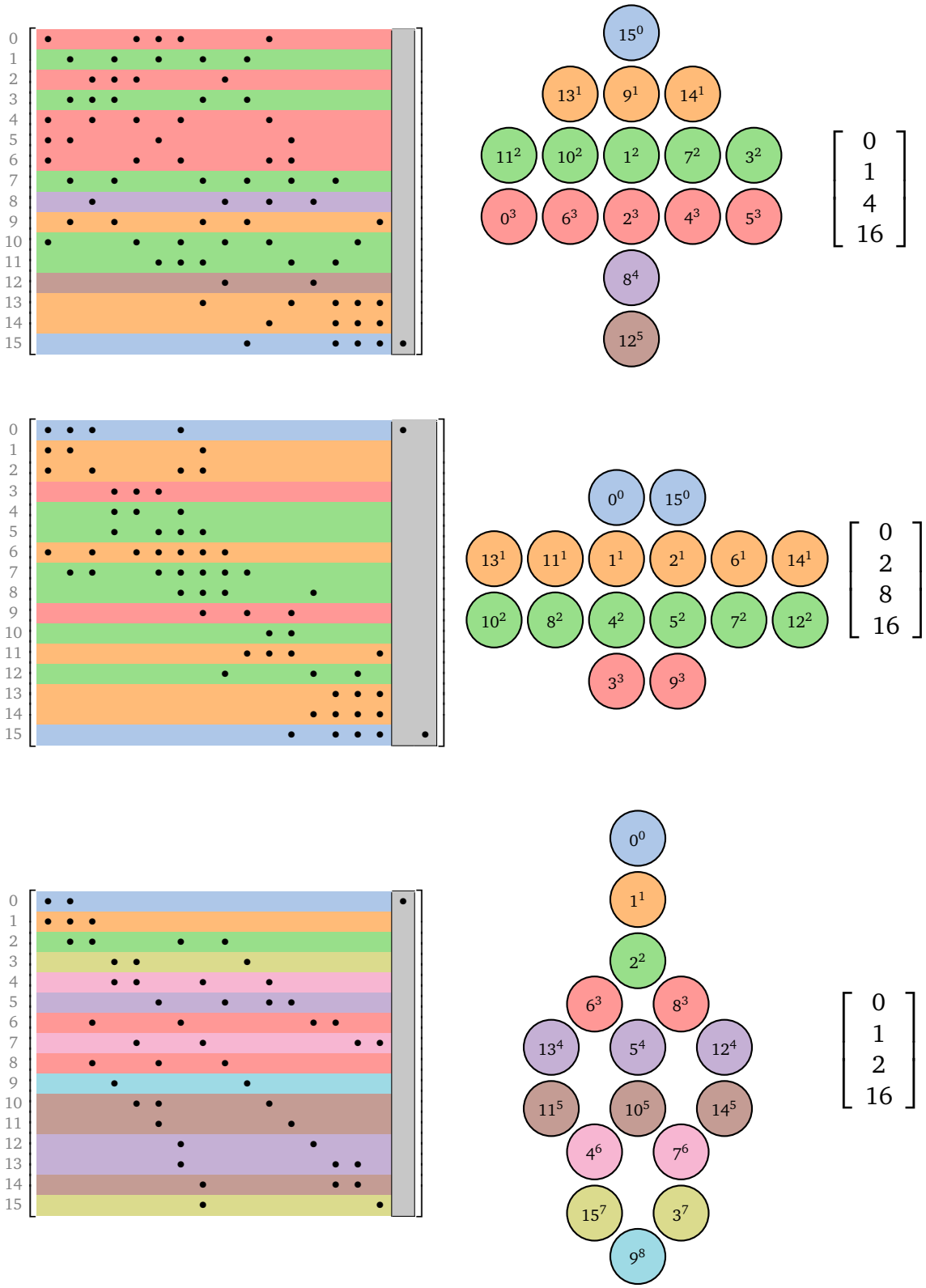


Figure 16: Local Partitions, Levels, and `distFromRemotePtr[]` arrays

In the left column of Figure 16, we first evenly segment the matrix into 16 rows on each MPI process. Here, we chose to move columns that contain remote elements to the right of the columns that contain local elements [16]. Then in the middle column of Figure 16, levels are collected locally on each process using the boundary nodes as roots for the BFS. Finally in the right column, the `distFromRemotePtr[]` is shown for each local process, assuming we want to compute A^3x (i.e. `size(distFromRemotePtr[]) == 4`). For clarity of the figure, the edges of the graph are omitted.

Traversing each local graph in this manner – i.e. using these boundary nodes as our roots for the BFS – and then collecting the cumulative sum of same distance vertices into `distFromRemotePtr[]` gives us two ways to better understand MPK. More generally, say we wanted to compute $A^{p_m}x$ across all MPI processes. One way to interpret `distFromRemotePtr[]` is through data dependency fulfillment. The vertices in the levels that fall between `distFromRemotePtr[0]` and `distFromRemotePtr[1]` (i.e. the boundary nodes) need k communications to fulfill the data dependencies necessary to compute A^kx for all levels, $0 \leq k \leq p_m$. So as previously mentioned, just as in the traditional back-to-back SpMV approach of Algorithm 4, DLB-MPK requires p communications to compute $A^{p_m}x$. This would correspond to filling up the grey boxes in Figure 14.

Another, perhaps more fruitful way to interpret `distFromRemotePtr[]` is through local LB-MPK efficiency. Vertices in the levels that fall between `distFromRemotePtr[k]` and `distFromRemotePtr[k+1]` for $0 \leq k \leq p$ have all the required dependencies to be computed to power $k+1$, after the initial halo communication (as discussed in Section 3.3.2). So if we are trying to compute A^3x as we are in the above example, and most vertices come after `distFromRemotePtr[2]` on each MPI process, we expect that DLB-MPK will perform very well. Refer to the three Figure 16 in Example 3.4. By this logic, we would expect DLB-MPK to perform higher on MPI process 2 than on processes 0 or 1, since it has the largest "main ring".

Let `computeDistRemote` be the routine that constructs the `distFromRemotePtr[]`. It does this by first summing all boundary nodes (i.e. rows which contain elements with a column index larger than N_r), and placing that sum into `distFromRemotePtr[1]`. Then it traverses the graph in a BFS manner, and successively places the cumulative sums at the appropriate indices of `distFromRemotePtr[]` as described above and in Example 3.4. The relevant initialization steps in RACE would look like Algorithm 7. We provide our own `commFunc` and `spmvFunc` as an example, from Algorithm 3 and 1 respectively, although in practice these would be supplied by the user.

The structures given to RACE by the user in Algorithm 7: the CRS arrays, LHS vector y, RHS input vector x, permutation vector `perm[]`, and communication information is omitted in the input for brevity.

The ability of a user to register their own routines for communication and computations, seen in the RACE-provided routine calls `registerCommFunc` and `registerSpMVFunc`, is what makes the final DLB-MPK algorithm so flexible. Furthermore, arguments for these functions are packed into structures (`commArgs` and `spmvArgs`), also defined by the user. The RACE library just requires `*void` pointers which point to these structures.

Algorithm 7: Initialize RACE

Input : *(All user provided args, omitted for brevity)*
```
*void haloComm;
*void SpMV;
```
Output:
```
int distFromRemotePtr[p_m + 1];      // boundary level array
commFuncType commFunc;
commFuncArgType commArgs;
spmvFuncType spmvFunc;
spmvFuncArgType spmvArgs;
```
```
1 distFromRemotePtr ← computeDistRemote();
2 commFunc ← registerCommFunc(haloComm);
3 commArgs ← x, y, localContext, perm;
4 spmvFunc ← registerSpmvFunc(SpMV);
5 spmvArgs ← x, y, val, col, rowPtr;
```

Remark. Though it may seem counter-intuitive, we need both x and y for the communication and SpMV routines for our implementation (at least, for our implementation). The reason is that, depending on the method of computation, the role of x (i.e. the RHS input array, or the LHS results array) could change based on the parity of the power for which we are currently computing. If we are keeping only the highest power in y (as opposed to all powers), than the two parities are handled as follows. For computing odd powers p in $y \leftarrow A^p x$, x is the input array and y the output array. For even powers, these roles reverse.

3.3.2 MPI Pre-Computations

This phase accomplishes two things: to call the initial communication routine, and carry out some "helper computations" for the local LB-MPK invocation which follows.

Recall from the distributed Lp-diagrams in Figure 14, we do not even have the dependencies present to compute Ax on all levels. The first goal of the MPI Pre-Computations phase is to fulfill these dependencies. We will use the ring diagrams to show how these dependencies are fulfilled, but from the point of view of the Lp-diagrams it is very similar. The second part supports the central "diamond" parallelogram of the ring diagrams, as will be seen in Section 3.3.3. Essentially, this part of the MPI Pre-Computation phase computes the bottom corners of the ring diagrams in Figure 14, such that the cache-blocked parallelogram region can have all of its dependencies satisfied. Once these bottom corners are fulfilled, we can apply LB-MPK on each MPI process.

In the ring diagrams in the left column of Figure 17, the numbers on the boxes show the step number, similar to the shared memory context of Figure 13. In the right column, we have the exact same procedure of rings growing inwards, viewed differently as a visual aid.

We first illustrate the "communication" part of this phase. It is worth emphasizing

that communication will always precede a computation, as the communication part of this phase fulfills the dependencies required by the computation part of this phase. This is also seen in the pseudocode of the MPI Pre-Computations phase in Algorithm 8. The communication is again represented by the dark blue arrows. Even though in this example, communication is depicted as only occurring between neighboring processes; in general any process can communicate with any other process. This is why we've chosen to place buffer levels on both sides of the ring diagrams. The colors in the left column of the following figures do not correspond to the colors in the right column.

Remark. Processes can only receive elements into their remote buffers, shown here in the outer grey ring as "Remote elements". On any given MPI process, only elements from the outer-*local* ring (i.e. the collection of boundary nodes edged by dark blue, *not* the Remote elements) are sending halo elements.

In Figure 18, we illustrate the "computation" part of this phase. With the dependencies fulfilled in the box at coordinates $(B_L, 0)$ and $(B_R, 0)$ by the communication part of this phase, the boxes at $(R_0, 1)$ can now be computed (notice the symmetry on both sides of the ring diagram). In line 4, we begin to iterate over all rings and perform supporting computations when necessary. The C++ style ternary operator is used on lines 7 and 8 to save space, but it is equivalent to an if-else statement. In lines 10 and 11, we make use of the `distFromRemotePtr[]` to extract the proper start and end rows for our `spmvFunc`. Recall, we need this functionality to execute on a subset of the rows in order for cache blocking. Lastly in line 12, we make use of this flexible `in` and `out` notion for the RHS and LHS vector respectively, which was remarked at the end of Section 3.3.1.

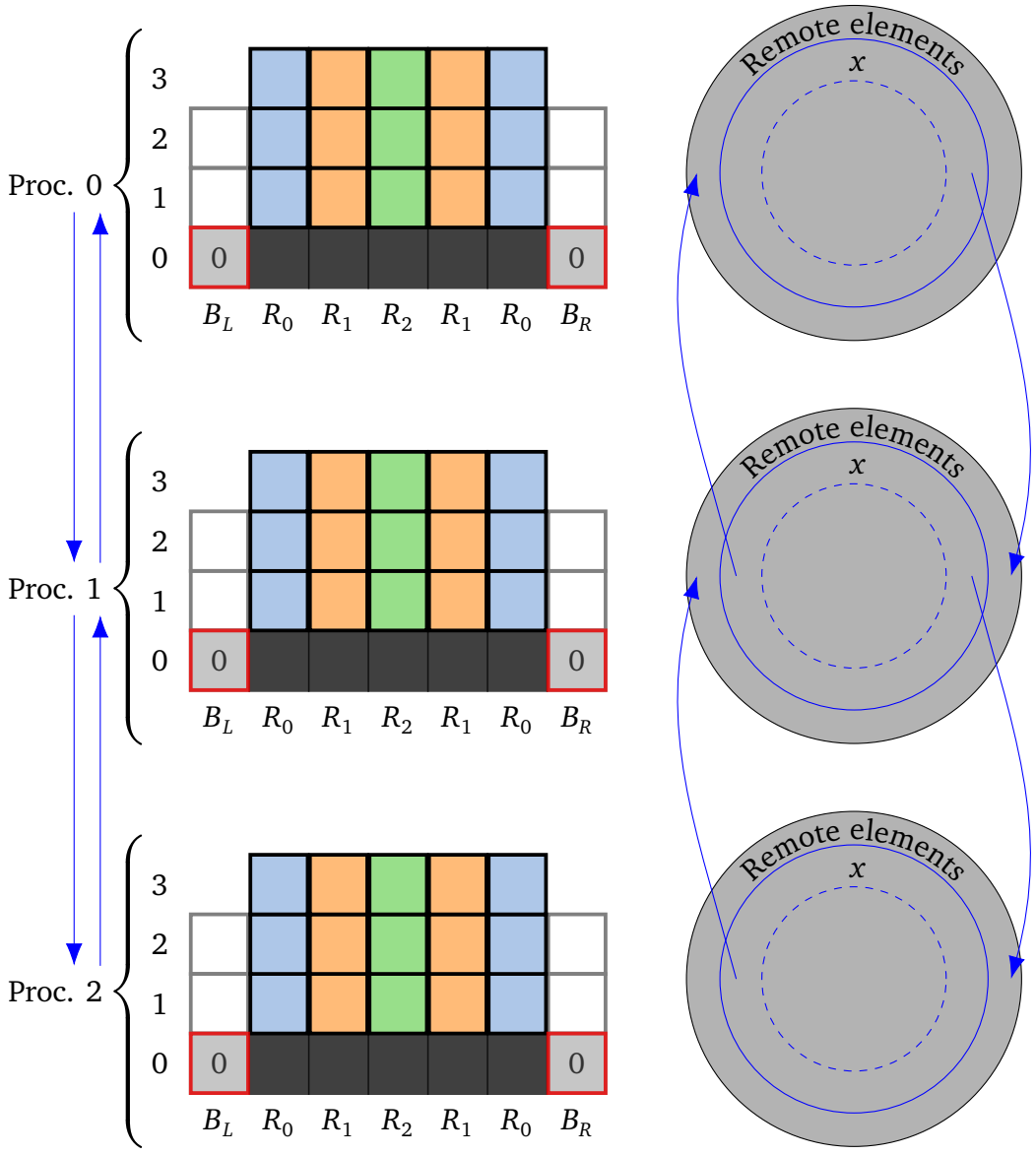


Figure 17: MPI Pre-Computations: Initial Communication

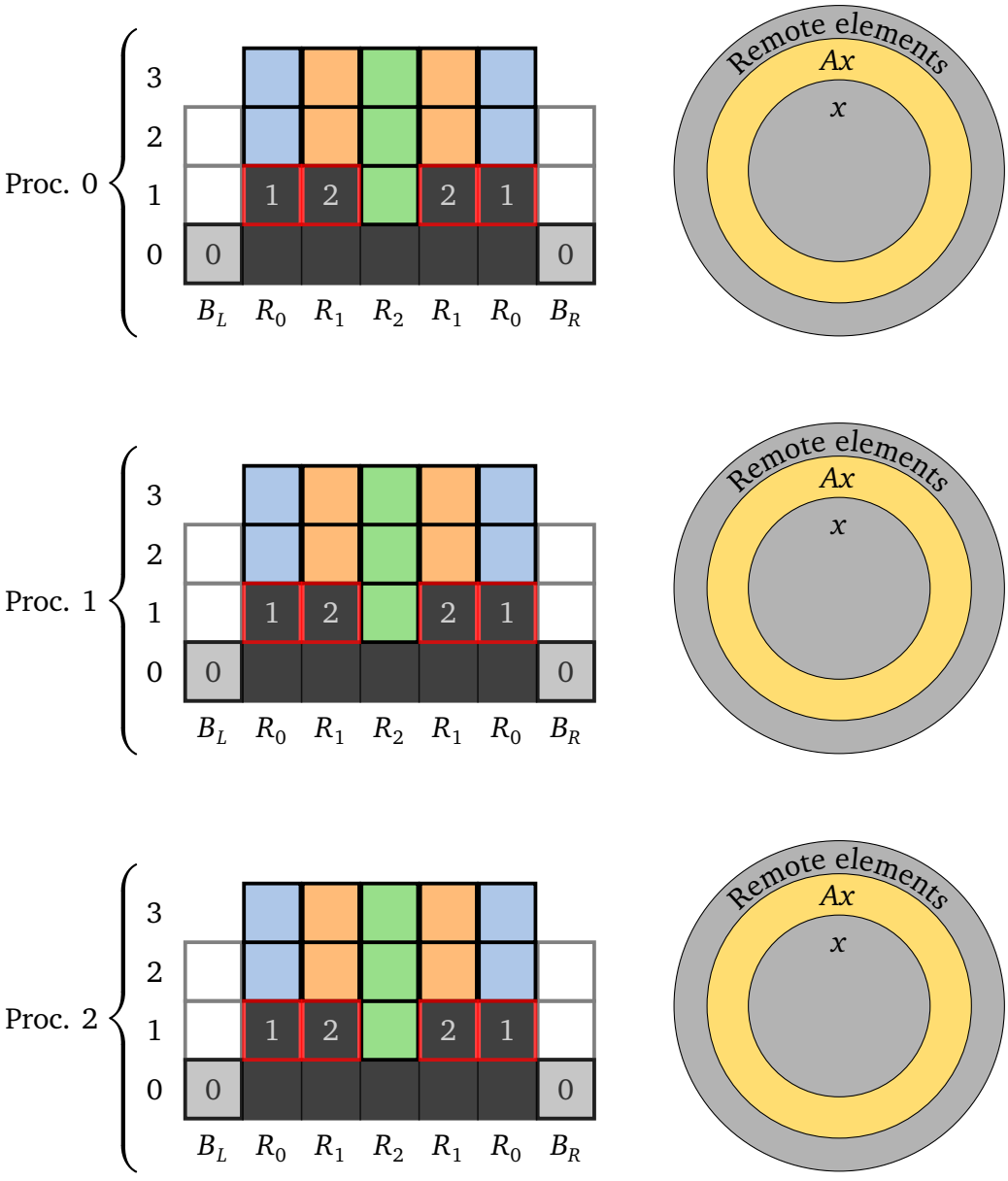


Figure 18: MPI Pre-Computations: Initial Computation

Algorithm 8: MPI Pre-Computation Phase

Input : `int distFromRemotePtr[`$p_m + 1$`];`
`commFuncType commFunc;`
`commFuncArgType commArgs;`
`spmvFuncType spmvFunc;`
`spmvFuncArgType spmvArgs;`
Output: `double x[`$N_r + N_p$`];`
`double y[`$N_r + N_p, p_m$`];`

```
1  p_m ← size(distFromRemotePtr) - 1;
2  [x,y] ← commFunc(commArgs, 1);                    // halo exchange
3  int totalPow ← 1;
4  for mpiRingIdx ← 1 to totalPow do
5  |  int curLevel ← (totPower-1)-mpiRingIdx;
6  |  for pow ← 0 to totalPow do
7  |  |  int in ← (pow % 2) ? x : y;
8  |  |  int out ← (pow % 2) ? y : x;
9  |  |  int powLevel ← curLevel+pow;
10 |  |  int sL ← distFromRemotePtr[powLevel];
11 |  |  int eL ← distFromRemotePtr[powLevel+1]-1;
12 |  |  out ← spmvFunc(sL,eL,in,out,spmvArgs.crsArrays);
13 |  end
14 |  if mpiRingIdx % 2 != 0 then
15 |  |  totalPow ← totalPow + 1;
16 |  end
17 end
```

3.3.3 Local LB-MPK

For us, LB-MPK will be a black box that computes the inner supported parallelogram. For more details, see [2]. The matrix in Figure 15 was deliberately chosen to give variety in the number of levels in each process. This was done with the intention of demonstrating that, from the perspective of DLB-MPK, the pattern of computation does not change.

The number of levels in a matrix tends to be large in practice, and so the size (i.e. the number of vertices stored inside) of this main central ring far exceeds the size of the other rings. While no routine will be given for LB-MPK in this thesis, the computation pattern can just be assumed to be the inner parallelogram, whose edges in the ring diagrams are supported by the MPI Pre-Calculation phase. For more examples of this pattern, see [2].

It is worth noting that no MPI communication takes place in this phase of DLB-MPK. Furthermore, the RACE library has no knowledge of MPI (i.e. does not use any MPI routines or header files).

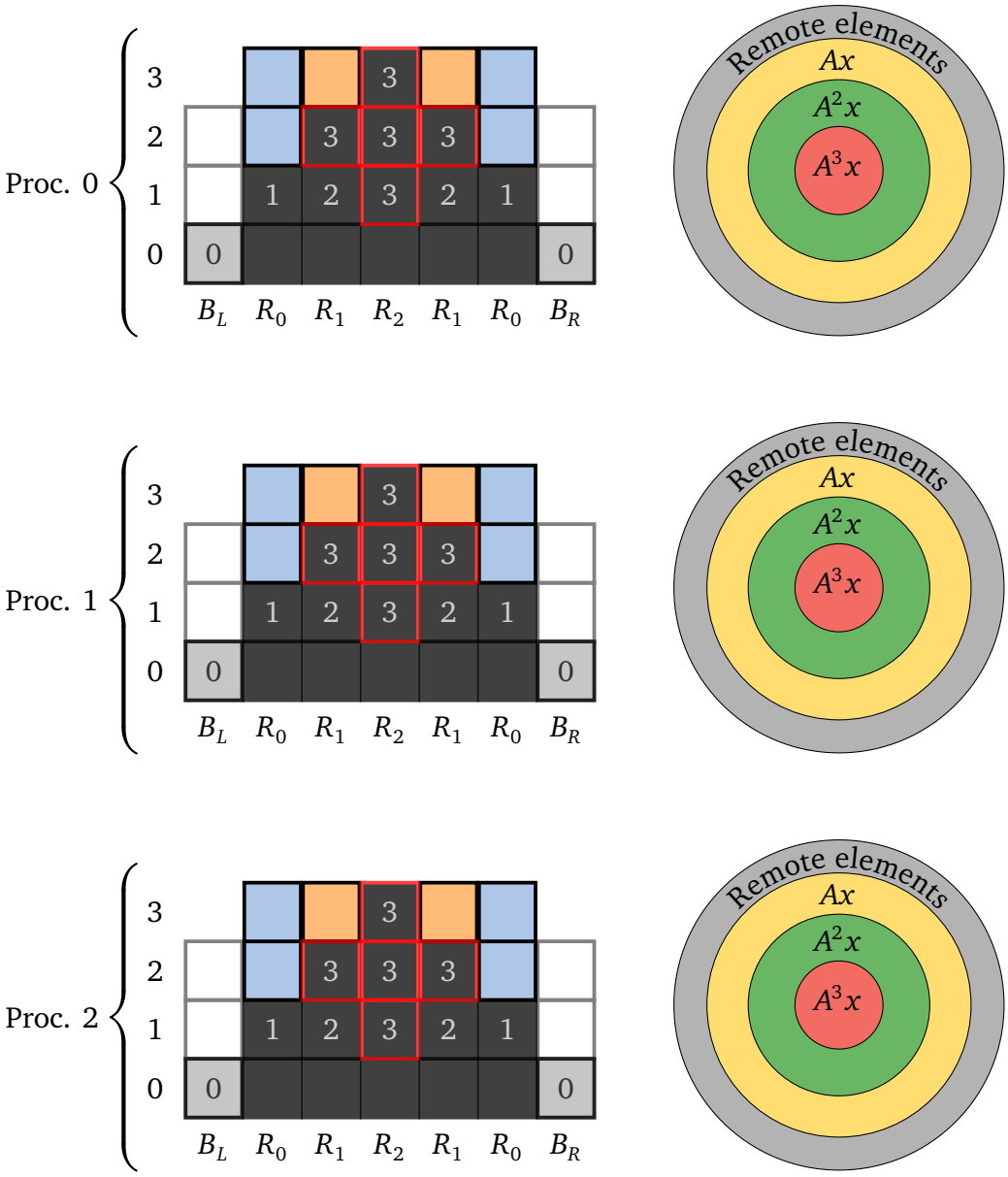


Figure 19: Local LB-MPK: Full Computation of Inner-most Ring

3.3.4 MPI Post-Computations

As mentioned before, this is the "clean up" phase in which we do the computations not taken care of by the MPI Pre-Computation phase or the Local LB-MPK phase. In Figure 19, this is seen as the boxes which are not yet blacked out. This region of the ring diagrams which have not yet been computed are guaranteed to always be right triangles, and so the pattern of computation will not change depending on the matrix structure or power p_m. The same thing cannot be said of the MPI Pre-Computation phase or the Local LB-MPK phase.

In regards to performance, this phase is more interesting than the MPI Pre-Computation phase of Section 3.3.2, because (potentially) more than one iteration of the communication / computation cycle is required. Since multiple communication routines are likely invoked, the "lowest fruit" for optimization resides here (i.e. overlapping communication and computation). For the pattern of computation, we would like to raise the rings in an iterative manner, *one power at a time*, starting with the outermost rings and then moving to the innermost rings (seen in Figures 20 to 23). The role of each cycle is to fully compute one (symmetrical) "diagonal" in the ring diagram, as shown in Figure 21. As explained in Section 3.2 and [2], this diagonal computation pattern is key for allowing data reuse. Preceding this diagonal computation pattern is the MPI communication routine, in which we fill the dependencies of the diagonal. This is what is meant by the "communication / computation cycle". We emphasize here that a communication routine will always precede a computation.

The pseduocode is shown in Algorithm 9. Line 1 signals that if we are just computing $y \leftarrow Ax$, than there are no post computations to do. Similar to MPI Pre-Computations (Algorithm 8), MPI Post-Computations (Algorithm 9) needs to mind the parity of `pow+mpiRingIdx` (i.e. line 6 and 7).

Taken all together, the figures of Sections 3.3.2 - 3.3.4 illustrate the main idea of DLB-MPK: to shift the load of computation ahead in time to better utilize the cache, while shifting the load of communication further to the end.

Algorithm 9: MPI Post-Computation Phase

```
Input  : int distFromRemotePtr[p_m + 1];
         commFuncType commFunc;
         commFuncArgType commArgs;
         spmvFuncType spmvFunc;
         spmvFuncArgType spmvArgs;
Output: double x[N_r + N_p];
         double y[N_r + N_p, p_m];
1  if p_m > 1 then
2      for pow ← 0 to p_m do
3          p_m ← size(distFromRemotePtr) - 1;
4          [x,y] ← commFunc(commArgs,pow);            // halo exchange
5          for mpiRingIdx ← 0 to p_m − pow do
6              int in ← (pow+mpiRingIdx % 2) ? x : y;
7              int out ← (pow+mpiRingIdx % 2) ? y : x;
8              int sL ← distFromRemotePtr[mpiRingIdx];
9              int eL ← distFromRemotePtr[mpiRingIdx+1]-1;
10             out ←spmvFunc(sL,eL,in,out,spmvArgs.crsArrays);
11         end
12     end
13 end
```

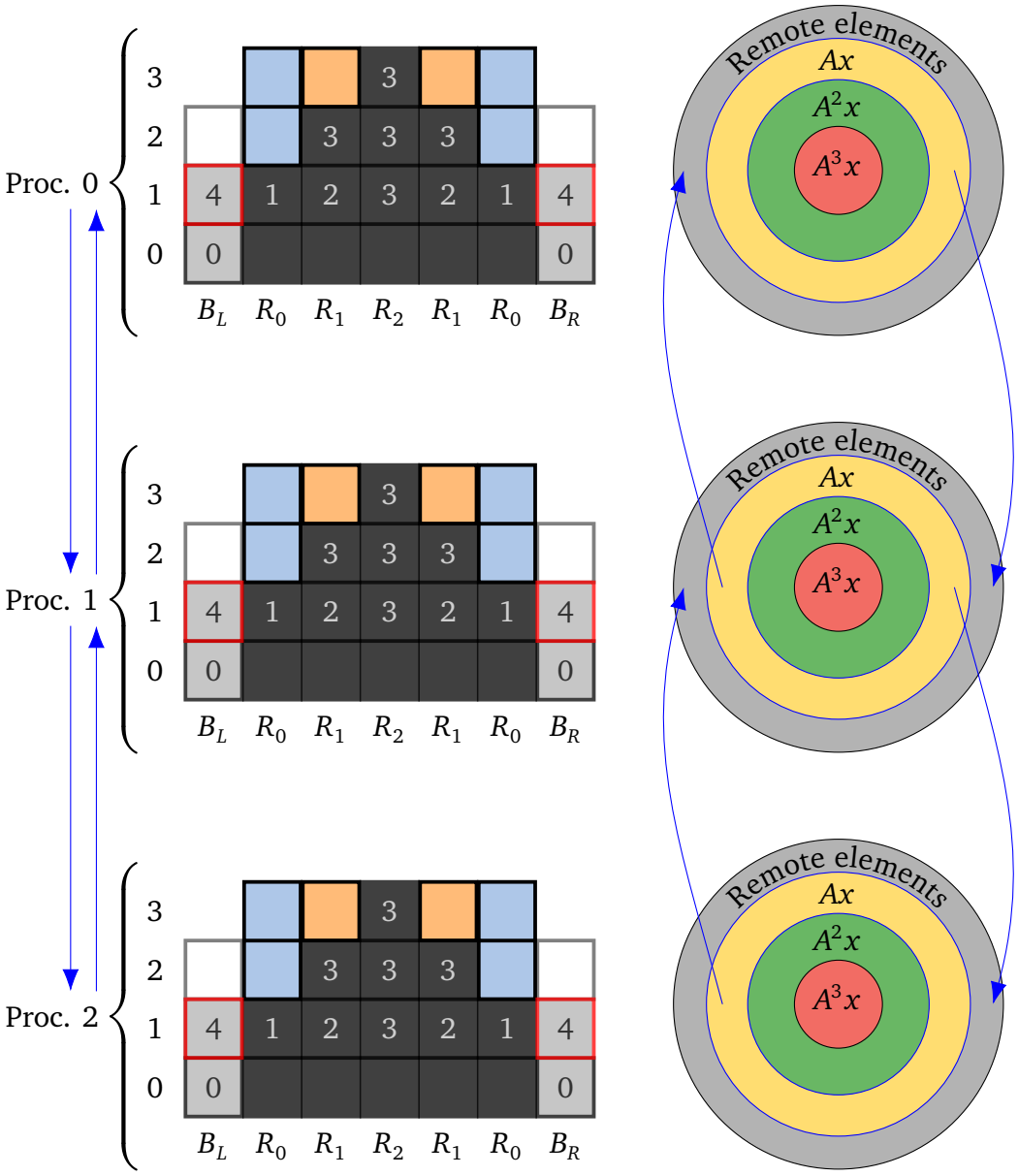


Figure 20: MPI Post-Computations: First Post Communication

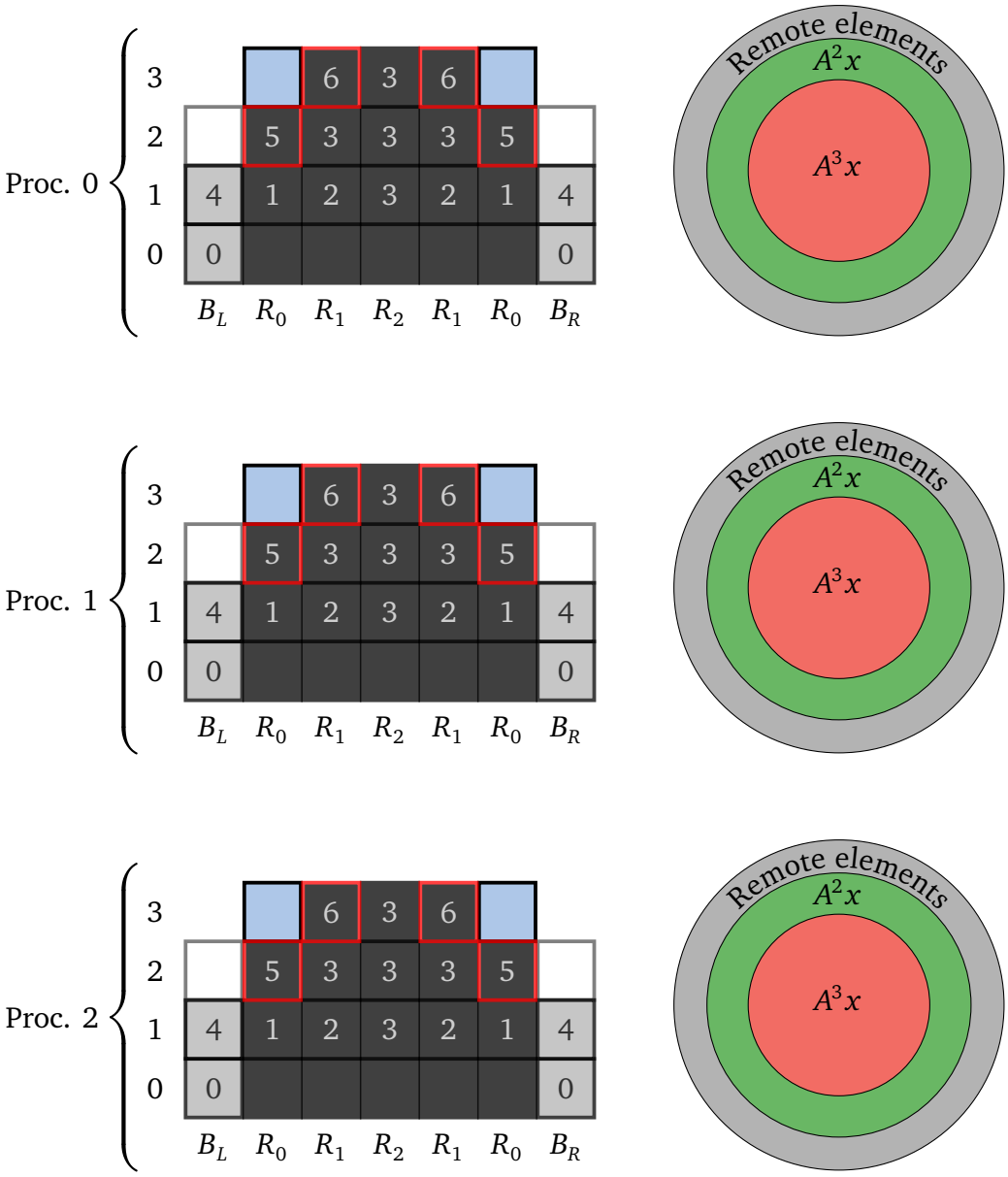


Figure 21: MPI Post-Computations: First Post Computation

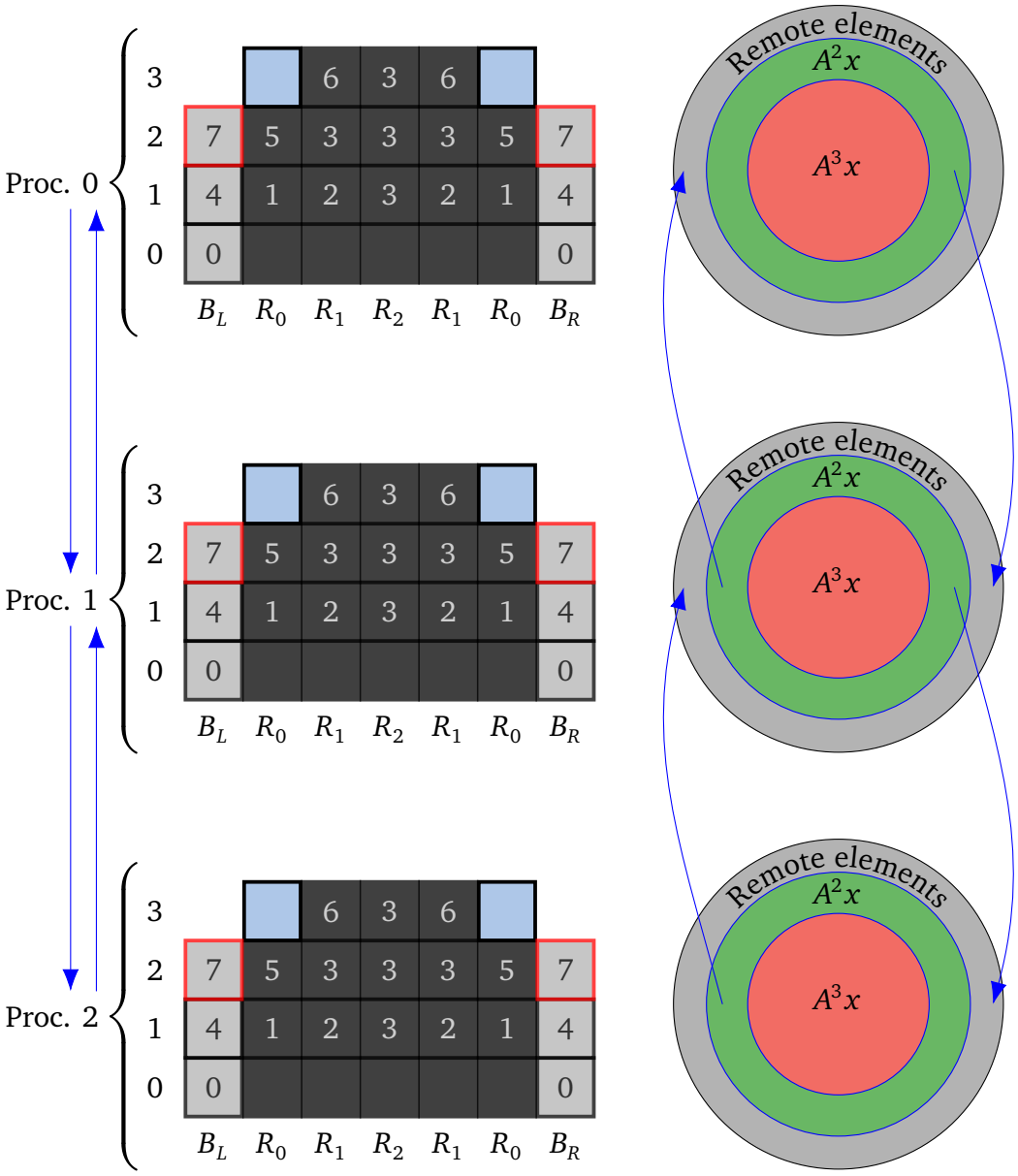


Figure 22: MPI Post-Computations: Second Post Communication

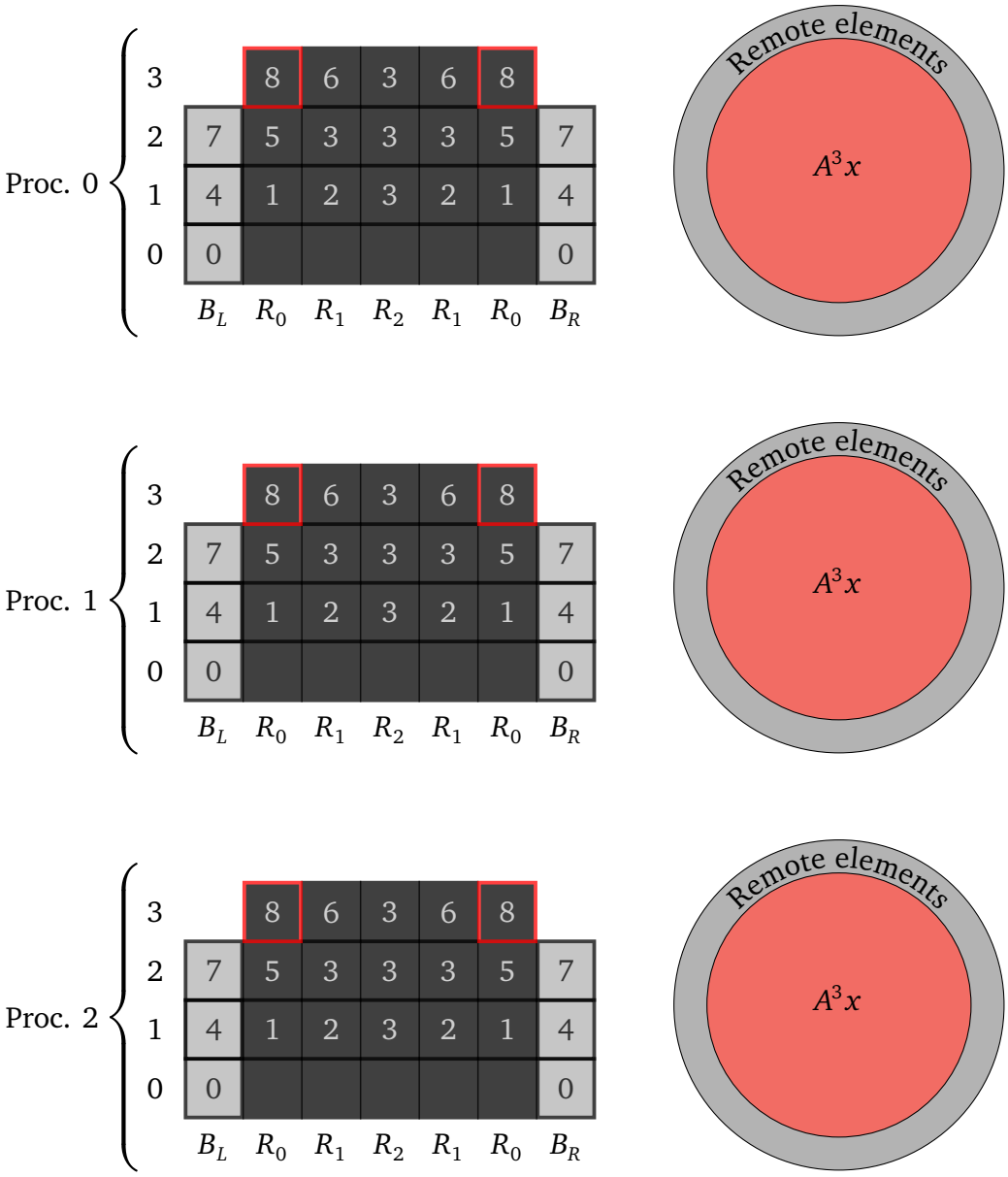


Figure 23: MPI Post-Computations: Second Post Computation

4 Results

Using the Intel Xeon "Icelake" Platinum 8360Y 32-core CPU "Fritz" cluster computer [2], we will examine the performance of DLB-MPK and how it compares to TRAD.

First, we briefly discuss the ccNUMA architecture and cache hierarchy used by Fritz, and how it differs from our parallel execution model used thus far (see Section 2.1). While we are on the topic of architecture, we also measure the bandwidth in order to connect our observed performance with the Roofline Model from Section 2.4.

Then, we perform a parameter study in order to better understand the influence of the various parameters on performance on a few select matrices from the Suite Sparse Matrix Collection [9]. All benchmark matrices used in this section can be found in Section 6. We will initially look at how the graph partitioning method impacts the overhead introduced by MPI, particularly the percentage of total rows that need to be communicated between MPI processes (halo elements). We will then scan various cache sizes and powers p for the calculation of $A^p x$ for a fixed hardware configuration, as was similarly done in [2]. As a final parameter study, we allow the allocated hardware resources to grow as our problem size remains the same (i.e. "strong scaling") and focus on the performance yielded by various powers p. Such a study could be performed by a user of DLB-MPK when tuning for the optimal p for their application.

Afterwards, we give a concise single-node performance summary of how DLB-MPK compares to TRAD with a large set of matrices in a similar fashion to [2]. Lastly, we will examine some inter-node strong scaling and weak scaling (i.e. allocated resources grow proportionally with the problem size) results, paying special attention to the overheads introduced by DLB-MPK.

Remark. For the remainder of this thesis, all benchmarks and results are validated against the Intel Math Kernel Library[3].

AVX-512 SIMD instructions on 64 bit double precision floating point numbers and standard 32 bit integers are used. This corresponds to a single register being able to process 8, 64 bit double precision floating point numbers or 16, 32 bit integers at once. While this is not imperative for SpMV and MPK results on the Icelake architecture (see Section 2.4 for reasoning), it may provide some performance benefit, especially when the working set is small enough to fit into a cache [17]. The compiler we use is Intel's `mpiicpc` with compiler flags `-Ofast -xHOST` on the OS AlmaLinux 8.7.

[2] https://hpc.fau.de/systems-services/documentation-instructions/clusters/fritz-cluster/

[3] https://www.intel.com/content/www/us/en/develop/documentation/oneapi-mkl-dpcpp-developer-reference/top.html

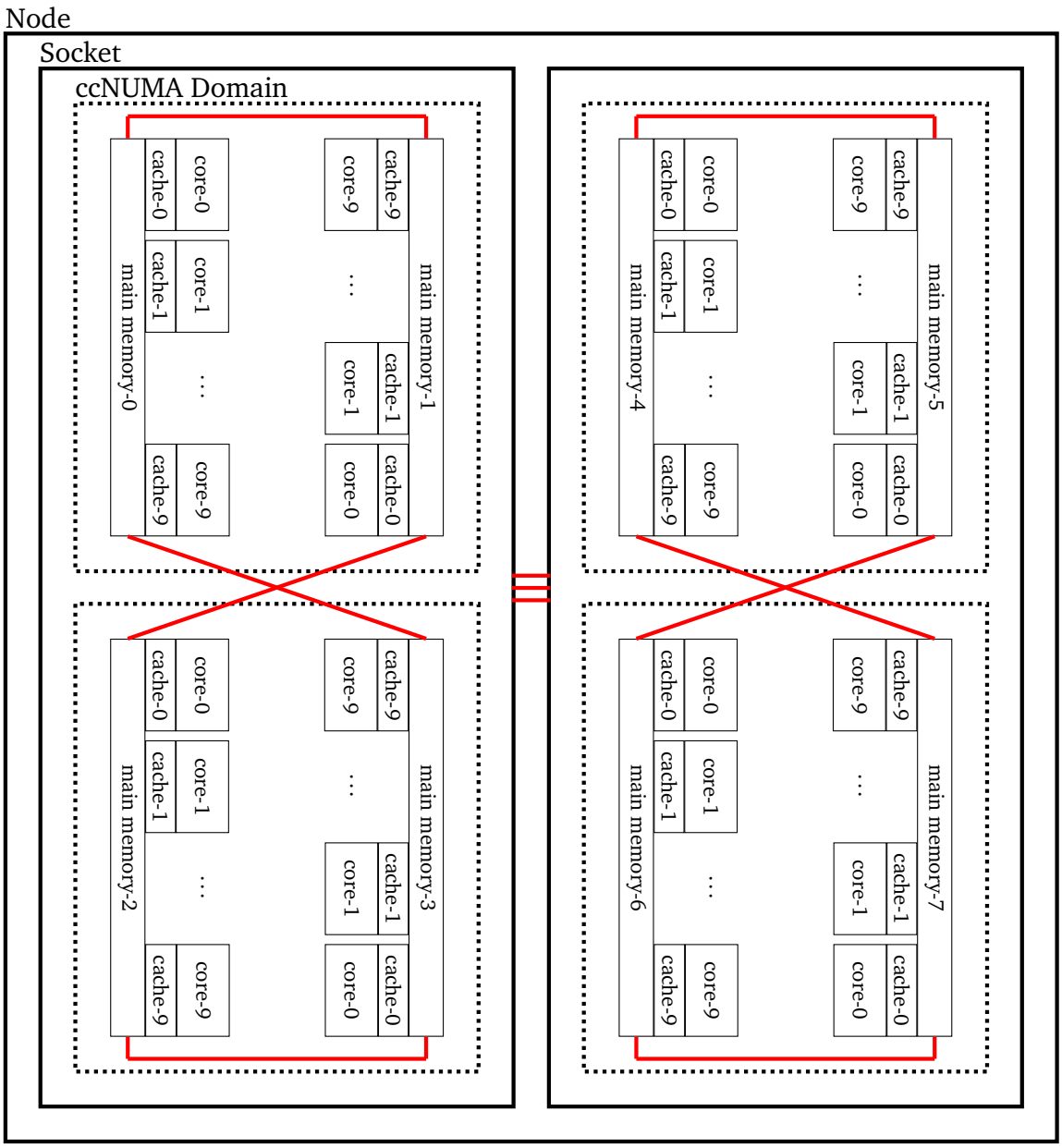


Figure 24: Dual Socket Fritz Node, each socket containing 2 ccNUMA Domains

4.1 Testbed

The "Fritz" system we will be using for our benchmarks is a 992 node cluster computer. A node itself can be considered a "computer", logically separate from the other nodes. Each node contains two CPU sockets, each containing 32 cores. Each socket is internally separated into two partitions, called "ccNUMA domains", to which each has 18 cores assigned (i.e. "Sub-Numa Clustering" is on). Simultaneous multi-threading is turned off so that one thread of execution corresponds to one physical core on the CPU.

The difference between ccNUMA and UMA is that now memory is physically distributed. Yet, this is still a shared memory architecture, so to the programmer, there is no difference. It appears as though memory is still one contiguous block. In other words, memory is physically distributed but still logically shared. A ccNUMA architecture can be thought of as being "built of" UMA architectures. See a simplified Fritz node in Figure 24 for reference. The red paths between address spaces denote a bus connection.

But what does one gain from logically partitioning the cores of each socket into ccNUMA domains? With the added topological complexity of ccNUMA versus UMA from before, now performance depends a great deal on *where* data is located on the node. A programmer writing performant code would need to be aware of this, and manage data accordingly. The main reason behind this architecture style is that it provides scalable bandwidth for large processor counts. Since SpMV and MPK are memory-bound kernels, this is especially relevant for us. For more details, see Chapter 4 of [12].

We need to briefly discuss some aspects of Icelake's cache system before moving to the results. Until now, we have not been distinguishing between different levels of the cache system in our parallel execution model, for simplicity. Icelake's cache system is a hierarchy, consisting of three levels. This terminology should not be confused with the "graph levels" of Section 3.1, and are a different concept entirely. In the order of increasing size (and decreasing bandwidth), these levels are referred to as L1-data (as opposed to L1-instruction), L2, and L3. The L1 and L2 cache levels are private to each CPU core, while the L3 level is shared. The sizes of the respective levels are 32 KB, 1.25 MB, and 54 MB [4]. What we are concerned with in this work are levels L2 and L3, as RACE can block only for these cache levels.

Icelake's cache hierarchy is "exclusive", meaning that the different levels of cache can be understood as "non-overlapping". Taking a simplified view, the total aggregate cache size which we can block for on one Icelake socket is the sum of the 36 private L2 caches and the single L3 cache

$$(36 * 1.25)\ \text{MB} + 54\ \text{MB} = 108\ \text{MB}. \tag{5}$$

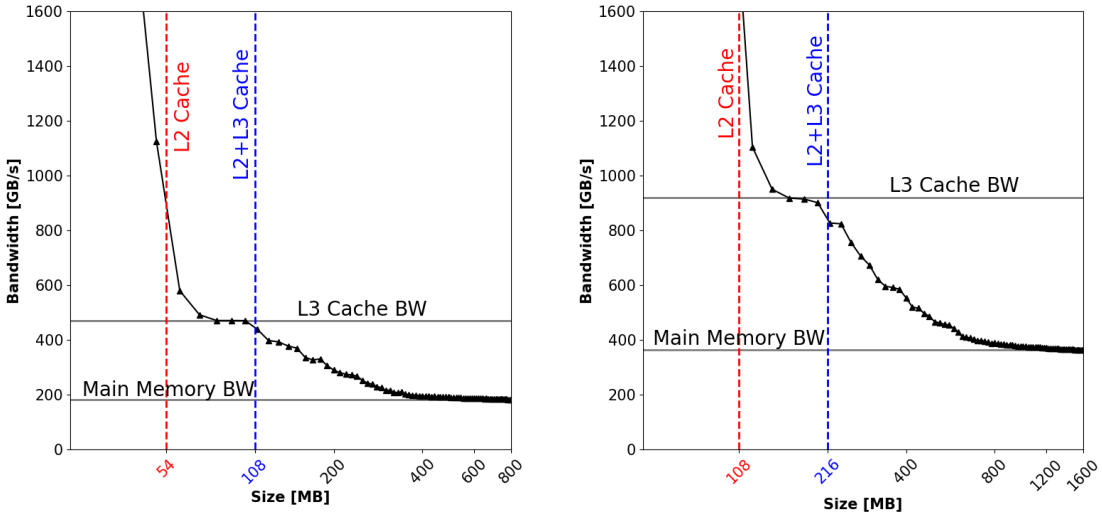
Recall from Section 2.4 that SpMV and MPK are main memory-bound kernels, and from Equation 1, their performance is limited by

$$P = \frac{b_s}{6 + 4\alpha + 10/N_{nzr}}$$

[4] https://www.cpu-world.com/CPUs/Xeon/Intel-Xeon%208360Y.html

where b_s is the bandwidth, α essentially quantifies data traffic caused by accessing the RHS x vector and N_{nzr} is the average number of non-zero elements per row of the matrix.

If we fix a matrix (i.e. fix N_{nzr} and α), Equation 1 tells us that the only way to improve performance is to increase b_s in this main memory-bound regime. This is exactly what cache blocking aims to do. Figure 25 shows the (load-only) bandwidth when loading data sets of increasing sizes. Notice that the logarithmic scale of the x-axis in Figure



(a) Single Socket Bandwidth

(b) Single-Node Bandwidth

Figure 25: Load Benchmark: Bandwidths from L3 Cache and Main Memory

25b is doubled, to account for the two Icelake sockets present on a single Fritz node.

The main memory bandwidth is estimated as the limiting bandwidth for large messages. The L3 cache bandwidth can be estimated at these "plateaus" seen in Figure 25. This data was collected using the `load_avx512` microbenchmark, apart of the `likwid-bench` tool from the `likwid` tool suite [30]. All ccNUMA domains are given an equal workload for the microbenchmark. For Figure 25a, the bandwidth to main memory is estimated at 182.3 GB/s, and the bandwidth from L3 cache is estimated at 470.6 GB/s. For Figure 25b, these estimates are 363.2 GB/s and 920.8 GB/s respectively.

The vertical lines illustrate the corresponding cache boundaries, assuming Equation 5. Looking at 25, the loaded data that comes from the left of the dashed red line is assumed to have come from L2 cache, and from the left of the dashed blue line from L2 and L3 cache. In terms of bandwidth, we still see some caching benefits for datasets larger than L2+L3 cache (i.e. to the right of the dashed blue line). This is shown as the sloping line to the right of the blue dashed line. This is due to the fact that Intel has a high-quality "dynamic replacement policy", which makes intelligent use of the cache even for large data sets [3]. This is one instance of the "LRU"-like cache eviction strategies mentioned in Section 3.2.

The bandwidth plot of a single ccNUMA domain is not depicted in Figure 25, as it cannot be said for certain if it would share L3 cache with the other ccNUMA domain in the socket. Thus, the bandwidths cannot be ascertained easily.

The main takeaway from Figure 25 is that, for both a single socket and a single-node, L3 cache bandwidth is roughly 2.5x greater than bandwidth from main memory. So even with perfect cache blocking and minimal overheads from MPI, we can expect DLB-MPK performs at most x2.5 faster than TRAD. Other architectures could have a much higher ratio of L3 cache bandwidth to main memory bandwidth. For example, AMD's "Epyc Zen2" (Rome), has L3 cache bandwidths roughly 17x greater than the main memory bandwidth [2].

4.2 Benchmarks and Parameter Study

The main goals of the parameter study are to quantify and understand the overhead introduced by MPI, as well as understanding how various parameters influence performance.

The way in which we partition our matrix over the MPI processes (or more specifically, the graph representing the matrix) influences the overheads in the communication scheme. We first describe the partitioning methods used, then how they incur this overhead, and what can be done to minimize it. Figure 26 shows three different ways in which we can partition a fully connected six-point graph.

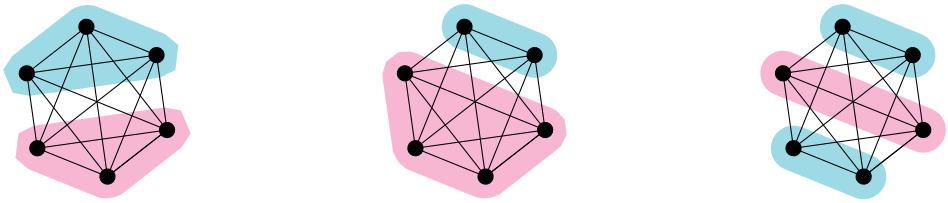


Figure 26: Various Graph Partitionings

Let the two partitions be assigned to two MPI processes. From left to right, we briefly describe what these partitionings are meant to represent.

1. On the left, this figure gives an example of a "by-row" partitioning, in which each MPI process is given control over a contiguous (near) equal number of rows (vertices). These vertices are collected by scanning the matrix from top to bottom, and evenly segmenting the rows between processes.

2. In the center, this figure shows a "by-nnz" partitioning in which each MPI process should contain a contiguous portion of the graph such that the total number of non-zero elements associated with those vertices is (near) equal. The smaller partition in this figure has only two vertices, but would be associated with just as many non-zero elements as the larger partition. Compared to the "by-rows" method, the structure of the matrix would determine if this would yield a more balanced workload over the MPI process.

3. In the rightmost figure, the graph is partitioned in a non-contiguous way. This figure represents the "by-metis" partitioning method, in which we call the METIS library[5], specifically the routine `METIS_PartGraphKway` (where "K" corresponds to the number of MPI in our case), to partition our graph for us. Our goal with the use of METIS is to minimize communication between processes and optimize load balancing. METIS achieves this by using sophisticated heuristics within a multilevel graph partitioning paradigm. Since MPI requires that contiguous messages be communicated, this method requires additionally a "global matrix permutation" (in addition to the local BFS permutations described in Section 3.1) in order to yield such a contiguous partition. Unless otherwise stated, we use this partitioning method for the remainder of the thesis.

Recall from Section 2.6 that SpMV, and by extension MPK, require "buffer padding" in the RHS x vectors in order to receive the needed dependencies incoming from other MPI processes. The way in which we partition our graph dictates how many vertices have dependencies with vertices across an MPI boundary. So, the amount of x vector buffer padding needed by each MPI process is a direct result of the graph partitioning method used. See Table 1 for a summary of halo row percentages for matrices `ML_Geer`, `Fault_639`, and `crankseg_1`, using the three graph partitioning methods described. Section 6 contains more details on these matrices.

Matrix	N_r	MPI Procs	Percentage Halos [%]		
			by-rows	by-nnz	by-metis
ML_Geer	1_504_002	2	0.4	0.4	0.4
		4	1.5	1.5	1.3
		8	3.2	3.2	2.6
		16	6.7	6.8	4.4
		32	13.8	13.8	6.8
		64	27.8	27.8	10.3
		128	55.9	55.9	15.4
Fault_639	638_802	2	5.3	5.3	3.2
		4	13.9	13.8	7.6
		8	30.4	30.5	12.2
		16	63.1	63.4	18.9
		32	128.0	128.5	26.7
		64	199.4	199.6	36.9
		128	237.4	237.3	49.2
crankseg_1	52_804	2	95.5	96.9	3.8
		4	191.5	200.5	13.0
		8	330.2	302.5	42.7
		16	421.3	420.6	72.8
		32	522.1	513.6	112.0
		64	621.7	643.8	190.5
		128	769.3	787.0	310.2

Table 1: Halo Row Percentages

[5]https://github.com/KarypisLab/METIS

While the main focus of DLB-MPK is on the distributed memory setting, there are also benefits of using DLB-MPK in the shared memory setting as compared to the original LB-MPK. This benefit is in the form of simplifying proper ccNUMA utilization and data placement control. Previously, there were serious difficulties in placing data correctly across the ccNUMA domains with LB-MPK, and support was eventually dropped. Now, one can use DLB-MPK and pin MPI processes to ccNUMA domains, greatly simplifying data accesses even though a single node (or socket) is in the "shared memory context" as we've been describing. The term "pinning" (or "binding") refers to the way in which a system distributes work across available resources.

In Figure 27, we fix a hardware configuration of two nodes and scan various powers p and cache sizes C as is done in [2]. Two nodes were chosen, as it exemplifies the distributed nature of DLB-MPK, while not providing so much cache that all associated matrix data fits into cache automatically, in which case there would be no benefit of using DLB-MPK. We pin MPI processes to the 8 ccNUMA domains and pin OpenMP threads to 18 physical cores on each ccNUMA domain in a "close" manner in which worker threads are close to the master in contiguous partitions (also called a "fill" pinning). Unless stated otherwise, we fix the recursion depth of RACE at 8.

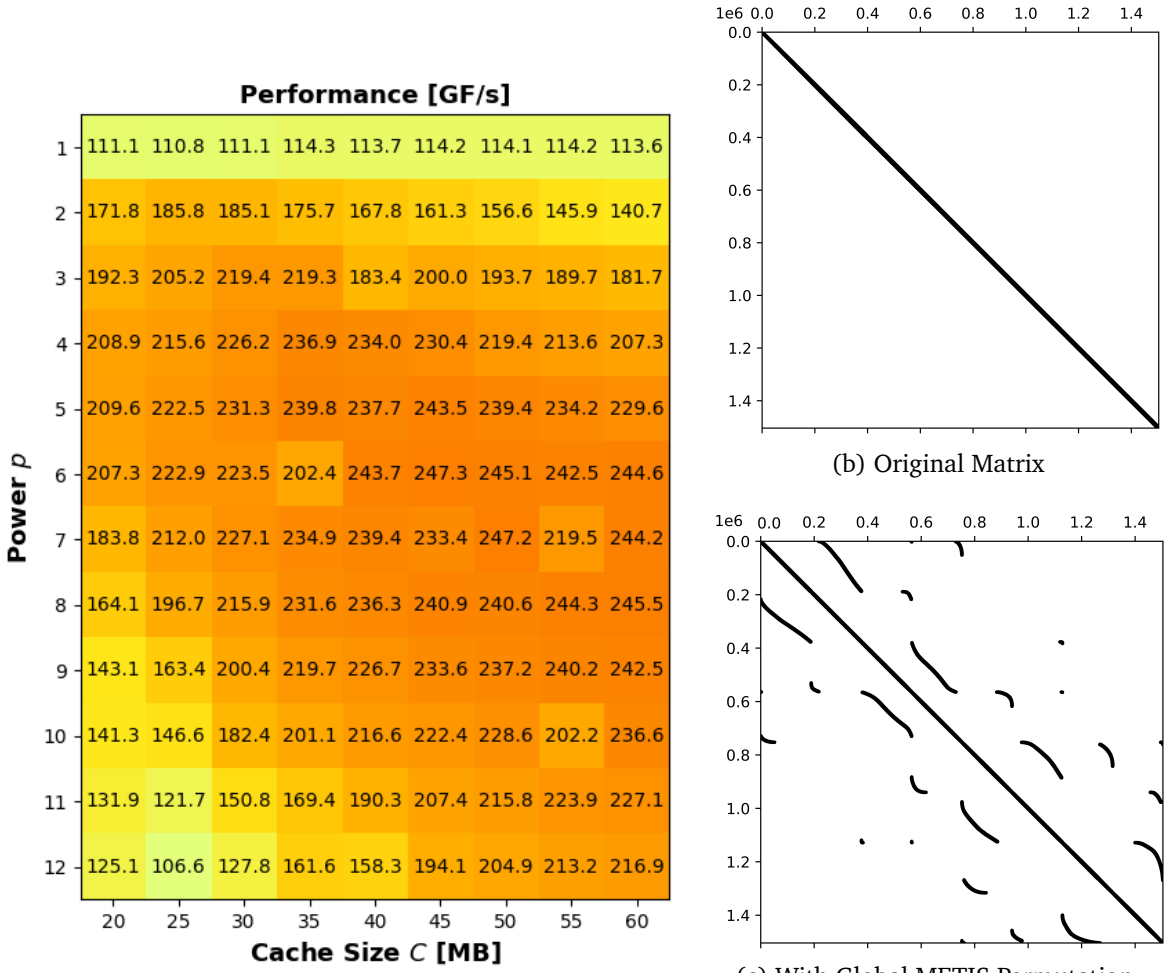
In Figure 27a, we can see how the two parameters, power p and cache size C, influence the performance of DLB-MPK on `ML_Geer`. The global METIS permutation for `ML_Geer` and 8 MPI processes is shown in Figure 27c. We can see there is a "sweet spot" around $p = 6$ and around $C = 50$, after which performance starts to degrade for higher p and C. From Section 4.1 we know that a single Icelake socket has 108 MB L2+L3 cache. So when pinning MPI processes to ccNUMA domains, we would expect an optimal C to be around 50 to 55. The value of C should correspond to the amount of available cache (in this case L2+L3 cache).

A user of DLB-MPK would tune these two parameters in order to achieve the best possible performance for their use case. Notice that the DLB-MPK performance for $p = 1$ stays roughly constant as cache size grows. This corresponds to our intuition since computing $y \leftarrow Ax$ does not make use of any cache blocking as A is loaded from main memory only once.

We now fix a cache size $C = 50$, and examine how the performance of DLB-MPK varies with $p \in \{1, \ldots, 8\}$ for an increasing amount of resources. The reason we stop at $p = 8$ is because the most performant power typically appears before then, regardless of C or matrix structure. After which point, performance would only degrade. Say the power p which yields the best performance is $p = k$. If a user or library requires a power of $A^p x$ higher than $p = k$ (i.e. for use in some larger algorithm), then this vector can be stored and successive applications of this $A^k x$ can be made. For example, if an algorithm requires $A^{15}x$, but $p = 4$ yields the most performance, we could just compute

$$A^{15}x = A^4x * A^4x * A^4x * A^3x.$$

The goal here is to understand the performance gained from cache blocking through the parameter p, not the benefit of the improved data accesses on the RHS x vector through the local symmetric BFS permutations (which RACE must do implicitly). We



(a) Performance Heatmap

(b) Original Matrix

(c) With Global METIS Permutation

Figure 27: 2 Node `ML_Geer` Parameter Study

have to be careful not to conflate the two. Therefore, we take the baseline performance as the maximum of TRAD executed on both the original matrix and local symmetric BFS permuted matrix. The reason that a BFS permutation might yield a lower performance for TRAD is due to recursion negatively affecting data locality on the RHS vector [2].

For the purposes of this work, we define "speed-up" as the improvement of performance of DLB-MPK compared to TRAD:

$$\text{Speed-up} := \frac{\text{TRAD execution time}}{\text{DLB-MPK execution time}} \tag{6}$$

Figure 28 scans powers $p \in \{1,\dots,8\}$ for an increasing amount of ccNUMA domains $\{1,\dots,8\}$ over two Fritz nodes. These ccNUMA domains are added in a compact manner, filling a socket before moving to the next socket in the node, and similarly filling one node before moving to the next. The vertical grey line in each plot denotes the node boundary.

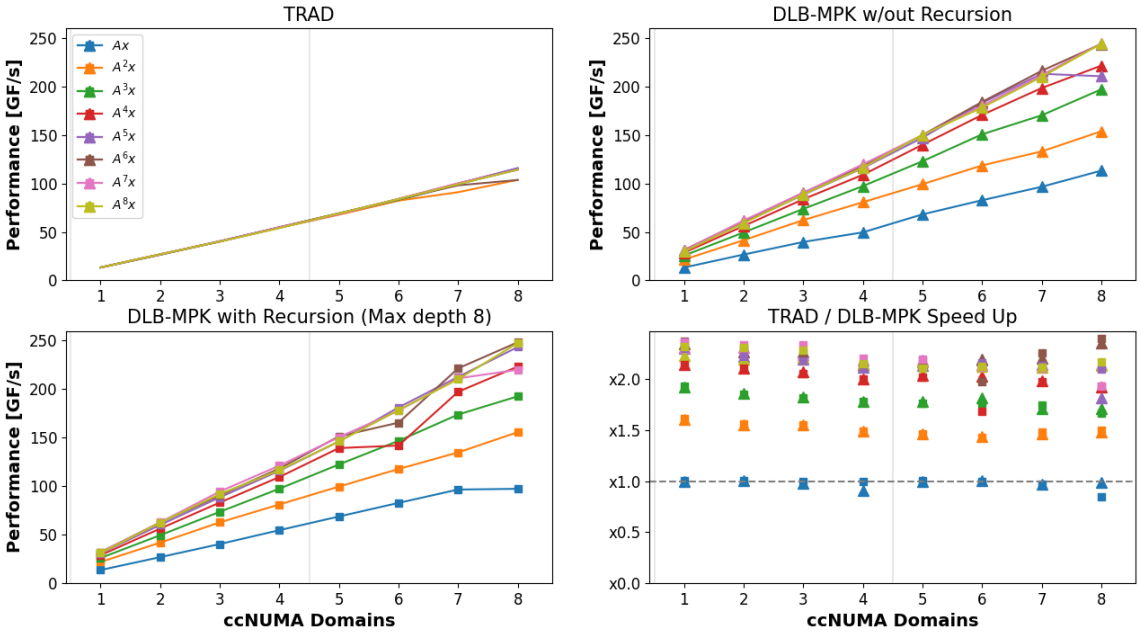
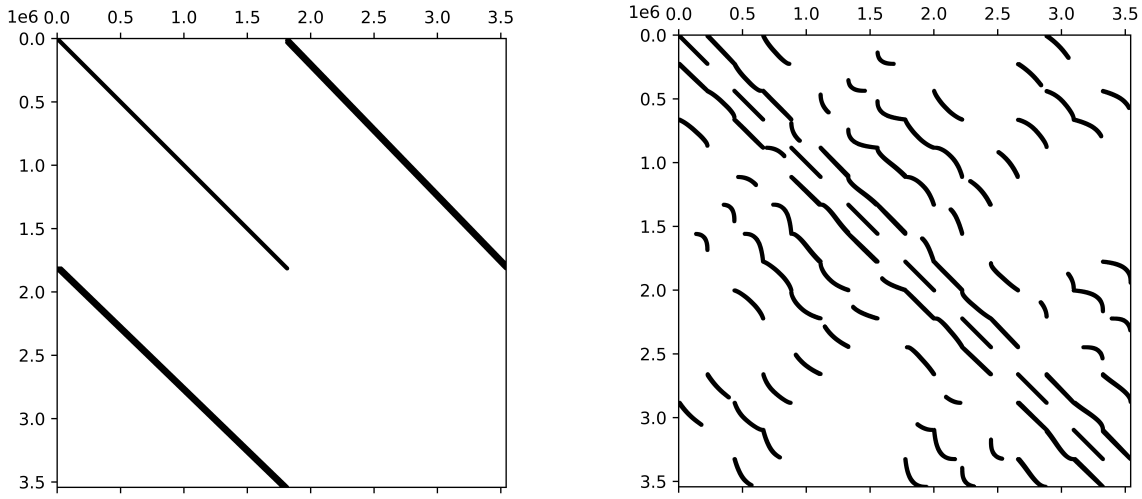


Figure 28: Various Powers of `ML_Geer`, 2 Nodes

In the top left plot of Figure 28, we have the performance of the traditional MPK implementation, TRAD. In this instance, TRAD was more performant with the local BFS permutations, and so that is taken as the baseline. The first observation we can make about the top left plot is that performance is the same, regardless of the power $A^p x$ being calculated. This is of course because no cache blocking is being used. The second observation we could make is that performance becomes less stable at around 7 or 8 ccNUMA domains. The reason for this is that the matrix is partitioned into small enough segments such that we can start to see automatic (i.e. unintentional) caching effects. Therefore, communication and other internal processes have a much larger impact on performance, and performance variation is likely to be seen. In the top right plot of Figure 28, we have the same p+hardware scaling scheme, but with DLB-MPK in place of TRAD, and no recursion. These results correspond to what we expected. The benefit of cache blocking for this matrix is sizable for A^2x, A^3x, and A^4x, but tapers off after A^5x without much performance gain or degradation for higher powers. The performance of DLB-MPK without recursion is indicated with a triangle marker. In the bottom left plot of Figure 28, recursion is used, but it does not appear to improve performance for this matrix. The performance of DLB-MPK with recursion, which agrees with the results in Figure 27a, is indicated with a square marker. Finally, in the bottom right plot of Figure 28, we take the ratio of the performance of DLB-MPK and TRAD to obtain the "speed-up". Notice that, for computing Ax there is no speed-up, and in most cases, DLB-MPK performs slightly worse than TRAD due to the inherent overheads that come with the initializations internal to DLB-MPK (i.e. Section 3.3.1). The fact that the square and triangle markers are usually overlapping indicates that recursion does not improve performance for `ML_Geer`. Finally, notice that the speed-up tops out at around x2.4, which is what we expect the upper bound on speed-up to be from Section 4.1.

See Figure 30 for another example of how matrix structure affects performance. We provide a similar analysis to Figure 28 on the matrix `nlpkkt120`. The sparsity pattern of `nlpkkt120` is given before and after the global METIS permutation in Figure 29.



(a) Original Matrix

(b) With Global METIS Permutation

Figure 29: `nlpkkt120` before and after permutation

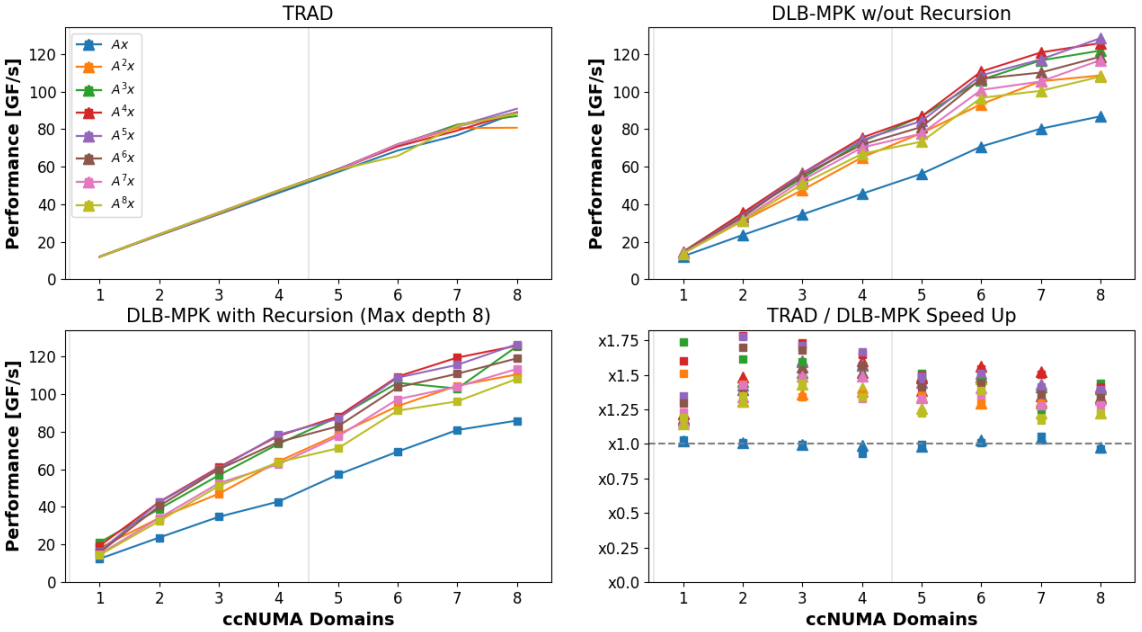


Figure 30: Various Powers of `nlpkkt120`, 2 Nodes

There are several observations one can make about Figure 30. First, notice that the *Ax* computation does not exhibit any speed up, as in Figure 28. Recursion plays a larger role in performance when compared to Figure 28, at least for smaller ccNUMA counts. This can be seen in the bottom right speed-up plot, where the square markers are higher than the triangle markers. As we will see in Section 5, recursion typically

plays a bigger role in performance for large matrices (subject to matrix structure of course). Unlike Figure 28, performance begins to degrade after A^5x computations for larger powers p. Lastly, notice that the speed-up does not reach the predicted x2.5 bandwidth limit that `ML_Geer` does. This is due to the poor matrix structure of `nlpkkt120`.

Similar to [2], we give a performance summary of TRAD versus DLB-MPK in Figure 31. For the baseline, we again take the maximum of TRAD's performance, executed on both the original matrix and local symmetric BFS permuted matrix. Fix a hardware configuration of a single Fritz node, i.e. 4 ccNUMA domains. We order the matrices in terms of size, from smallest to largest.

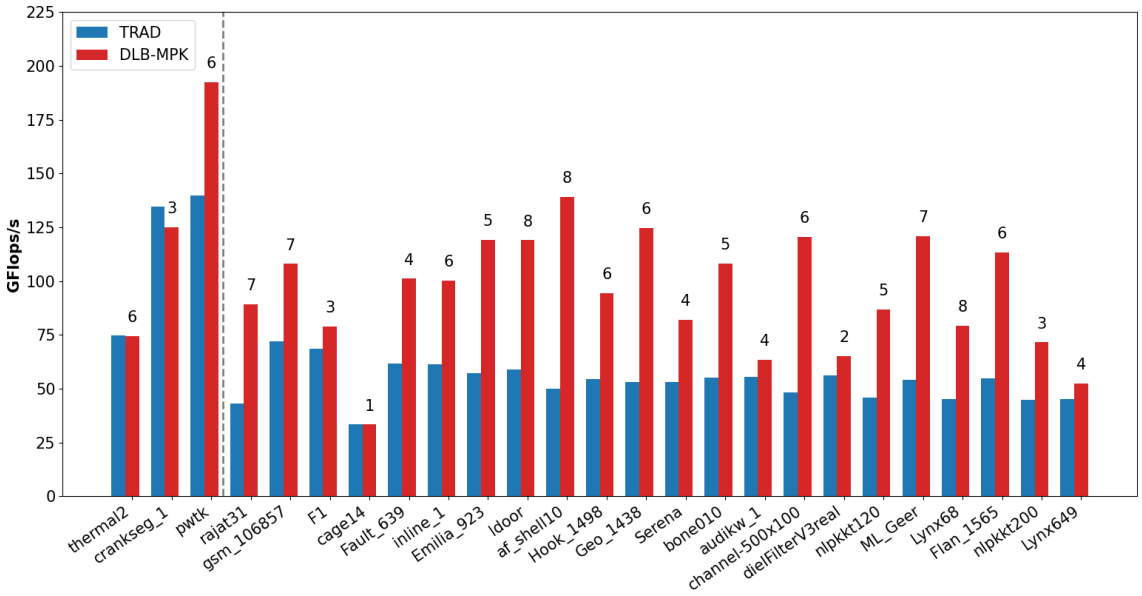


Figure 31: Single Node Performance Summary

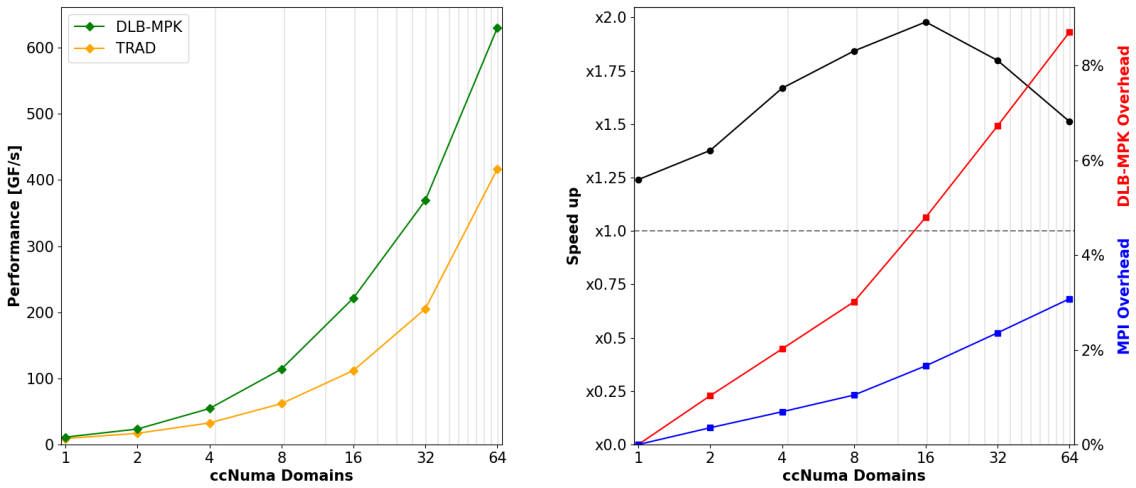
The grey dashed vertical line in Figure 31 denotes the cache limit of a single node of Fritz. The matrices `thermal2`, `crankseg_1`, and `pwtk` all automatically reside in-cache (after the initial load from main memory). Notice that TRAD performs roughly the same for all benchmark matrices that do not automatically reside in cache. Just as in the shared memory OpenMP only LB-MPK in [2], DLB-MPK does not show significant speed-up over the baseline with smaller matrices in Figure 31. Speed-up is more noticeable after `Fault_639`. In this region, associated matrix data is large enough to not automatically fit into cache (partially or entirely). Since we are trying to determine the benefit of our distributed cache-blocking scheme, the matrices should ideally be large enough as to not fit into cache on their own.

We proceed with both a strong and weak scaling study, in order to better understand the performance of DLB-MPK for larger processor counts. The ScaMaC (Scalable Matrix Collection) library[6] provides routines for matrix generation of scalable size and for a wide variety of matrix structures. These structures originate from real-

[6]https://alvbit.bitbucket.io/scamac_docs/index.html

world and scientific research applications. We will use the "Anderson" pattern from ScaMaC as our benchmark matrix for the strong and weak scaling studies. Anderson, a matrix describing "the motion of a quantum-mechanical particle in a disordered solid". The relevant parameters to us are ones that control the size of the matrix, and so we define $\texttt{Anderson}(Lx, Ly, Lz)$ as being parameterized by three variables representing the dimensions of a cube, Lx, Ly, and Lz, and contains the number rows $N_r = Lx * Ly * Lz$.

For the strong scaling, we fix the matrix $\texttt{Anderson}(600, 600, 600)$. This matrix is not a good candidate for recursively breaking down bulky levels with RACE, so recursion is turned off for strong and weak scaling results. Figure 32 shows the performance for both TRAD and DLB-MPK for the tuned parameters $C = 50$ and $p = 4$. The corresponding speed-up can be seen in Figure 35b. Notice in Figure 35b that for this matrix, peak speed-up is about x2.0 at 16 ccNUMA domains (4 nodes).



(a) Strong Scaling Performance

(b) Strong Scaling Overheads

Figure 32: Anderson Strong Scaling, 16 Nodes

There are two distinct overheads to understand here. The first is one that we have already seen before, the percentage of halo rows to local rows discussed in Table 1. This halo row percentage, or what we are calling it here "MPI Overhead", is not unique to DLB-MPK. The second overhead, the "DLB-MPK Overhead" is unique to DLB-MPK, and is defined as the percentage of rows outside of the "main ring" described in Section 3.3.3. To compute this overhead, we first need to collect the Local LB-MPK Overhead on each process, where we define

$$\text{Local LB-MPK Overhead} := 1 - \frac{\#\text{ Rows in Local Main Ring}}{\#\text{ Local Rows}}. \tag{7}$$

These two metrics are present on each MPI process, and are meant to quantify the domain over which LB-MPK can cache block. This is actually a fairly naive view of

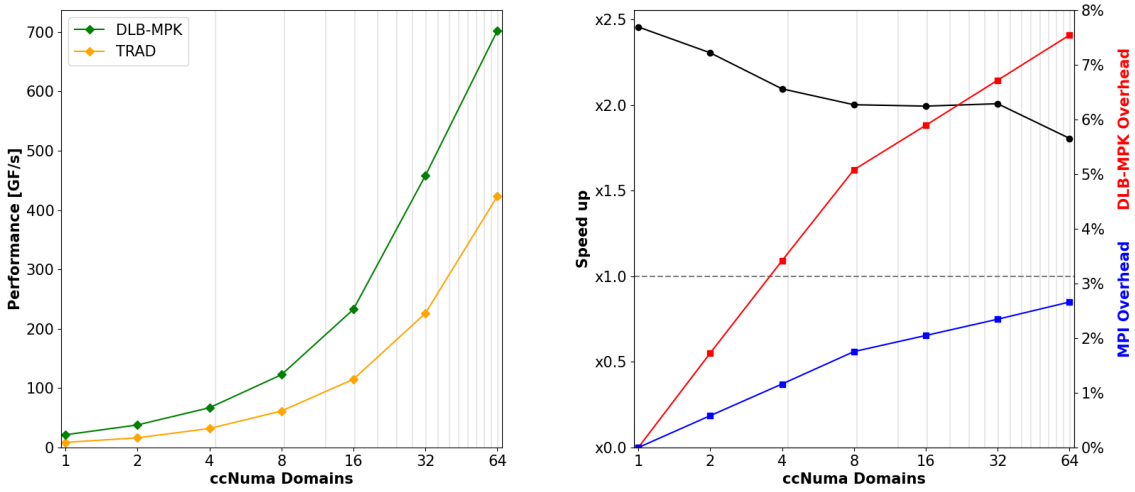
LB-MPK overhead, as there also exists cache blocking in the "remainder/edge" parts of the cache-blocked diamond [2], but it is sufficient for this work.

We can collect the local overheads for each process $\{0,\ldots,N := \#\text{ MPI Procs.}\}$, and normalize them in order to obtain the important global metric

$$\text{DLB-MPK Overhead} := \frac{\sum_{i=0}^{N}(\#\text{ Local Rows}_i * \text{Local LB-MPK Overhead}_i)}{N_r}. \tag{8}$$

Remark. For A^2x, it will always be the case that DLB-MPK Overhead = MPI Overhead. This is obvious from the definition of `distFromRemotePtr[]` from Section 3.3.1.

Weak scaling is the process by which we can understand how performance varies with processor count, given a fixed problem size per MPI process. So, in our weak scaling study, we exactly double the amount of work as we also double the allocated resources to do that work. For the weak scaling, we do not fix one matrix size but scale exactly with the MPI process count. See `Anderson` at the end of the Benchmark Matrices of Section 6 for parameter values and matrix data. The values chosen for Lx, Ly, and Lz ensure two things: that each process has enough data such that the local matrices are large enough to be well outside of the range in which parts can be automatically cached, and that we are not at risk of "integer overflow" (as the current implementation of DLB-MPK only supports basic `int` data types which hold a maximum value of 2_147_483_647).



(a) Weak Scaling Performance

(b) Weak Scaling Overheads

Figure 33: `Anderson` Weak Scaling, 16 Nodes

5 Application to Cardiac Arrhythmia Simulations

Cardiovascular disease is the most common cause of death globally, across all ages and backgrounds. Sudden cardiac arrest accounts for approximately 50 percent of cardiovascular deaths [14] [15]. Cardiac arrest happens when rapid and abnormal electrical impulses impede the heart's natural rhythm. Another word for these abnormal electrical impulses is "arrhythmia".

In order for specialists in the field of Cardiac Electrophysiology to prevent sudden cardiac arrest, they need to understand and predict the risk of cardiac arrhythmia in their patients. Recent studies have shown the effectiveness of "in-silico" experiments, as opposed to "in-vitro" studies [28] [4]. While an active area of research for almost 50 years (the "Computing in Cardiology" conference held annually since 1974[7]), these in-silico cardiac arrhythmia simulations (CAS) have been quickly developing in utility as hardware capabilities improve.

A typical CAS use case is to answer the question "Given some representation of a patient's heart, what is the risk of this patient developing cardiac arrhythmia?" Multiple CAS instances are executed in which electrical signals start at various locations, and propagate along the muscle fibers in the cardiac model. If an arrhythmia was observed in these simulations, the patient was deemed to be at risk of developing arrhythmia. The hope is that, once the in-silico experiments mature and are more widely adopted by the medical community, doctors would perform such experiments in a clinical setting instead of the more invasive in-vitro examinations.

However, there still exist major limiting factors to in-silico CAS studies. These simulations are computationally expensive, sometimes taking days to complete (depending on the resolution of the cardiac model). Other limiting factors are irregularities in the cardiac model impeding the electrical impulses, such as scar tissue. For practical clinical use, the speed at which a CAS could simulate the electrical activity in a patient's heart would need to be near real-time, as diagnostics can be highly time-sensitive. This would require a tremendous amount of computing power and algorithmic advancement. This is why CAS codes are a perfect application of HPC, and a great way to show the usefulness of the main contributions of this thesis from Section 3.

In this section, we use DLB-MPK to accelerate LYNX [18], a preexisting CAS code, in the case of CPU-only execution (i.e. without the presence of accelerators/GPUs). First, we explain the basic ideas of LYNX, and how it simulates electrical activity in the heart. Then, we show the speed-up of DLB-MPK compared to the method currently implemented.

LYNX solves the reaction-diffusion equation:

$$\frac{\partial u(\vec{x},t)}{\partial t} + \chi I_{\text{ion}}(t,u,\vec{\phi}) = \nabla \cdot (K(\vec{x})\nabla u)\,. \tag{9}$$

It achieves this through the Finite Volume Method (FVM), a popular variant of the

[7] https://cinc.org/

Finite Element Method. What the FVM essentially does, is divide the domain into discrete "chunks/cells", called finite volumes, over which we can approximate our "state variables". This discretization of the solution space is typically described as a "mesh", which consists of vertices, faces, and cells. Figure 34 is an example of what a mesh might look like for a cardiac model[8].

Equation 9 is known as the "monodomain model", and is very frequently used by researchers to understand the main features of electrical signal propagation within the heart [8]. As our cardiac models live in 3D space, it is understood that $\vec{x} := (x, y, z)$. At the location $\vec{x}$ for a time t, $u(\vec{x}, t)$ describes the "membrane potential" at that point in time and space. The action of a muscle "depolarizing" (i.e. u decreasing) is synonymous with contracting, which we know in this context as a heartbeat.

The "diffusion" part of Equation 9 is very basic in form. The term $K(\vec{x})$ is the conductivity tensor field describing the cardiac muscle structure, which of course depends on the location $\vec{x}$. LYNX uses "transversely isotropic conductivity tensors", so that conductivity only depends on the longitudinal fiber direction. That is, electrical signals are assumed to travel parallel to muscle fibers.

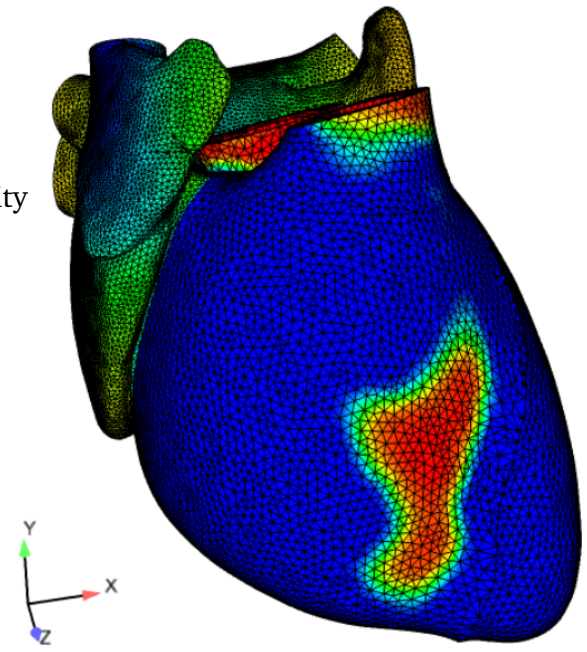


Figure 34: Example Tetrahedral Discretization

The "reaction" part of Equation 9 is more nuanced, as a function of state variables is present. The scalar χ is the ratio of the cell membrane area to the cell volume area. The function $I_{\text{ion}}(t, u, \vec{\phi})$ denotes the total ionic current across the cell membrane. The term $\vec{\phi}$ denotes the vector of state variables that contribute to the evolution of I_{ion} at each point in time. The "ten Tusscher-Panfilov" model [31] is used for the modeling of I_{ion}. One interesting feature of this model is that internal parameters are assigned depending on where the cardiac cell lies within the different muscle wall layers of the cardiac ventricles. That is, the model can distinguish between the: outer "epicardium" layer, the middle "myocardium" layer, and the inner "endocardium" layer. The complete description of the modeling of I_{ion} is outside the scope of this thesis, and further details are omitted.

As a simplifying step, we can decouple the reaction part $\chi I_{\text{ion}}(t, u, \vec{\phi})$ and the diffusion

[8] https://www.sintef.no/contentassets/3eb4691190f2451fb21349eb24cb9e8e/part-2-cpugpu.pdf

part $\nabla \cdot (K(\vec{x})\nabla u)$ of Equation 9 by using an operator splitting method. The reaction part is reformulated as a system of non-linear ordinary differential equations (ODEs)

$$\frac{\mathrm{d}u}{\mathrm{d}t} = -\chi I_{\text{ion}}(t, u, \vec{\phi}) \tag{10}$$

$$\frac{\mathrm{d}\vec{\phi}}{\mathrm{d}t} = \vec{f}(t, u, \vec{\phi}). \tag{11}$$

The diffusion part becomes a linear partial differential equation (PDE)

$$\frac{\partial u}{\partial t} = \nabla \cdot (K(\vec{x})\nabla u). \tag{12}$$

Whereas the ODEs in Equations 10 and 11 can be solved individually (i.e. in a manner isolated from the other cells), Equation 12 must be solved in a way that involves all other tetrahedral cells (i.e. there exists numerical coupling between the cells due only to Equation 12). Equations 10 and 11 – together called the "cell model" – are considered to be handled by a separate part of LYNX, and no longer considered. Instead, we focus on the PDE in Equation 12 for the remainder of this section.

Equation 12 cannot be used as it is, because both the time and space variables are over continuous sets of real numbers. It would not be possible to solve on a computer, which is inherently finite and discrete. We must first discretize Equation 12 over both space and time to a set of linear equations, that can then be solved to obtain the value of u over each finite volume for each time step t.

Remark. The FVM is favored for this application – for example, over the Finite Difference Method – because due to the geometric complexity, we cannot be restricted to structured meshes.

We begin with the spatial variable. The cell-centered variant of the FVM (CC-FVM) solves PDEs by placing degrees of freedom at the center of each of these computational cells [19], and is what is used by LYNX. Essentially, the cell center "stores" the average variable value of the membrane potential u in the cell.

We will assume the natural "no-flux" boundary condition for CC-FVM:

$$\frac{\partial u}{\partial n} = 0.$$

The 3D geometry of the object of interest – be it a section, or the entirety of – the heart, is tessellated into tetrahedrons so that each cell has four faces. Such a model can be obtained by medical imaging techniques like "Magnetic Resonance Imaging" (MRI). As will be seen, the volume integrals that contain divergence terms are replaced with surface integrals via Gauss's Theorem. This means that we will directly be needing to know the value of the variable of interest u_k for the four first-order neighbors of the finite volume of interest V for a particular time t_k and location $\vec{x}_k$. That is, the cells directly adjacent to V. Furthermore, LYNX uses a second-order spatial discretization

scheme, and due to this coupling on the tetrahedral faces, we also need u_k for all cells adjacent to the first-order neighbors, which we call the second-order neighbors.

Assuming the mesh is an approximation of a complex geometry, as would be the case in personalized cardiac simulations, there is no structured way to order these tetrahedra. Hence, LYNX uses a CC-FVM discretization over a (necessarily) unstructured mesh.

Our goal with the time discretization is to have a method that finds the discrete u_i^ℓ as the approximation of the continuous $u(\vec{x}_i, \ell\Delta t)$, where $\vec{x}_i$ is the geometrical center of tetrahedron i, ℓ the time step, and Δt the length of the time step [18]. If Equation 12 is true, then it must also be true that

$$\frac{\partial u^\ell(\vec{x},t)}{\partial t} = \boldsymbol{\nabla}\cdot\left(K(\vec{x})\boldsymbol{\nabla}u^\ell\right) \tag{13}$$

for all time steps ℓ.

By the Forward Euler Method, we know that

$$\frac{\partial u^\ell}{\partial t} = \frac{u^{\ell+1}-u^\ell}{\Delta t}.$$

Substituting this into Equation 13, we take the integral of both sides over an arbitrary tetrahedron V:

$$\int_V \frac{\partial u^\ell(\vec{x},t)}{\partial t}\mathrm{d}\vec{x} = \int_V \boldsymbol{\nabla}\cdot\left(K(\vec{x})\boldsymbol{\nabla}u^\ell\right)\mathrm{d} \tag{14}$$

As mentioned earlier, we apply Gauss's Theorem to the RHS to form a surface integral over the tetrahedron:

$$\int_V \boldsymbol{\nabla}\cdot\left(K(\vec{x})\boldsymbol{\nabla}u^\ell\right)\mathrm{d} = \int_{\partial V} n\cdot K(\vec{x})\boldsymbol{\nabla}u^\ell \mathrm{d}o_x \tag{15}$$

where n is the outward unit normal vector.

And we now have

$$\int_V \frac{\partial u^\ell(\vec{x},t)}{\partial t}\mathrm{d}\vec{x} = \int_{\partial V} n\cdot K(\vec{x})\boldsymbol{\nabla}u^\ell \mathrm{d}o_x \tag{16}$$

For the approximation of the LHS, let V_V be the volume of the arbitrary cell V. [18]:

$$\int_V \frac{\partial u^\ell(\vec{x},t)}{\partial t}\mathrm{d}\vec{x} \approx \frac{(u^{\ell+1}-u^\ell)V_V}{\Delta t} \tag{17}$$

For the RHS approximation, we can approximate the surface integral iteratively on each of the four faces of the tetrahedron. For each time step, the value at the next time step depends on the four first-order neighbors directly adjacent to V, as well as the 12 second-order neighbors. As previously mentioned, the second-order neighbors are the tetrahedra adjacent to the first-order neighbors of V, not including V. Collect each neighbor i into the set N_i. Then, we have that

$$\int_{\partial V} n \cdot K(\vec{x}) \nabla u^{\ell} \mathrm{d}o_x \approx \sum_{j \in N_i} a_{i,j}(u_j^{\ell} - u^{\ell}) \tag{18}$$

where $\alpha_{i,j}$ is a weight, depending on the first and second-order neighbors[9].

Remark. Notice that $|N_i| \leq 16$, since we allow second-order neighbors to coincide [15].

Then, for every finite volume i, we can collect both sides into the equation:

$$\frac{(u_i^{\ell+1} - u_i^{\ell}) V_{V,i}}{\Delta t} = \sum_{j \in N_i} \alpha_{i,j}(u_j^{\ell} - u_i^{\ell}) \tag{19}$$

And after rearranging terms,

$$u_i^{\ell+1} = u_i^{\ell} + \frac{\Delta t}{V_{V,i}} \sum_{j \in N_i} \alpha_{i,j}(u_j^{\ell} - u_i^{\ell}) \tag{20}$$

To avoid the unstable case of $\alpha < 0$, we choose Δt such that

$$\Delta t < \frac{1}{\sum_{j \in N_i} \left| a_{i,j} \right|} \text{ for all } i.$$

Let Z be the matrix whose elements $\alpha_{i,j}$ are defined

$$\alpha_{i,j} = \begin{cases} \frac{\Delta t}{V_{V,i}} \alpha_{i,j} & \text{if } j \in N_i, \\ 1 - \frac{\Delta t}{V_{V,i}} \sum_{k \in N_i} \alpha_{i.k} & \text{if } j = i, \\ 0 \quad \text{elsewise} \end{cases}$$

Remark. The matrix Z has one row per tetrahedron. Each row has a non-zero element on the diagonal, and up to 16 off-diagonal non-zero elements representing connections to its neighbors. In Section 6 we see evidence for this fact, as the two matrices `Lynx68` and `Lynx649` contain less than 17 non-zeros per row on average. The column positions of these non-zero entries are irregular due to the unstructured nature of the tetrahedral mesh.

Finally, for a given tetrahedron i, we can write Equation 20 as

$$u_i^{\ell+1} = \alpha_{i,i} u_i^{\ell} + \sum_{j \in N_i} \alpha_{i,j} u_j^{\ell}$$

which can be expressed in terms of an SpMV

$$u^{\ell+1} = Z u^{\ell},$$

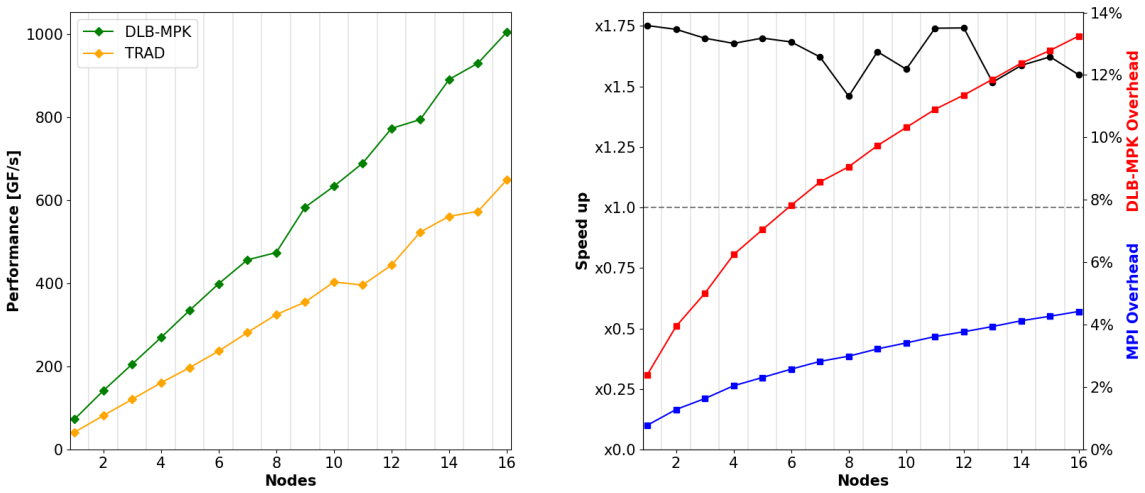
[9] The details of exactly how $\alpha_{i,j}$ is calculated have not been published, but have been mentioned in both [19] and [15]

where the vectors $u^{\ell+1}$ and u^{ℓ} contain the numerical solutions at $t = (\ell + 1)\Delta t$ and $t = \ell \Delta t$ respectively.

Hence, to obtain the membrane potentials u for the next time step, one would need to compute one SpMV. Instead of computing these time steps one after another, we apply DLB-MPK in order to compute groups of p-many time steps, where p is the most performant power for the calculation $A^p x$ as described in Section 4.2.

Remark. Recall from Section 1 that it is exactly these types of applications that generate large, sparse matrices to describe finite volumes. Hence, this problem of progressing one step forward in time is main memory-bound. So this is a prime example of where DLB-MPK could be used.

The matrix Lynx649 was generated by LYNX and represents a cardiac model tessellated into roughly 64.9 million tetrahedra. See Figure 34 for a reference on what such a tessellation could look like for a given time step, where the color on each cell indicates the intensity of the electrical signal u. We refer to the current implementation internal to LYNX as TRAD.



(a) Strong Scaling Performance

(b) Strong Scaling Overheads

Figure 35: Lynx649 Strong Scaling, 16 Nodes

In Figure 35, we tune DLB-MPK to use $p = 4$. Due to the size and structure of Lynx649, we set the maximum recursion depth for RACE at 80. TRAD with local BFS permutations were taken as the baseline. MPI processes are pinned to ccNUMA domains in groups of 4, so only the node count is shown on the x-axis.

6 Summary

Over the course of this thesis, we: motivated and developed a novel cache-blocked MPI-parallel MPK, analyzed its performance on a modern Intel architecture, and applied our algorithm to a real-world problem in Cardiac Electrophysiology.

In Section 2, the first main section of this work is concerned with the background material needed to understand and give context to the contributions. We began with a discussion on our parallel execution model in Section 2.1, as well as some basic terms and ideas in HPC that we will be using. Then in Sections 2.2 and 2.3, we discussed the two kernels that are the main focus of the thesis, SpMV and MPK. The "Roofline Model" was introduced in Section 2.4, since both kernels are main memory-bound. Permutations, specifically "symmetric" permutations are offered as a technique to improve the scattered data accesses of SpMV and MPK in Section 2.5. In Section 2.6, the 1 to 1 correspondence that matrices have with graphs is introduced as a convenient way to visualize and understand dependencies between MPI processes, once we move to the distributed memory setting. The important concept of halo communication is introduced in this section as well.

The main contributions of the thesis are described in Section 3. First, The Recursive Algebraic Coloring Engine (RACE) – the library around which this entire work is based – is described in Section 3.1. We pay special attention to the formation of "levels", and the Breadth First Search (BFS) done by RACE to collect these levels. We then apply RACE to the MPK, and examine the data dependencies between successive row-segmented SpMV iterations. From Section 3.3 onwards, we focus on the development of the Distributed Level Blocked Matrix Power Kernel (DLB-MPK). The algorithm is segmented into a pre-processing phase described in Section 3.3.1, and three computational phases: MPI Pre-Computation (Section 3.3.2), Local LB-MPK (Section 3.3.3), and MPI Post-Computations (Section 3.3.4). The main idea being to fulfill the data dependencies in a clever way such that LB-MPK execute locally on each MPI process.

The results of the DLB-MPK compared to DLB-MPK – the traditional back-to-back SpMV implementation of MPK – are given in Section 4. The testbed machine we use is described in Section 4.1, and the full set of benchmarks and parameter studies are given in Section 4.2. Given that bandwidth from L3 cache is about x2.5 greater than bandwidth from main memory, we can expect a maximum speed-up of x2.5 with our cache blocking DLB-MPK. In practice, this speed-up depends strongly on the sparsity pattern of the matrix and manner of partitioning. We examine performance through the lens of both strong and weak scaling. Finally, in Section 5, we apply our new algorithm to a real-world problem: that of simulating the electrical impulses in computerized cardiac models to detect arrhythmia risk. The Cardiac Arrhythmia Simulation (CAS) code used to do this is called LYNX. On one such realistic simulation, our algorithm performs x1.5 to x1.75 faster than what is currently implemented in LYNX.

There are several interesting avenues for future work.

1. Support for matrix storage formats other than CRS. As mentioned in Section 2.2, the ELLPACK storage format is more conducive to GPU implementations and is what LYNX actually uses in practice. Apart from the opportunity to experiment with different storage formats for varying applications, this flexibility would give us another parameter to tune for, giving further opportunity for performance optimizations.

2. There is nothing inherent about DLB-MPK or RACE that restricts us to use only the CPU, as was done in this thesis. Especially with the trend of server-side GPUs being made with larger cache sizes, a DLB-MPK implementation for GPUs would be a very exciting next step.

3. One long-term goal of this work is integration into LYNX so that users can make automatic use of our distributed cache-blocking MPK. The `Lynx649` matrix is taken from one instance of LYNX, merely for our benchmarking purposes. But in order for DLB-MPK to be fully integrated into LYNX, significant time needs to be invested.

4. The "templating" of RACE and DLB-MPK would allow us to support a wider range of datatypes (i.e. `long int` and `float`). This would provide support for larger applications to make use of DLB-MPK without the danger of integer overflow, and enable us to experiment with mixed precision methods.

5. As mentioned in Section 3.3.4, one could overlap the communication and computation parts of the MPI Post-Computation phase in order to (at least, partially) hide the cost of communication. This may be the most direct way of improving DLB-MPK across all matrix storage formats.

Benchmark Matrices

Matrix	N_r	N_{nz}	N_{nzr}	CRS Size [MB]
thermal2	1_228_045	8_580_313	6.9	102.8
crankseg_1	52_804	10_614_210	201.0	121.6
pwtk	217_918	11_634_424	53.3	133.9
rajat31	4_690_002	20_316_253	4.3	250.3
gsm_106857	589_446	21_758_924	36.9	251.2
F1	343_791	26_837_113	78.0	308.4
cage14	1_505_785	27_130_349	18.0	316.2
Fault_639	638_802	28_614_564	44.7	329.9
inline_1	503_712	36_816_342	73.0	423.2
Emilia_923	923_136	41_005_206	44.4	472.7
ldoor	952_203	46_522_475	48.8	536.0
af_shell10	1_508_065	52_672_325	34.9	608.5
Hook_1498	1_498_023	60_917_445	40.6	702.8
Geo_1438	1_437_960	63_156_690	43.9	728.2
Serena	1_391_349	64_531_701	46.3	743.8
bone010	986_703	71_666_325	72.6	823.9
audikw_1	943_695	77_651_847	82.2	892.2
channel-500x100x100-b050	4_802_000	85_362_744	17.7	995.2
dielFilterV3real	1_102_824	89_306_020	80.9	1_026.2
nlpkkt120	3_542_400	96_845_792	27.3	1_121.8
ML_Geer	1_504_002	110_879_972	73.7	1_274.6
Lynx68	6_811_350	111_560_826	16.3	1_302.6
Flan_1565	1_564_794	117_406_044	75.0	1_349.5
nlpkkt200	16_240_000	448_225_632	27.6	5_191.4
Lynx649	64_950_632	978_866_282	15.0	11_450.0
Anderson(600,600,600)	216_000_000	1_293_840_000	5.9	15_630.7
Anderson(175,175,175)	5_359_375	31_972_500	5.9	386.3
Anderson(350,175,175)	10_718_750	64_006_250	5.9	773.3
Anderson(350,350,175)	21_437_500	128_135_000	5.9	1_548.1
Anderson(350,350,350)	42_875_000	256_515_000	5.9	3_099.1
Anderson(700,350,350)	85_750_000	513_275_000	5.9	6_201.0
Anderson(700,700,350)	171_500_000	1_027_040_000	5.9	12_407.7
Anderson(700,700,700)	343_000_000	2_055_060_000	5.9	24_826.7

List of Algorithms

Acronyms and Terms

BFS Breadth First Search.

CAS Cardiac Arrhythmia Simulation.

ccNUMA cache-coherent Non Uniform Memory Access.

CPU Central Processing Unit.

CRS Compressed Row Storage Sparse Matrix Storage Format.

DLB-MPK Distributed Level-Blocked Matrix Power Kernel.

LB-MPK Level-Blocked Matrix Power Kernel.

MPI Message Passing Interface.

MPK Matrix-Power kernel.

RACE Recursive Algebraic Coloring Engine.

SIMD Single Instruction Multiple Data.

SpMV Sparse Matrix Vector Multiplication kernel.

TRAD Traditional Back-to-Back SpMV implementation of MPK.

UMA Uniform Memory Access.

References

[1] ALAPPAT, Christie ; BASERMANN, Achim ; BISHOP, Alan R. ; FEHSKE, Holger ; HAGER, Georg ; SCHENK, Olaf ; THIES, Jonas ; WELLEIN, Gerhard: A Recursive Algebraic Coloring Technique for Hardware-Efficient Symmetric Sparse Matrix-Vector Multiplication. 7 (2020), jun, Nr. 3. http://dx.doi.org/10.1145/3399732. – DOI 10.1145/3399732. – ISSN 2329–4949

[2] ALAPPAT, Christie ; HAGER, Georg ; SCHENK, Olaf ; WELLEIN, Gerhard: Level-based Blocking for Sparse Matrices: Sparse Matrix-Power-Vector Multiplication. In: *IEEE Transactions on Parallel and Distributed Systems* (2022), S. 1–18. http://dx.doi.org/10.1109/TPDS.2022.3223512. – DOI 10.1109/TPDS.2022.3223512

[3] ALAPPAT, Christie L. ; HOFMANN, Johannes ; HAGER, Georg ; FEHSKE, Holger ; BISHOP, Alan R. ; WELLEIN, Gerhard: Understanding HPC Benchmark Performance on Intel Broadwell and Cascade Lake Processors. In: SADAYAPPAN, Ponnuswamy (Hrsg.) ; CHAMBERLAIN, Bradford L. (Hrsg.) ; JUCKELAND, Guido (Hrsg.) ; LTAIEF, Hatem (Hrsg.): *High Performance Computing*. Cham : Springer International Publishing, 2020. – ISBN 978–3–030–50743–5, S. 412–433

[4] AREVALO, Hermenegild J. ; VADAKKUMPADAN, Fijoy ; GUALLAR, Eliseo ; JEBB, Alexander ; MALAMAS, Peter ; WU, Katherine C. ; TRAYANOVA, Natalia A.: Arrhythmia risk stratification of patients after myocardial infarction using personalized heart models. In: *Nature Communications* 7 (2016), Nr. 1. http://dx.doi.org/10.1038/ncomms11437. – DOI 10.1038/ncomms11437

[5] BARRETT, Richard ; BERRY, Michael ; CHAN, Tony F. ; DEMMEL, James ; DONATO, June ; DONGARRA, Jack ; EIJKHOUT, Victor ; POZO, Roldan ; ROMINE, Charles ; VORST, Henk van d.: *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics, 1994. http://dx.doi.org/10.1137/1.9781611971538. http://dx.doi.org/10.1137/1.9781611971538

[6] BENDER, E ; WILLIAMSON, S ; LISTS, Decisions: Graphs With an Introduction to Probability. In: *University of California at San Diego* (2010)

[7] CHECCONI, Fabio ; TITHI, Jesmin J. ; PETRINI, Fabrizio: *Ridgeline: A 2D Roofline Model for Distributed Systems*. http://dx.doi.org/10.48550/ARXIV.2209.01368. Version: 2022

[8] CLAYTON, R.H. ; BERNUS, O. ; CHERRY, E.M. ; DIERCKX, H. ; FENTON, F.H. ; MIRABELLA, L. ; PANFILOV, A.V. ; SACHSE, F.B. ; SEEMANN, G. ; ZHANG, H. ; AL. et: Models of cardiac tissue electrophysiology: Progress, challenges and open questions. In: *Progress in Biophysics and Molecular Biology* 104 (2011), Nr. 1-3, S. 22–48. http://dx.doi.org/10.1016/j.pbiomolbio.2010.05.008. – DOI 10.1016/j.pbiomolbio.2010.05.008

[9] DAVIS, Timothy A. ; HU, Yifan: The University of Florida Sparse Matrix Collection. In: *ACM Trans. Math. Softw.* 38 (2011), dec, Nr. 1. http://dx.doi.org/10.1145/2049662.2049663. – DOI 10.1145/2049662.2049663. – ISSN 0098–3500

[10] FLYNN, M. J.: Some Computer Organizations and Their Effectiveness. In: *IEEE Transactions on Computers* C-21 (Sept. 1972), S. 948–960. http://dx.doi.org/10.1109/TC.1972.5009071. – DOI 10.1109/TC.1972.5009071

[11] GOUMAS, Georgios ; KOURTIS, Kornilios ; ANASTOPOULOS, Nikos ; KARAKASIS, Vasileios ; KOZIRIS, Nectarios: Performance evaluation of the sparse matrix-vector multiplication on modern architectures. In: *Journal of Scientific Computing* 50 (2009), 36-77. http://dx.doi.org/10.1007/s11227-008-0251-8. – DOI 10.1007/s11227–008–0251–8. – ISSN 1573–0484

[12] HAGER, Georg ; WELLEIN, Gerhard: *Introduction to High Performance Computing for Scientists and Engineers*. 1. Taylor and Francis Group, LLC, 2011

[13] HOEMMEN, Mark F.: *Communication-avoiding Krylov subspace methods*, EECS Department, University of California, Berkeley, Diss., Apr 2010. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-37.html

[14] HUIKURI, Heikki V. ; CASTELLANOS, Agustin ; MYERBURG, Robert J.: Sudden Death Due to Cardiac Arrhythmias. In: *New England Journal of Medicine* 345 (2001), Nr. 20, 1473-1482. http://dx.doi.org/10.1056/NEJMra000650. – DOI 10.1056/NEJMra000650. – PMID: 11794197

[15] HUSTAD, Kristian G.: *Solving the monodomain model efficiently on GPUs*, Department of Informatics Faculty of mathematics and natural sciences, University of Oslo, Diss., Autumn 2019. https://www.duo.uio.no/

[16] KREUTZER, Moritz: *Performance engineering for exascale-enabled sparse linear algebra building blocks = performance-engineering fur Extrascalefahige Grundbausteine linearer algebra MIT dunn Besetzten Matrizen*. FAU University Press, 2018

[17] KREUTZER, Moritz ; HAGER, Georg ; WELLEIN, Gerhard ; FEHSKE, Holger ; BISHOP, Alan R.: A Unified Sparse Matrix Data Format for Efficient General Sparse Matrix-Vector Multiplication on Modern Processors with Wide SIMD Units. In: *SIAM Journal on Scientific Computing* 36 (2014), Nr. 5, S. C401–C423. http://dx.doi.org/10.1137/130930352. – DOI 10.1137/130930352

[18] LANGGUTH, Johannes ; AREVALO, Hermenegild ; HUSTAD, Kristian G. ; CAI, Xing: Towards Detailed Real-Time Simulations of Cardiac Arrhythmia. In: *Computing in Cardiology* 46 (2019), S. 1–4. http://dx.doi.org/10.22489/CinC.2019.301. – DOI 10.22489/CinC.2019.301

[19] LANGGUTH, Johannes ; CAI, Xing: Heterogeneous CPU-GPU computing for the finite volume method on 3D unstructured meshes. In: *2014 20th IEEE International Conference on Parallel and Distributed Systems (ICPADS)*, 2014, S. 191–199

[20] In: LOE, Jennifer ; THORNQUIST, Heidi ; BOMAN, Erik: *Polynomial Preconditioned GMRES in Trilinos: Practical Considerations for High-Performance Computing*. 2020. – ISBN 978–1–61197–613–7, S. 35–45

[21] MESSAGE PASSING INTERFACE FORUM: *MPI: A Message-Passing Interface Standard Version 4.0*, Juni 2021. https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[22] MOHIYUDDIN, Marghoob ; HOEMMEN, Mark ; DEMMEL, James ; YELICK, Katherine: Minimizing Communication in Sparse Matrix Solvers. In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA : Association for Computing Machinery, 2009 (SC '09). – ISBN 9781605587448

[23] MONAKOV, Alexander ; LOKHMOTOV, Anton ; AVETISYAN, Arutyun: Automatically Tuning Sparse Matrix-Vector Multiplication for GPU Architectures. In: PATT, Yale N. (Hrsg.) ; FOGLIA, Pierfrancesco (Hrsg.) ; DUESTERWALD, Evelyn (Hrsg.) ; FARABOSCHI, Paolo (Hrsg.) ; MARTORELL, Xavier (Hrsg.): *High Performance Embedded Architectures and Compilers*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2010. – ISBN 978–3–642–11515–8, S. 111–125

[24] OPENMP ARCHITECTURE REVIEW BOARD: *OpenMP Application Program Interface Version 3.0*. http://www.openmp.org/mp-documents/spec30.pdf. Version: Mai 2008

[25] SAAD, Youcef: SPARSKIT: A basic tool kit for sparse matrix computations. 1990. – Forschungsbericht

[26] SAAD, Yousef: *Iterative Methods for Sparse Linear Systems*. SIAM, 2003

[27] SIMONCINI, Valeria ; HIGHAM, Nicholas J. ; AL. et: *The Princeton Companion to Applied Mathematics: Krylov Subspaces*. Princeton University Press, 2015. – 113–114 S.

[28] STRØM, Vilde N.: *Using personalized virtual hearts to assess arrhythmia risk in acute infarction patients*. https://www.duo.uio.no/handle/10852/63482. Version: Aug 2018

[29] SUHOV, A. Y.: An Accurate Polynomial Approximation of Exponential Integrators. In: *Journal of Scientific Computing* 60 (2014), Nr. 3, 684-698. http://dx.doi.org/10.1007/s10915-013-9813-x. – DOI 10.1007/s10915–013–9813–x. – ISSN 1573–7691

[30] TREIBIG, J. ; HAGER, G. ; WELLEIN, G.: LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In: *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*. San Diego CA, 2010

[31] TUSSCHER, K H. ; PANFILOV, A V.: Cell model for efficient simulation of wave propagation in human ventricular tissue under normal and pathological conditions. In: *Physics in Medicine and Biology* 51 (2006), Nr. 23, S. 6141–6156. http://dx.doi.org/10.1088/0031-9155/51/23/014. – DOI 10.1088/0031–9155/51/23/014

[32] VATAI, Emil ; SINGHAL, Utsav ; SUDA, Reiji: Diamond Matrix Powers Kernels. In: *Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region*. New York, NY, USA : Association for Computing Machinery, 2020 (HPCAsia2020). – ISBN 9781450372367, 102–113

[33] WILLIAMS, Samuel ; OLIKER, Leonid ; VUDUC, Richard ; SHALF, John ; YELICK, Katherine ; DEMMEL, James: Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms. In: *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. New York, NY, USA : Association for Computing Machinery, 2007 (SC '07). – ISBN 9781595937643

[34] WILLIAMS, Samuel ; WATERMAN, Andrew ; PATTERSON, David: Roofline: An Insightful Visual Performance Model for Multicore Architectures. In: *Commun. ACM* 52 (2009), apr, Nr. 4, 65–76. http://dx.doi.org/10.1145/1498765.1498785. – DOI 10.1145/1498765.1498785. – ISSN 0001–0782

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, dass alle Stellen der Arbeit, die wörtlich oder sinngemäß aus anderen Quellen übernommen wurden, als solche kenntlich gemacht sind und dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegt wurde.

Erlangen, den March 9, 2023 Hierher die Unterschrift

Lebenslauf

Dane Lacey

Zur Person

Name	Dane Lacey
Geboren	06.11.1996
Adresse	Georg-Krauss-Strasse 8 91056 Erlangen

Schule

08/2012 - 06/2015	Bingham High School (3.25 US)

Studium

08/2018 - 12/2019	University of Utah, B.Sc. Applied Mathematics (3.95 US)
01/2016 - 05/2018	Salt Lake Community College, A.Sc. Mathematics (3.8 US)

Sonstige Tätigkeiten

04/2022 - Present	Research Assistant at Erlangen National High Performance Computing Center
07/2021 - 03/2022	Research Assistant at Fraunhofer Institute for Wind Energy Systems
12/2020 - 06/2021	Research Assistant at University Erlangen-Nuremberg Erlangen, Chair for Analytics & Mixed-Integer Optimization

---