FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Distrubuted Sparse Matrix-Vector Multiplication
Student:	Bc. Boris Rúra
Supervisor:	doc. Ing. Ivan Šimeček, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of winter semester 2022/2023

Instructions

1) Review existing libraries for multithreaded sparse matrix-vector multiplication (SpMV) at least [1,2].
2) Review existing approaches to distributed SpMV [1].
3) Discuss distributed SpMV with selected libraries from point 1).
4) Implement ideas from point 3) using OpenMP and MPI libraries.
5) Measure the resulting performance and speedup. Compare the performance of the implementation with similarly focused libraries.

[1] Kozický. Parallel Joint Direct and Transposed Sparse Matrix-Vector Multiplication, diploma thesis, CTU FIT, 2019
[2] W. Liu and B. Vinter. 2015. CSR5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In Proceedings of the 29th ACM on International Conference on Supercomputing. ACM,339–350.
[3] J. Eckstein and G. Matyasfalvi. Efficient distributed-memory parallel matrix-vector multiplication with wide or tall unstructured sparse matrices.CoRR, abs/1812.00904, 2018.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 13 August 2021 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Distributed Sparse Matrix-Vector Multiplication

Bc. Boris Rúra

Department of Theoretical Computer Science
Supervisor: doc. Ing. Ivan Šimeček, Ph.D.

January 5, 2022

Acknowledgements

I'm extremely grateful to my parents for always supporting me and to my sister for showing me the ropes when I moved to Prague. I would also like to extend my deepest gratitude to my supervisor doc. Ing. Ivan Šimeček, Ph.D., for providing invaluable guidance during the process of writing this thesis. The access to the computational infrastructure of the OP VVV funded project CZ.02.1.01/0.0/0.0/16_019/0000765 "Research Center for Informatics" is also gratefully acknowledged.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on January 5, 2022 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Rúra, Boris. *Distributed Sparse Matrix-Vector Multiplication.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2022.

Abstrakt

Cílem práce bylo prozkoumat možnosti distribuovaného násobení řídké matice vektorem několika procesy s použitím MPI a CSR5. Výsledkem tohto výzkumu je C++ knihovna `dim`, která poskytuje potřebné stavební bloky pro distribuované násobení řídkých matic vektorem za pomoci formátu CSR5. Potencionálni zrychlení distribuovaného násobení řídké matice vektorem pak bylo meřeno na implementaci metody konjugovaných gradientů a porovnávano s jednoprocesovou implementací založenou na CSR5 i distribuovanou implementací pomocí PETSc.

Klíčová slova násobení řídké matice vektorem, C++, OpenMP, MPI, metoda konjugovaných gradientů, CSR5, HDF5

Abstract

The aim of this thesis is to research possibilities of distributing sparse matrix-vector multiplication among multiple processes using MPI and CSR5 storage format. The result of this research is a C++ library `dim`, which provides the building blocks for distributed SpMV using CSR5. The potential speedup of distributed SpMV is then benchmarked on a conjugate gradient algorithm implementation against a single-process CSR5 based implementation as well as PETSc based multi-process implementation.

Keywords sparse matrix-vector multiplication, C++, OpenMP, MPI, conjugate gradient method, CSR5, HDF5

Contents

List of Figures

List of Tables

Contents

Introduction

Sparse matrix-vector multiplication is a fundamental computational kernel used for many scientific computations, such as graph algorithms, numerical analysis, conjugate gradients as well as some machine learning algorithms, such as support vector machines. While matrix-vector multiplication is a simple multiplication task, it is non-trivial to load balance properly for every sparsity structure matrix when parallelized.

The conjugate gradient algorithm is one of the best-known iterative methods, which can be used to solve large symmetric positive definite linear systems. With sparse matrix-vector multiplication being the most computationally intensive step of a conjugate gradient iteration, parallelizing the kernel and distributing the computation across multiple nodes can result in significantly shorter iteration times.

The goals of this thesis are to:

1. Review existing sparse matrix storage formats.
2. Review existing approaches to parallel SpMV.
3. Implement the parallel SpMV algorithm introduced in [2].
4. Introduce efficient on-disk storage format for sparse-matrix format outlined in [2] suitable for distributed version.
5. Implement a distributed version of parallel SpMV (D-CSR5) using the parallel implementation and storage format introduced in this thesis.
6. Measure and review the viability of distributing SpMV.
7. Benchmark the distributed implementation against PETSc suite.

Chapter 1

Overview of used technologies

This chapter introduces the technologies used in the implementation of this thesis. SIMD parallel processing, as well as SSE/AVX instructions sets, are defined in Section 1.1. Section 1.2 introduces OpenMP API, used to parallelize the computation. Section 1.3 introduces MPI, a Message Passing Interface, used for inter-process communication in the distributed SpMV implementation. Lastly Section 1.5 introduces Matrix Marked Exchange Format for storing sparse and dense matrices, and Section 1.6 introduces Hierarchical Data Format v5, used for efficient sparse matrix on-disk storage as described in Section 3.5.

1.1 SIMD

As defined in Flynn's taxonomy, Single Instruction Multiple Data (or SIMD) is a type of parallel processing, where a single instruction is applied to multiple data streams.

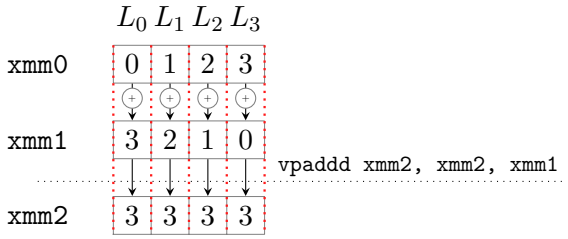

Figure 1.1: Example of SIMD addition using SSE

Modern CPUs provide Streaming SIMD Extensions (or SSE) Instruction Set Extensions which work on 16 byte registers, as well as Advanced Vector Extensions (or AVX), which work with 32-byte registers. To illustrate, Figure 1.1 shows addition (single instruction) of two four-integer arrays (multiple data)

using the `vpaddd` instruction from SSE. The data streams are sometimes referred to as lanes (marked as L_0-L_3 in Figure 1.1).

In C++, there are several ways to achieve vectorization[1]. First of these are SIMD intrinsics. Intrinsics are assembly-coded functions that let you use C++ function calls and variables in place of assembly instructions, improving readability. To generate the `vpaddd` instruction from Figure 1.1 intrinsic `_mm_add_epi32` can be used.

Vectorization may also be achieved by using a pragma directive from OpenMP described in Section 1.2, as shown in Figure 1.2.

```
#pragma omp simd
for (int lane = 0; lane < 4; ++lane)
    c[i] = a[i] + b[i]
```

Figure 1.2: Vectorization using OpenMP

Lastly, standard C++ may be used to produce vectorized code by utilizing `<execution>` header and its `par_unseq` and `unseq` execution policies. However, these only aid the compiler auto-vectorization and do not guarantee the vector instructions will actually be used.

1.2 OpenMP

OpenMP API is defined by a collection of compiler directives, library routines, and environment variables. It provides a model for parallel programming that is portable across architectures from different vendors. Compilers from numerous vendors support the OpenMP API[3]. OpenMP uses the fork-join model of parallel execution.

For C++ the directives have the form of preprocessor `#pragma` directives, with the following syntax.

```
#pragma omp <directive-name> [clause[[,]clause]...] new-line
```

To enable the use of OpenMP, the compiler must be invoked with the `-fopenmp` switch for GCC/clang-based compilers, while MSVC accepts the `/openmp` flag.

1.2.1 `parallel` directive

`parallel` directive marks a region to be executed in parallel. A **team** within the parallel region is a set of one or more threads participating in the execution

[1]Note that the target architecture needs to be specified for the compiler to enable vector instructions.

of this region. If the team has at least two active threads, the parallel region is **active**, else it is called **inactive**.

The threads in OpenMP have a **numerical identifier**. In this thesis, this numerical identifier will be referred to as **thread id**[2]. The thread which has thread id equal to 0 is called the **master** thread within the team while other consecutive thread ids are assigned to remaining threads. It can be obtained by calling `omp_get_thread_num`.

1.2.2 `for` directive

Referred to as Worksharing-Loop Construct in [3]. It specifies that the iterations of the associated loop(s) will be executed in parallel by the threads of the currently active team. As a result, the execution is only parallelized if the loop is performed inside of a `parallel` region in the first place.

All of the allowed clauses for the directive can be found in [3]. The only relevant clause for this thesis is `schedule`, specified as
`schedule([modifier [, modifier]:]kind[, chunk_size])`.

Modifier isn't immediately relevant but `kind` is. **kind** specifies, how the work is distributed among the threads. When `dynamic` kind is specified, the iterations are distributed among threads dynamically. Meaning when a thread finishes processing its assigned chunk (of `chunk_size` elements), it requests a new chunk until no chunks remain.

`static` kind distributes chunks of `chunk_size` among the participating threads in a round-robin fashion. This kind is used heavily in the computational kernel described in Section 3.3.

`parallel` and `for` directives may be combined as shown in Figure 1.3, to create a so-called parallel worksharing-loop construct. Which is a shortcut for specifying `parallel` construct containing a worksharing-loop.

```
#pragma omp parallel for
for (int i = 0; i < num_iters; ++i)
    // performed in parallel
```

Figure 1.3: Loop parallelized using OpenMP directives.

OpenMP contains many other primitives for parallel programming such as atomics, barriers and semaphores and can also be used to vectorize loops using the `simd` construct as shown in Figure 1.2.

[2]Not the thread id in the context of an OS.

1.3 MPI

MPI (Message-Passing Interface) is a message-passing library interface specification. All parts of this definition are significant. MPI addresses primarily the message-passing parallel programming model, in which data is moved from the address space of one process to that of another process through cooperative operations on each process.

Extensions to the "classical" message-passing model are provided in collective operations, remote-memory access operations, dynamic process creation, and parallel I/O, which is used to speed-up loading sparse matrices from disk as described in Section 4.1. MPI is a specification, not an implementation; there are multiple implementations of MPI. This specification is for a library interface; MPI is not a language, and all MPI operations are expressed as functions, subroutines, or methods, according to the appropriate language bindings which, for C and Fortran, are part of the MPI standard. [4]

1.3.1 Initialization

Before invoking any MPI routines, a participating process must call `MPI_Init` which initializes internal state of an MPI implementation, which must then be freed by calling `MPI_Finalize`.

1.3.2 Communicators

Most MPI routines require a communicator. In MPI a **communicator** a communicator is a context for a communication operation [4]. Participating processes are also referred to as **process group** and are each process is assigned a non-negative **rank** within this group.

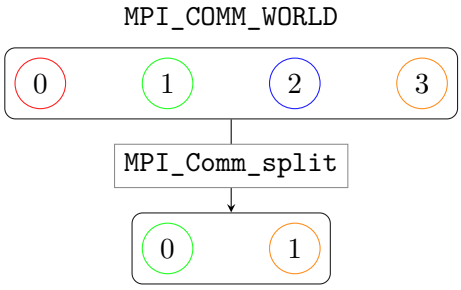

Figure 1.4: Splitting a communicator to create a new one

MPI implementations provide some predefined communicators such as `MPI_COMM_WORLD` which contains every process accessible after initialization described in Section 1.3.1 and `MPI_COMM_SELF` which contains only the calling process (self). One of the most common methods for creating new communicators is splitting an existing one. This can be done by calling `MPI_Comm_split`

(Shown in Figure 1.4) and is utilized for synchronization of sub-results as explained in Section 4.2.

1.3.3 MPI operations

An MPI operation is a sequence of steps performed to establish and enable data transfer/synchronization[4]. It consists of four stages:

- **Initialization** hands over the argument list to the operation, but not the content of the data buffers.
- **Starting** hands over the control of the data buffers.
- **Completion** returns control of the content of the data buffers and indicates that output buffers and arguments may have been updated.
- **Freeing** returns the control of the rest of the argument list to the caller.

1.3.4 `MPI_Allgatherv`

`MPI_Allgatherv` is a collective, blocking operation; useful for distributing arrays of data between processes. A **collective** operation in MPI is an operation in which a group of processes participates, for which the completion stage may or may not finish before all processes in the group have started the operation. A **blocking** operation combines all four stages outlined in Section 1.3.3.

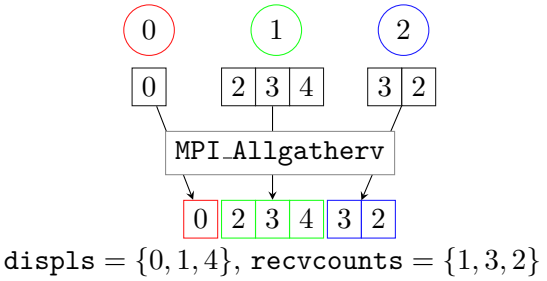

Figure 1.5: Distributing data with `MPI_Allgatherv`

The routine takes an input array, its size, an output array, `displs` and `recvcounts`. The last two arrays contain offset into the output array and the amount of contributed data for each participating process respectively and have to be the same for every participating process.

1.3.5 `MPI_Allreduce`

`MPI_Allreduce` performs parallel reduction among all participating processes. Different reduction operators can be used, such as `MPI_SUM`, `MPI_PROD`, `MPI_MAX` etc. The implementation produced alongside this thesis mostly uses its non-blocking counterpart, `MPI_Iallreduce`. A **non-blocking** operation combines the first two MPI operation stages into a single non-blocking call. It then must be finished by either waiting by calling `MPI_Wait` (or `MPI_Waitall` if waiting on multiple operations). This operation is the backbone of synchronization for parallel SpMV outlined in Section 4.2.

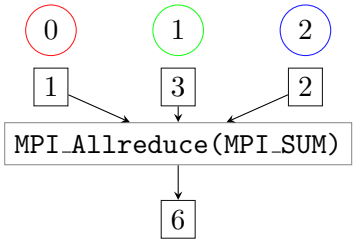

Figure 1.6: Performing parallel reduction with `MPI_Allreduce`

1.4 PETSc

Portable, Extensible Toolkit for Scientific computation (PETSc) is a suite of data structures and routines for scalable solutions of scientific applications modeled by partial differential equations. It supports MPI, and GPUs through CUDA, HIP or OpenCL, as well as hybrid MPI-GPU parallelism [5].

PETSc contains three main algebraic objects `Mat` for matrices, `Vec` for vectors and `IS` for index sets indexing into the previous two. While it also contains many other components, only `Mat` and `Vec` are immediately relevant to this thesis.

1.5 Matrix Market Exchange Format

Matrix Market Exchange Format is a textual format, used as a native format in Matrix Market[3]. It has two flavors, Array format for dense matrices, Coordinate format for sparse matrices [6].

[3]Matrix Market is a repository of sparse matrices, which can be found at https://math.nist.gov/MatrixMarket/

1.5.1 General Format Specification

All Matrix Market exchange format files contain three sections, which must appear in order:

1. **Header** First line of a file, must follow the template:

   ```
   %%MatrixMarket object format [qualifier...]
   ```

   Where object type indicates the mathematical object (e.g., vector or matrix) stored in the file. Type indicates the format (array or coordinate) used to store the object. Qualifiers are used to indicate special properties of the stored object (e.g., symmetry, field). Their number, as well as allowed values, depends on the stored object.

2. **Comments** Zero or more line of comments[4].

3. **Data** Remainder of the file contains data representing the object. The format of data is dependent on the stored object, but for simplicity, each data entry should occupy a single line.

1.5.2 Coordinate Format for Sparse Matrices

Header format for sparse matrices:

```
%%MatrixMarket matrix coordinate <field> <symmetry>
```

Where:

- **field** determines the type and number of values listed for each entry and is one of: Real, Complex, Integer or Pattern.
- **symmetry** determines how to interpret matrix entries and is one of: General, Symmetric, Skew-Symmetric or Hermitian.

While data is specified as:

```
M N L
I J A(I, J)
I J A(I, J)
...
```

First line of data contains exactly three integers:

- `M` - number of rows.
- `N` - number of columns.
- `L` - number of non-zero entries that follows.

[4]Comments are lines starting with %.

The following `L` lines each contain `I` - the row index, `J` - the column index and the corresponding value. Indices are 1-based. Entries not explicitly provided are considered to be zero, except for those known by symmetry.

1.6 HDF5

HDF5 is a data model, library, and file format for storing and managing data. It supports an unlimited variety of data types and is designed for flexible and efficient I/O for high volume and complex data. HDF5 is portable and is extensible, allowing applications to evolve in their use of HDF5. The HDF5 Technology suite includes tools and applications for managing, manipulating, viewing, and analyzing data in the HDF5 format. [1] HDF5 is defined by HDF5 File Format Specification, which specifies the bit-level organization of an HDF5.

1.6.1 HDF5 Abstract Data Model

The HDF5 data model defines building blocks for data organization and specification in HDF5. It's two primary objects are **groups** and **datasets**.

1.6.1.1 Group

HDF5 groups organize data. Every file contains at least a root group. Each group can contain other groups or be linked to objects in other files.

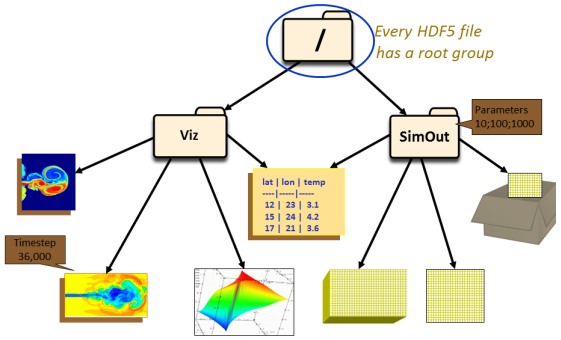

Figure 1.7: HDF5 Group (Source: [1])

Working with groups and their members is similar to working with files and directories in UNIX. Objects are described by giving their pathnames (e.g., `/Viz`).

1.6.1.2 Dataset

Datasets organize and contain the ”raw” data. They consist of raw data and metadata needed to describe it. The metadata needed to describe a dataset consists of datatypes, dataspaces, properties, and optionally attributes.

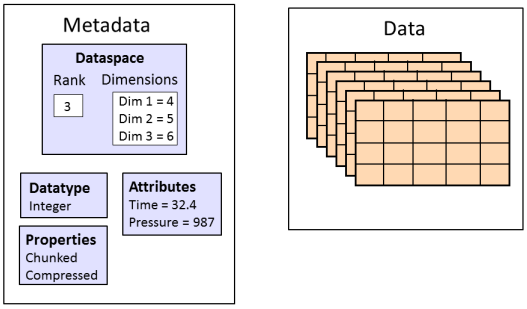

Figure 1.8: HDF5 Dataset (Source: [1])

1.6.1.3 Datatype

Datatypes describe individual elements in dataset. There are two groups of datatypes:

- **pre-defined** - created by HDF5, contain standard datatypes that are stable across platforms (e.g IEEE-754 encoded floating point), as well as native datatypes (e.g. `double` on platform on which the application is running).

- **derived** - created or derived from pre-defined datatypes (e.g. a string).

1.6.1.4 Dataspace

Dataspaces describe the layout of datasets elements. It may be empty (NULL), contain a single element (scalar), or an array[5]. They provide a logical layout of the dataset stored in a file (including rank and dimensions). As well as the application's data buffers and data elements participating in I/O. Thus an application can select subsets of a dataset.

[5]The number of dimensions of the array is referred to as rank of the dataspace.

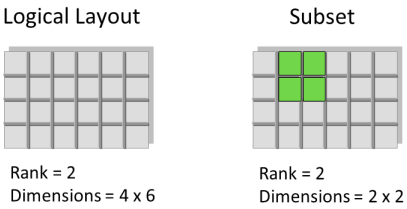

Figure 1.9: HDF5 Dataspace (Source: [1])

1.6.1.5 Attribute

Attributes may optionally be associated with objects. An attribute consists of a name-value pair. They are similar to datasets in that they too have a datatype and dataspace. However, they do not support partial I/O and cannot be compressed or extended.

Chapter 2

Sparse matrix storage formats

Section 2.1 of this chapter introduces sparse matrices. Sections 2.2 and 2.3 provide a quick overview of the two most used storage formats for sparse matrices in Coordinate format (COO) and Compressed Sparse Row (CSR). Compressed Sparse Row 5 introduced in [2] is then outlined in Section 2.4.

2.1 Sparse matrices

A matrix is a rectangular array of numbers. The numbers in the array are called the entries in the matrix [7]. Matrix entries are usually addressed by the column and row in the rectangle they occupy.

In numerical analysis and scientific computing, a sparse matrix or sparse array is a matrix in which most of the elements are zero. There is no strict definition regarding the proportion of zero-value elements for a matrix to qualify as sparse, but a common criterion is that the number of non-zero elements is roughly equal to the number of rows or columns. By contrast, if most elements are non-zero, the matrix is considered dense. The number of zero-valued elements divided by the total number of elements (e.g., m × n for an m × n matrix) is sometimes referred to as the sparsity of the matrix. [8] The number of non-zero elements of a sparse matrix is usually denoted as n_{nz} number of columns as n and the number of rows as m.

To better illustrate the storage formats, let us define sparse matrix A.

$$A = \begin{pmatrix} 1 & 0 & 2 & 0 \\ 0 & 3 & 0 & 0 \\ 4 & 0 & 5 & 0 \\ 6 & 0 & 0 & 7 \end{pmatrix}$$

2.2 Coordinate (COO)

Coordinate storage format, commonly referred to as COO, is the simplest of the three formats presented in this chapter. COO stores the matrix in a structure consisting of three arrays of length n_{nz}[9]:

vals an array containing all the real (or complex) values of the nonzero elements of A in any order.

row_idx an integer array containing their row indices.

col_idx a second integer array containing their column indices.

vals	1	2	3	4	5	6	7
row_idx	0	0	1	2	2	3	3
col_idx	0	2	1	0	2	0	3

Table 2.1: A stored in COO format

As shown in Algorithm 1, to perform SpMV with a matrix stored in the COO format, it is necessary to iterate over all non-zero elements, extracting their value, column index, and row index from the relevant arrays in the COO structure. Then performing the multiplication `value` · `x[col]` and finally add the sub-result to `y[row]`. This algorithm is not well suited for parallelization, as all the additions to the result vector `y` would have to be atomic operations. It may also degrade single-thread performance as the order of non-zero elements is not specified, meaning accesses to `x` as well as `y` are random and thus may not have the best cache locality.

Algorithm 1 SpMV for matrix stored in COO format

```
function SPMV(A, x)
    y ← result vector
    for i = 0; i < nnz; ++i do
        value ← A.vals
        col ← A.col_idx[i]
        row ← A.row_idx[i]
        y[row] += value · x[col]
    end for
     return y
end function
```

2.3 Compressed Sparse Row (CSR)

Compressed Sparse Row is probably the most popular format for storing general sparse matrices. [9] Similarly to the coordinate format, it too consists of three arrays.

vals a real array of size n_{nz} containing all the real (or complex) values of the nonzero elements of A stored in row major order.

col_idx a second integer array of size n_{nz} containing their column indices.

row_ptr an integer array of size $m + 1$ containing offsets into `vals` and `col_idx`, at which each row of the matrix begins.

Storing only $m + 1$ elements in `row_ptr` array, leads to non-negligible storage savings.

vals	1	2	3	4	5	6	7
col_idx	0	2	1	0	2	0	3
row_ptr	0	2	3	5	6		

Table 2.2: A stored in CSR format

SpMV with matrix in CSR format lends itself to parallelization better than the COO format. Since CSR groups the non-zero elements by their rows, the outer loop of Algorithm 2 can be parallelized with no synchronization necessary (no overlapping outputs in `y`). The issue with the parallelized version of this algorithm is load-balancing, as some rows may have disproportionately more non-zero elements than others.

Algorithm 2 SpMV for matrix stored in CSR format

```
function SPMV(A, x)
    y ← result vector
    for row = 0; row < m; ++row do
        start ← A.row_ptr[row]
        stop ← A.row_ptr[row + 1]
        row_result ← 0
        for i = start; i < stop; ++i do
            col ← A.col_idx[i]
            value ← A.vals[i]
            row_result += value · x[col]
        end for
        y[row] ← row_result
    end forreturn y
end function
```

2.4 Compressed Sparse Row 5 (CSR5)

To achieve near-optimal load balance for matrices with any sparsity structures, CSR5 partitions all nonzero entries to multiple 2D tiles of the same size. It has two tuning parameters: ω and σ, where ω is a tile's width and σ is its height [2].

Further, extra information is needed to efficiently compute SpMV. For each tile, a tile pointer `tile_ptr` and a tile descriptor `tile_desc` are introduced. Meanwhile, the three arrays, i.e., row pointer `row_ptr`, column index `col_idx` and value val, of the classic CSR format are directly integrated. The only difference is that the `col_idx` data and the `vals` data in each complete tile are in-place transposed (i.e., from row-major order to column-major order) for coalesced memory access from contiguous SIMD lanes.

Each column of a tile has three characteristics:

- `y_offset` - relative offset into the Y for a column of tile (equal to number of rows that started in previous columns).
- `scansum_offset` - number of consecutive empty columns to the right of current column.
- `bit_flag` - of size σ where $i-th$ bit is set if $i-th$ value of this column is the first non-0 entry of its row or it is the $0th$ bit of $0th$ column.

The tile further may have an `empty_offset` array of size $\mathcal{O}(\omega \cdot \sigma)$, if it contains empty rows, because `y_offset` will be incorrect for such tile. Thus the correct offsets into Y for each segment of such a tile are stored in in `empty_offset`. The actual size of this array is number of segments in a tile (the number of bits set to 1 in `bit_flag`).

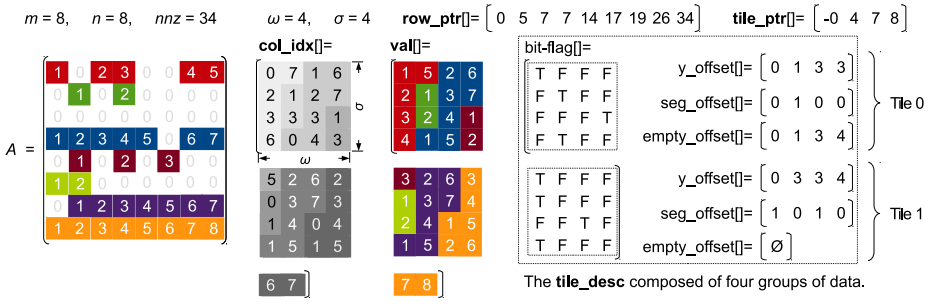

Figure 2.1: Matrix A stored in CSR5 format (Source: [2]).

Section 3.3 outlines the algorithm for performing parallel SpMV using matrices stored in CSR5 format.

Chapter 3

Implementation

This chapter introduces `dim`. `dim` is a library/toolkit which is the result of the research done for this thesis. `dim` is implemented in C++20 using OpenMP and MPI introduced in Sections 1.2 and 1.3 respectively.

While the implementation supports COO and CSR as storage formats for sparse matrices, the main storage format used is CSR5 described in Section 3.1. The SpMV algorithm for matrices stored in CSR5 format, introduced in [2] is further described in Section 3.3.

Lastly, an efficient on-disk storage format used by MATLAB for CSR matrices, storing the matrices in binary format using HDF5, described in Section 3.4 and its extension to support matrices stored in CSR5 is outlined in Section 3.5.

3.1 CSR5

A common issue of parallel SpMV is load balancing. Storing matrices in CSR5 format avoids this by grouping data into tiles of a fixed size. Furthermore, the tiles are independent, allowing for a high degree of parallelism and small critical sections. For this reason `dim`, stores all matrices in CSR5 format and implements sparse vector multiplication algorithm introduced in [2].

Since the implementation of CSR5 SpMV in `dim` utilizes AVX2 instructions, it has some preconditions. Namely alignment of data passed to the AVX2 intrinsics. Most AVX2 instructions either require data to be aligned on a 32-byte address or have significantly worse performance for unaligned data.

To facilitate this, C++ offers a customization point in its own contiguous data container `std::vector`. By specifying an allocator, which returns allocated addresses with a specified alignment as the second template argument, the implementation can retain the interface of vector and satisfy preconditions of AVX2 instructions. Any usage of `vector` in this thesis in code examples will refer to a `std::vector` with a custom-aligned allocator.

3.1.1 In memory layout

As described in 2.4, CSR5 storage format, retains the original three arrays of CSR, with **column index** and **values** being in-place transposed. This forms the first part of CSR5 structure in dim.

The CSR5 structure is templated on floating point type it should use to represent values, as well as on unsigned type used to represent indices. It also uses concepts, a feature introduced in C++20, to ensure the passed in types satisfy these constraints. With plain `typename`, the UnsignedType could be an `int` or `FloatingType` a `std::string` and the compiler would only produce an error if a substitution failed. It is also templated on `Sigma` and `Omega`, the two tuning parameters of CSR5, passed in as non-type template parameters. The defaults are `FloatingType = double`, `Sigma = 16`, `Omega = 4` and `UnsignedType = uint32_t`. This limits the size of matrix to be $2^{32} \times 2^{32}$ but can be easily changed to `uint64_t` or another type if storage for a bigger matrix is required.

```
template<std::floating_point FloatingType = double,
         size_t Sigma = 16,
         size_t Omega = 4,
         std::unsigned_integral UnsignedType = uint32_t>
struct csr5 {
    // same as CSR.
    UnsignedType         num_cols;
    vector<FloatingType> vals;
    vector<UnsignedType> col_idx;
    vector<UnsignedType> row_ptr;
    // explained in following section.
    csr5_info            csr5_info;
};
```

Figure 3.1: Pseudo-code for structure holding CSR5 matrix

3.2 CSR5 info

`csr5_info` structure contains CSR5 specific information, introduced in 2.4. Namely, **tile pointer**, **tile descriptor** and optionally **empty offset pointer** and **empty offset** arrays. Each of these will be explained in a separate section.

3.2.1 Tile pointer

The tile pointer array contains the index of the row of the first element. It works similarly to a CSR **row pointer**. The range of rows covered by a tile is obtained by querying `tile_ptr[tid]` and `tile_ptr[tid + 1]` for the first and last rows respectively (range is open from the top).

The most significant bit of the tile pointer, used as a flag, is set if a tile is **dirty**. A tile is dirty if it contains any empty rows. This flag bit must be stripped when the pointer is used for indexing into **row pointer** array. The pointer is wrapped by a structure with `is_dirty` and `idx` methods to force the call site to specify the required usage explicitly, making the interface less error-prone.

```
struct tile_ptr {
    bool is_dirty();
    UnsignedType raw();
    UnsignedType idx();
};
```

Figure 3.2: Tile pointer pseudo-interface

3.2.2 Tile descriptor

Amid the new additions to C++20 are bit-manipulation functions in `<bit>` header. One of the new functions is `std::bit_width` which takes N and returns M such that values up to N can be stored in M bits. This function can be used in conjunction with bit-fields and non-type template parameters to pack descriptors very efficiently while retaining readability when using them. Since `y_offset` may be at most $\sigma \cdot \omega$ (the number of new sections is at most the same as the number of elements in a tile) and `scansum_offset` may be at most $\omega - 1$ (as it is the number of empty consecutive columns to the right of current column), the descriptor column and the descriptor itself can be defined as shown in Figure 3.3.

As CSR5 tile descriptors contain bit flag maps, `std::popcount` can be used to obtain the number of set bits, denoting the number of sections started in a column. Using this function allows the compiler to optimize the call to the instruction of the same name if the target ISA supports it.

```
template<size_t Sigma, size_t Omega>
struct descriptor_column {
    StorageT y_offset: bit_width(Sigma * Omega);
    StorageT scansum_offset: bit_width(Omega);
    StorageT bit_flag: Sigma;
};

template<size_t Sigma, size_t Omega>
struct descriptor {
    descriptor_column<Sigma, Omega> columns[Omega];
};
```

Figure 3.3: Tile descriptor pseudo-code.

Previous research about CSR5 [2] concluded, that for modern CPUs with AVX2 extensions, the optimal CSR5 tuning parameters are $\sigma = 16$ and $\omega = 4$.

3.2.3 Empty offset

With **empty offset** being dependent on number of sections in a tile (and being completely absent in tiles which **don't contain empty rows**), it would be inefficient to store it in the descriptor itself. To avoid too many allocations, two supporting arrays are added. An `empty_offset_ptr` of size `num_tiles + 1`, which contains offsets for each tile into the second array, `empty_offset` at which their `empty_offset` array begins. Thus for each tile, it's empty offset array is defined as `empty_offset[empty_offset_ptr[i]:empty_offset_ptr[i+1]]`.

The `csr5_info` structure combines these four arrays.

```
struct csr5_info {
    vector<tile_ptr>        tile_ptr;
    vector<tile_descriptor> tile_desc;
    vector<UnsignedType>    empty_offset_ptr;
    vector<UnsignedType>    empty_offset;
};
```

Figure 3.4: `csr5_info` pseudo-code.

3.3 Sparse matrix vector multiplication

`dim` uses CSR5 storage format, described in Section 2.4, which was purposefully designed to achieve better performance when performing parallel SpMV. This section provides an overview of this parallel SpMV algorithm (introduced in [2]) for CSR5 when performed on a multi-threaded CPU, which supports AVX2 extensions.

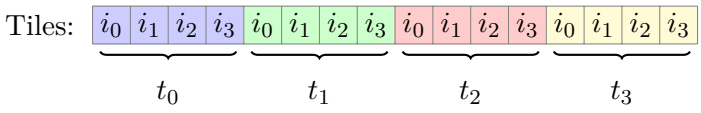

Figure 3.5: Processing tiles using 4 threads

Using N threads, each thread gets a contiguous chunk of tiles of the same size (except possibly the last thread) as shown in Figure 3.5. The computational kernel of CSR5 SpMV can be split into three main parts. First, each thread processes its tile chunk. Then the necessary synchronization of sub-results from all threads is performed. Lastly, the tail partition of the matrix, which was not in any tile (at most $\sigma \cdot \omega$ elements) is processed.

Algorithm 3 Corpus of the CSR5 SpMV

```
function SPMV(A, x)
    y ← result vector
    calib ←
    spmv_full_tiles(A, x, calib, y)
    spmv_sync(calib, y)
    spmv_tail(A, x, y)
     return y
end function
```

3.3.1 Processing full tiles

Each thread processes its assigned tile chunk sequentially. The notion of **sub-segments** within a tile column needs to be established before tile processing can be described. [2] defines three types of sub-segments.

A green sub-segment is a segment that contains all of the elements of a single row. Meaning, it starts with a set bit in `bit_flag` and ends with another set-bit (denoting start of a new row) in `bit_flag`. A blue sub-segment is a segment unsealed from the bottom (column does not end with a set bit in `bit_flag`). Red sub-segments are segments unsealed from the top (column does not start with a set bit in `bit_flag`). A tile column with no set bits is also marked as a red segment. Green sub-segments require no further

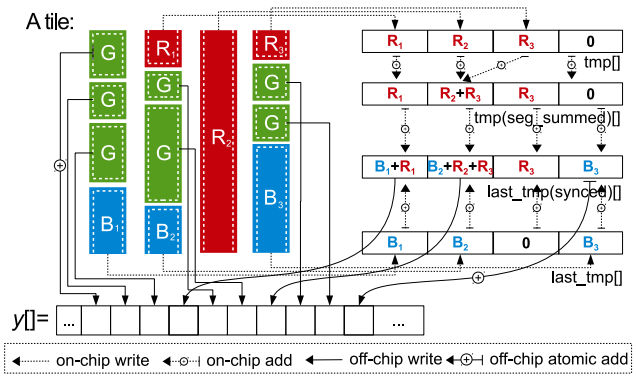

Figure 3.6: Synchronization of sub-segments within a tile (Source: [2]).

synchronization. Their sum can be written to the result vector y directly by combining `tile_ptr` and `y_offset` information. Red and blue sub-segment sub results must be synchronized as they do not form a complete segment.

Algorithm 4 Processing assigned tile chunk CSR5 SpMV

```
function SPMV_FULL_TILES(A, x, calib, y)
    for tile_id ∈ assigned tiles do
        row ← A.tile_ptr[tile_id]
        desc ← A.tile_desc[tile_id]
        direct ← desc starts with red sub-segment
        spmv_single_tile(A, desc)
        if first tile in assigned tiles and !direct then
            calib[tid] ← first output element
        else
            if direct then
                y[row] ← first output element
            else
                y[row] += first output element
            end if
        end if
    end for
end function
```

Each tile having 4 columns (ω) and 16 rows (σ), the thread will perform σ iterations, processing ω non-zero elements in each iteration using SIMD instructions, in other words, each column is assigned a SIMD-lane and processed in parallel. This is the reason for $\omega = 4$ as AVX2 instructions can work on 4 double precision floating point numbers at once as well as the reason for data being in-place transposed. Algorithm 5 shows how a single tile column is processed.

Algorithm 5 Single tile column processing CSR5 SpMV

```
function SPMV_SINGLE_TILE(A, desc)
    any_segment ← desc.bit_flag[0]
    sum ← tile_vals[0] · x[tile_col_idx[0]]
    red_segsum ← 0
    for tile_row ∈ range(1, σ) do
        segment_end ← desc.bit_flag[tile_row]
        if segment_end then
            if any_segment then
                store to y, green segment
            else
                red_segsum ← sum
            end if
            sum ← 0
            any_segment ← true
        end if
        sum += tile_vals[tile_row] · x[tile_col_idx[tile_row]]
    end for
```

▷ Here, `sum` will hold the blue sub-segment and `red_segsum` will hold the red sub-segment partial results. Except when no no segment has started (`any_segment` is false), then `sum` holds the red-subsegment partial result and there is no blue sub-segment. The synchronization between lanes is shown in Figure 3.6.

```
end function
```

3.3.2 Synchronizing overlapping elements

CSR5 SpMV minimizes the critical sections. The tiles within the chunk assigned to a thread do not have to be synchronized since they are processed sequentially in the context of the owning thread. Only a single element of y may be overlapping between neighboring chunks of tiles, which only happens if the first tile of the chunk starts with a red sub-section. Synchronization is a two-step process. First, each thread stores its first output element in a calibrator (see Algorithm 4) during the parallel computation. Second, after the parallel part of the algorithm is over, each threads calibrator is added to the result to perform the final synchronization as shown in Algorithm 6.

3.3.3 Tail partition

If the number of non-zero elements for a matrix isn't divisible by $\sigma \cdot \omega$, there is a "tail" partition which does not form a full tile. The algorithm introduced in Section 2.3 for CSR matrices is used to process this tail partition.

Algorithm 6 Synchronizing the overlapping elements in y CSR5 SpMV

```
function SPMV_SYNC(calib, y)
    for tid ∈ threads do
        y_idx ← tile_ptr of first tile assigned to thread
        y[y_idx] += calib[tid]
    end for
end function
```

3.4 Selecting on disk storage format

The matrices for which parallelized (and distributed) matrix-vector multiplication has the biggest benefit, have large on-disk size. This section compares Matrix Market Exchange Format [6] introduced in Section 1.5 and HDF5 backed format introduced by Matlab 7.3 for storing sparse matrices, comparing both on-disk size and throughput measured by the number of elements loaded in a second.

3.4.1 Matrix Market storage format

Matrix Market exchange format is designed to be easy to parse. It provides a library to parse the header and matrix dimensions [10]. The numerical entries need to be parsed manually, for which most implementations use variations of `scanf`.

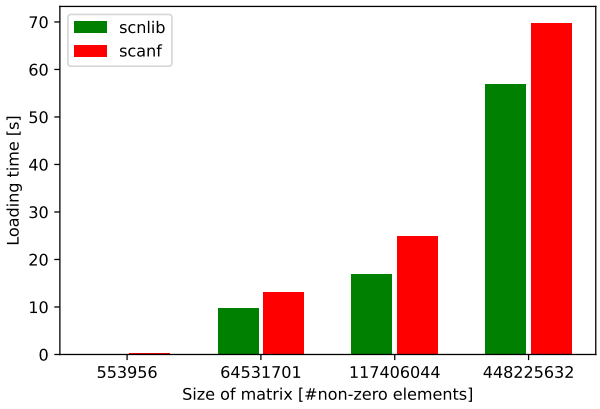

Figure 3.7: Loading times of MMEF

Initial implementation in dim used `scanf` as well. However, this proved to be a bottleneck as the average throughput was only 4.52×10^6 elements/s. Profiling has shown that parsing the string representation was the most computationally intensive step as 97.89% of time spent was in `scanf` itself. To improve the

loading times, to shorten the iteration times the process of benchmarking, an implementation using `scnlib`[6] was created. With an average throughput of 6.07×10^6 elements/s it resulted in 1.34x speed up, which was still a major development bottleneck.

3.4.2 Matlab compatible HDF5 sparse matrix storage

Loading matrices stored in binary format using HDF5 file format introduced in 1.6 is supported by MATLAB, as well as PETSc[7]. The matrices are stored in CSR format, represented as a single HDF5 group containing 3 datasets:

- data - values of elements.
- aj - column index.
- ir - row pointer.

And a single attribute MATLAB_sparse containing the number of columns of the stored matrix.

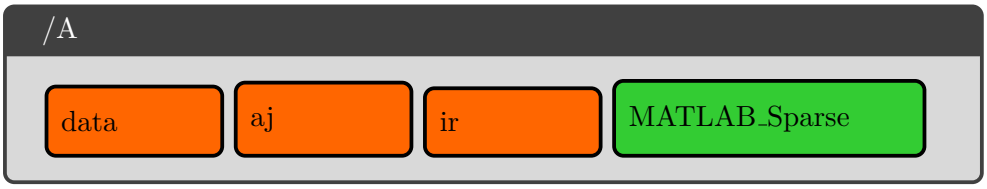

Figure 3.8: Layout of a sparse matrix stored as a HDF5 group

Storing the matrices in this format yielded a throughput of 1.10×10^8 elements/s which is 24.36x speedup over `scanf` based implementation of reading matrices stored in Matrix Market Exchange Format and 18.12x speedup over `scnlib` implementation.

Since MMEF uses the coordinate format to store sparse matrices, it may omit elements from symmetrical matrices[8], which can result in significant on-disk size reduction. Matrices 2, 3 and 4 in Figure 3.10 are symmetrical. Thus their on-disk size can be smaller with MMEF compared to HDF5.

However, the HDF5 format supports dataset compression by LZMA. Setting chunk size to 4096 and compression level to 5 for each of the three datasets, it is possible to achieve considerable storage savings (the largest matrix is 7.8x smaller stored as compressed HDF5 compared to MMEF) while maintaining better loading performance, with average throughput being 2.71×10^7 elements/s, about 4x slower than HDF5 with no compression, but still 4 and 6 times faster than `scnlib` and `scanf` implementations respectively.

[6] `scnlib` is a modern C++ alternative to `scanf`.
[7] PETSc has to be configured with HDF5 support when it is built
[8] Those can be inferred from their symmetrical counterparts.

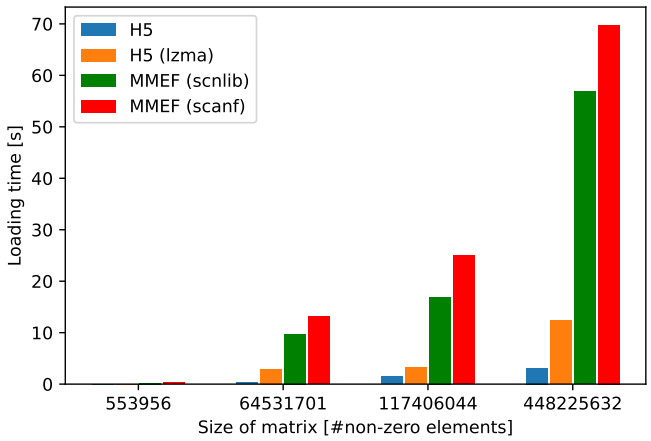

Figure 3.9: Load times of HDF5 and MMEF storage formats.

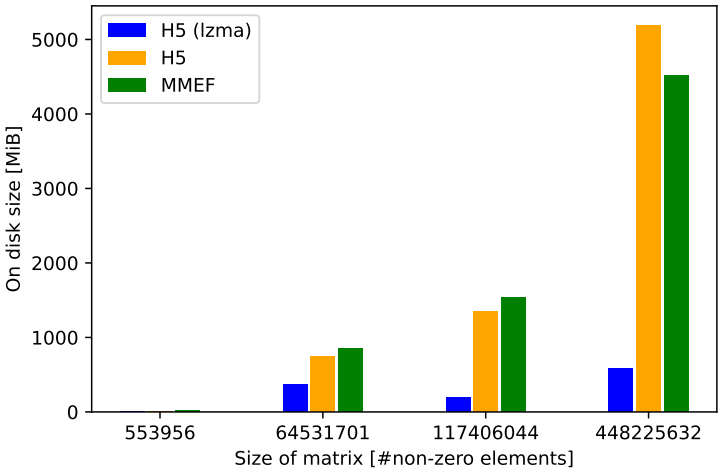

Figure 3.10: On-disk size of HDF5 and MMEF storage formats.

3.5 On disk storage for CSR5 matrices

With HDF5 proving to be the faster alternative to textual MMEF format, it is also used for storing sparse matrices in CSR5 format.

Each of the sequential data members row_ptr, col_idx, vals, tile_ptr, empty_offset, empty_offset_ptr, are stored as one dimensional datasets of same name and same datatype[9]. num_cols is stored as an attribute.

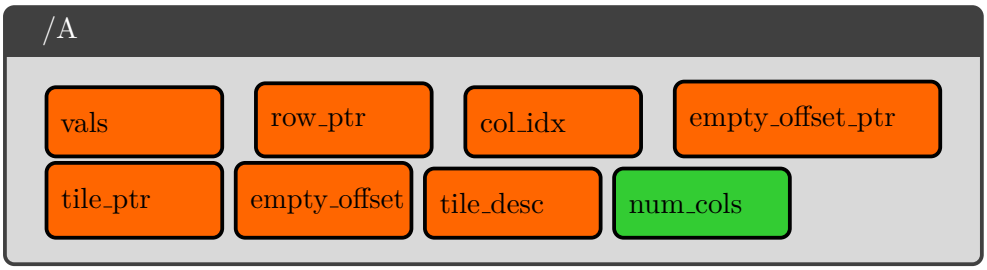

Figure 3.11: Layout of a sparse matrix stored as a HDF5 group

tile_desc is stored as a one-dimensional dataset, with datatype being an array of 4 little-endian unsigned 32-bit integers. In essence, storing the data to the HDF5 file as they appear in memory. This decision does put a soft requirement on the code storing the CSR5 matrix to a file and the code loading it compiled by the same version of a compiler, as the bit-field in-memory layout is implementation-defined but allows faster loading. Furthermore, both Clang and GCC implement this in the same way, so the requirement is only a soft one, as compatibility can be easily tested.

[9] `H5T_STD_U32LE` for all the index arrays and `H5T_IEEE_F64LE` for values

Chapter 4

Distributed Sparse Matrix-Vector Multiplication

Using the principles described in Section 3.3, sparse matrix-vector multiplication can be distributed across multiple processes. The tiles are first divided among processes as they would be among threads in a single process implementation (see Figure 3.5). Then, these chunks assigned to processes are further subdivided among the threads of each process. Furthermore, the same synchronization principles apply, thus only processes that have the first tile starting with red-subsegment (as defined in 3.3) need to synchronize with the process to the left[10]. This implementation will be referred to as D-CSR5.

Section 4.1 outlines the algorithm used to load the chunks for each process using HDF5 and MPI-IO. Then, the synchronization algorithm for edge elements of the result vector as well as an algorithm for distributing the result vector amongst the processes is introduced in 4.2.

4.1 Loading the matrices

Since CSR5 partitions the non-zero elements of a matrix into tiles of size $\sigma \cdot \omega$, each node can load only data relevant for tiles assigned to it. This is enabled by two features of HDF5, the first of which is reading hyperslabs. A hyperslab is a subset of an HDF5 dataset, so nodes do not have to read whole datasets.

The second feature enabling this is HDF5s ability to utilize MPI-IO, allowing concurrent reads from every process rather than reading the matrix in the main process and distributing it amongst the rest of the processes. This imposes a minor limitation on how the matrix can be stored, or rather how the dataset needs to be structured. Since MPI-IO was introduced in MPI revision 2.0 [11] in 1997, its API takes the size of data to be read as an integer argument, thus limiting the maximum amount of data read in a single call to 2GiB. The

[10] In the sense of MPI topology.

dataset needs to be chunked to avoid exceeding this limit [11], else HDF5 tries to read all of it at once, resulting in an error in MPI-IO.

The first step of loading a CSR5 matrix is obtaining the number of tiles of the matrix. This can be done by querying the dimensions of `tile_desc` dataset, which contains a descriptor for each tile. The number of tiles is then divided by the number of processes, meaning every process gets equal number of tiles (bar the last process, which can have less), which in turn means an equal number of non-zero elements.

Algorithm 7 Computing partition size for each process

```
function CALCULATE_PARTITION(tile_count, proc_id, proc_count)
    partition_size ← ⌈tile_count / proc_count⌉
    first ← partition_size · proc_id
    count ← min(partition_size, tile_count − first)
     return (first, count)
end function
```

Then, tile descriptors from `tile_desc` dataset, tile pointers from `tile_ptr` dataset and empty offsets pointers from `tile_desc_offset_ptr` can be loaded. Sizes of slabs are $\#tiles_{proc}$, $\#tiles_{proc} + 1$ and $\#tiles_{proc} + 1$ respectively, as `*_ptr` datasets use $n + 1^{st}$ element to denote end of values belonging to n^{th} element. Lastly empty offsets from `tile_desc_offset` are loaded. This forms complete information about CSR5 tiles, and as such, it is stored in a separate structure named `csr5_info`. This information is useful even on its own, for example queries about a certain tile, matrix element or row of matrix and its parent tile can be made just with this information.

Algorithm 8 Loading CSR5 info

```
function LOAD_CSR5_INFO(datasets, first, count)
    tile_desc ← datasets["tile_desc"][first:count]
    tile_ptr ← datasets["tile_ptr"][first:count+1]
    empty_off_ptr ← datasets["empty_offset_ptr"][first:count+1]
    empty_off ← datasets["empty_offset"][e_ptr[0]:e_ptr[-1]]
end function
```

Lastly, slab of `row_ptr` is loaded, starting at `tile_ptr[0]` and ending at last output row of chunk of this matrix. This varies, if it is the last process, the last output row is simply last row of the matrix. For every other tile, `y_index` in last column + number of bits set in the bit flag of last column (equal to number of started rows) gives relative offset, which, when added to `tile_ptr` of the tile will result in absolute offset in y where next row would be. This model maps onto C++ iterator model, where `end` denotes element

[11] experimentally, chunks of 1000000 elements or 8MiB of double-precision floating-point numbers performed best

one past the end of the range. With this information, `row_ptr` can be loaded. Since, `row_ptr` has the same meaning as in CSR, hyperslabs for values from `vals` dataset, and column indices from `col_idx` dataset are both starting at `row_ptr[0]` and ending at `row_ptr[-1]`.

Algorithm 9 Loading CSR data

```
function LAST_OUT_IDX
    lid ← 'last tile id'
    last_desc ← tile_desc[lid]
    secs_starting ← popcount(last_desc[lid].bit_flag)
    rel_off ← last_desc[lid].y_index + secs_starting
    if dirty(last_desc) then
        empty_start ← empty_off_ptr[lid]
        rel_off ← empty_off[empty_start + rel_off]
    end if
     return tile_ptr[lid] + relative_offset
end function
function LOAD_CSR_DATA(datasets, first, count)
    fr_id ← tile_ptr[0]
    lr_id ← last_out_idx()
    row_ptr ← datasets["row_ptr"][fr_id:lr_id]
    values ← datasets["vals"][row_ptr[0]:  row_ptr[-1]]
    col_idx ← datasets["col_idx"][row_ptr[0]:  row_ptr[-1]]
end function
```

4.2 Synchronization

Section 3.3 has established that a tile can share at most one output element with its neighbor, so a process has to synchronize at most two elements in its output range in y. The synchronization mechanism is explained in Section 4.2. If all processes need the whole result vector, it can be distributed as explained by Section 4.2.2.

To illustrate, let A be a 4×4 square matrix with 8 non-zero elements, multiplied by a vector x, while SPmV is distributed across four processes labeled p1-p4 and let $\sigma = 2$ and $\omega = 1$. Thus, each process gets one CSR5 tile containing 2 non-zero elements.

4.2.1 Synchronizing overlapping output ranges

Let **output index** be an index into result vector y of sparse matrix vector multiplication $Ax = y$ and **output range** be a pair of output indices, denoting first and last output index for each process.

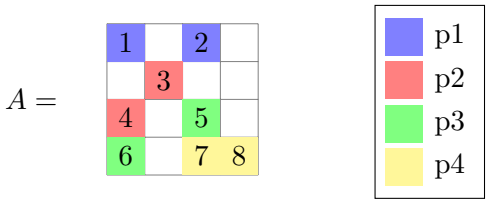

Figure 4.1: Distribution of A across 4 processes

$$p_1 = \{0, 0\} \quad p_2 = \{1, 2\} \quad p_3 = \{2, 3\} \quad p_4 = \{3, 3\}$$

$$y =$$

Figure 4.2: Output ranges for processes p1-p4

Neighboring processes may share an output index. In the case of $Ax = y$ established in the previous section, y_2 and y_3 are partially computed by processes p2, p3 and p3, p4 respectively. An ownership model needs to be defined before the synchronization can be described. The process with the lowest rank (in terms of MPI ranks) that writes to an output index owns the element of y. Thus y_2 is owned by p2 while y_3 is owned by p3.

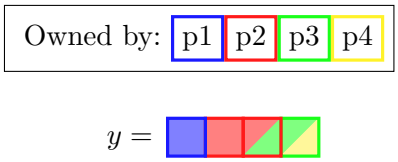

Figure 4.3: Ownership of y

Each process has two MPI communicators to synchronize the elements. One for synchronizing the element it owns referred to as OWN, and one for synchronizing the element it does not own referred to as LEFT. Either of these communicators can be `MPI_COMM_NULL` if no synchronization is necessary for the owned (or non-owned) element.

The first step in creating synchronization communicators is participating processes exchanging their output ranges. Output ranges consist of indices of the first and last output elements for each process. From these, each process can check if it needs to synchronize to the left and if it needs to create a communicator on which processes to the right of it will sync.

`MPI_Comm_Split` splits a communicator according to a `color` and a `key`. `color` is used to group processes, and `key` is used to create ordering within processes of same `color`. Furthermore, if only a single process is present, the returned communicator will be `MPI_COMM_NULL`. Thus if all processes first

Output ranges: $\{\{0,0\},\{1,2\},\{1,2\},\{2,2\}\}$
Left sync rank: $\{0,1,1,3\}$

Figure 4.4: Pre-computation step of synchronizing y

compute the process which will own the first output index in their output range, then it is possible to iterate over every rank and allow every process to establish an owning communicator in the iteration in which $i = rank$ and for all the processes which need to synchronize the element this process owns, to register to this communicator. If no process registers, the communicator returned is `MPI_COMM_NULL`, and no synchronization is performed.

Algorithm 10 Creating communicators for syncing edge elements

```
function CREATE_SYNCS(my_rank, comm_size)
    output_ranges ← ranges from all processes
    syncs_to ← my_rank
    this_node_range ← output_ranges[my_rank]
    while syncs_to ≠ 0 and elements overlap do
        syncs_to ← syncs_to − 1
    end while
    for rank = 0; rank < comm_size; ++rank do
        if rank ≡ my_rank then
            own_sync ← MPI_Comm_Split(comm, my_rank, my_rank)
        else if rank ≡ syncs_to then
            left_sync ← MPI_Comm_Split(comm, syncs_to, my_rank)
        else
            MPI_Comm_Split(comm, MPI_UNDEFINED, MPI_UNDEFINED)
        end if
    end for
end function
```

Depending on values of LEFT and OWN communicators the processes can be split into 4 categories:

- (NULL, NULL) - self-contained, not sharing any output indices (p1).
- (NULL, !NULL) - doesn't share first output element (p2).
- (!NULL, NULL) - doesn't share last output element (p4).
- (!NULL, !NULL) - shares both first and last elements (p3).

With the communicators established, processes that have at least one non-NULL communicator call `MPI_Allreduce` with either first, last, or both elements to synchronize the edge elements in every participating process.

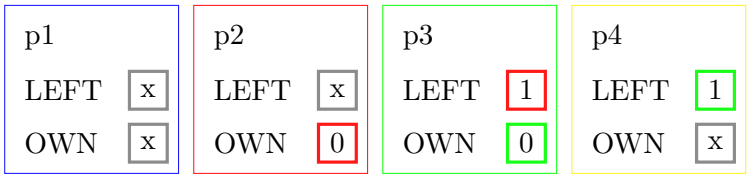

Figure 4.5: Synchronization communicators

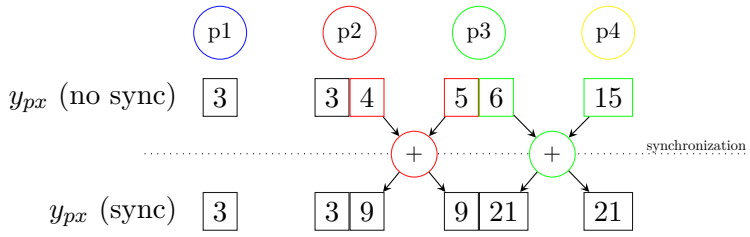

Figure 4.6: Synchronization of result of $Ax = y$ assuming $x = \mathbb{1}$

4.2.2 Distributing the result

After synchronizing overlapping output ranges, the result can be distributed to every process. Ownership rules established in the previous section apply when distributing the result vector. The first process that writes an element owns it and is the only process broadcasting it to others.

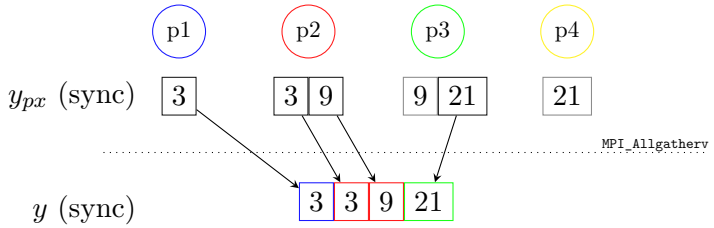

Figure 4.7: Distribution of result of $Ax = y$ assuming $x = \mathbb{1}$

Before partial results can be distributed, output ranges must be exchanged (which is done for the edge element synchronization). From these ranges, an array of offsets into the complete result vector are computed. Each process either broadcasts its whole output range or skips the first element of the range if it is owned by a process to the left. The actual distribution of results is performed by calling `MPI_Allgatherv`, introduced in Section 1.3.5, with offsets and sizes computed as explained previously.

Algorithm 11 Synchronizing partial results across processes

```
function COMPUTE_OFFSETS
    output_ranges ← ranges from all nodes
    recvcounts[0] ← output_ranges[0].count
    displs[0] ← output_ranges[0].first_idx
    for rank = 1; rank < comm_size; ++rank do
        skip_first ← range overlaps
        if skip_first then
            output_ranges[rank].first_idx += 1
        end if
        recvcounts[rank] ← output_ranges[rank].count
        displs[rank] ← output_ranges[rank].first_idx
    end for
     return recvcounts, displs
end function
function SYNC(owned_partial_result, full_result, comm)
    displs, recvcounts ← compute_offsets()
    MPI_Allgatherv(owned_partial_result, full_result, recvcounts, comm)
end function
```

Chapter 5

Benchmarks

To measure real-world performance of distributed sparse matrix-vector multiplication three implementations of numerical solvers using conjugate gradient method briefly explained in Appendix A, were produced. Single and multi process implementations using the `dim` toolkit and a `PETSc` based multi process implementation.

5.1 Specification

Each implementation will load matrix A and perform 100 iterations of conjugate gradient method, trying to find x satisfying $Ax = \mathbb{1}$.

Each step of conjugate gradient iteration is kept track of, and the total time for each step is output in a JSON file.

5.1.1 Benchmark data

Three square matrices were selected for benchmarking.

name	**n**	n_{nz}	**CSR5 on-disk size** [GiB]
nlpkkt240	27993600	774472352	9.1
GAP-web	50636151	1930292948	22.6
GAP-kron	134217726	4223264644	49.6

5.1.2 Compilation options

The benchmarks were run on the RCI cluster, using AMD nodes.

CPU: `2 x AMD EPYC 7543`
Network: `200GbE InfiniBand EDR`
RAM: `1TB`
Storage: `8TB NVMe`

Compiler: `GCC 10.3`
Optimization level: `-O3`
March: `native(znver2)`

5.2 Single process implementation

Single process implementation uses CSR5 format for storing the sparse matrix and the SpMV algorithm described in Section 3.3 for the $A \cdot s$ step. It was benchmarked with 16, 32, 64, and 128 CPUs available for each input matrix. The naming of steps in the graphs follows the naming scheme introduced in Section A.1.

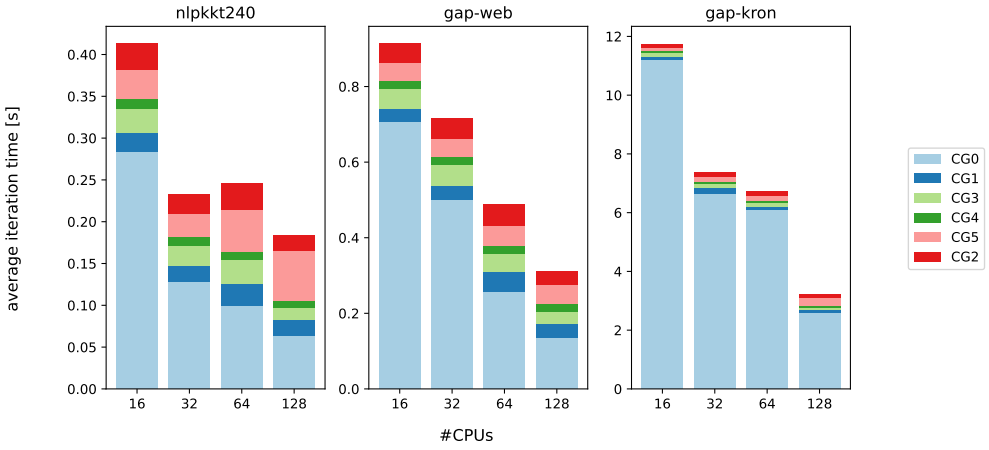

Figure 5.1: Scaling of single process implementation CG with CPU count

Benchmark results show that the sparse matrix-vector multiplication As is the most computationally intensive step as it takes 68.5, 77.5, and 95.4% of total iteration time for *nlpkkt240*, *gap-web*, and *gap-kron* matrices respectively when the single process implementation is running on 16 threads.

It can further be observed that sparse matrix-vector multiplication scales well with number of CPUs available.

5.2.1 IO

With the benchmarked matrices ranging in size from 9GiB up to 50GiB, loading the matrices may become a non-negligible part of total computation time. The following graphs show the impact of IO on total run-time of benchmark as well as achieved throughput using the fastest single process configuration with 128CPUs available.

By using HDF5 to store matrices in binary format and utilizing fast NVMe storage available on RCI nodes, the impact of IO on total run time can be minimized. However even after doing so, for `GAP-web` matrix, loading the matrix takes up 26% of the total benchmark runtime.

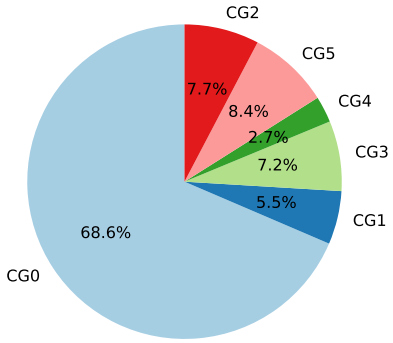

Figure 5.2: Percentage of time spent in steps of CG (nlpkkt240, 16CPUs)

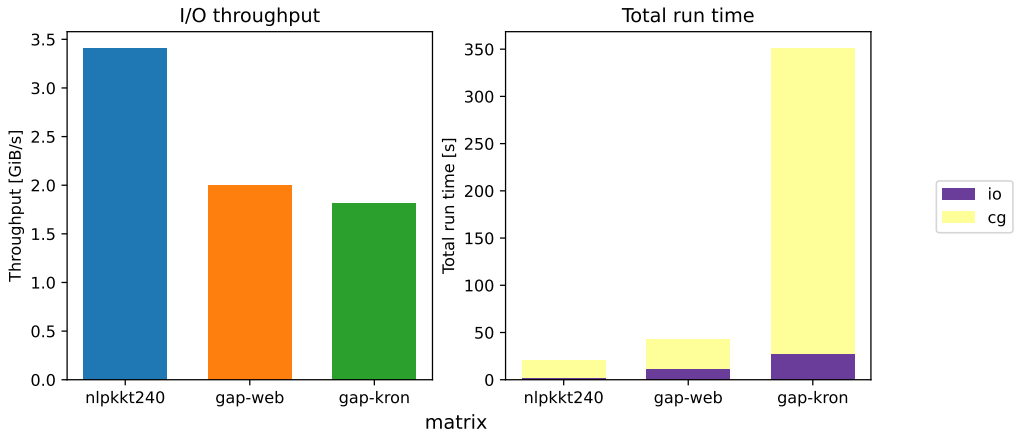

Figure 5.3: Time spent doing IO vs conjugate gradient (128 CPUs used)

5.3 Distributed conjugate gradient implementation

First, three MPI layouts were benchmarked, each totaling 128CPUs. A layout in this text will mean a triplet of **node count** N, **number of processes per node** P and **number of CPUs per process** C (usually abbreviated to `N-P-C`). The benchmarked layouts were `2-1-64`, `4-1-32` and `8-1-16`, keeping the same CPU count as the fastest single process implementation so that the results can be compared directly.

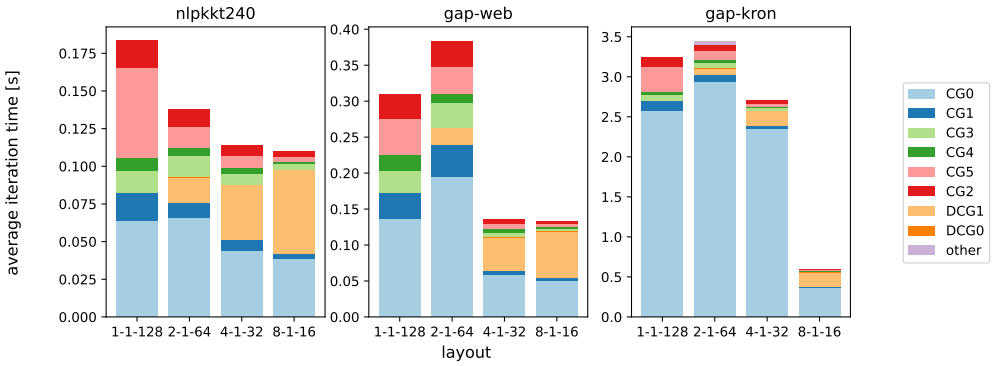

Figure 5.4: Average iteration times of single and multi process implementations

5.3.1 IO

RCI cluster provides BeeGFS parallel filesystem and by utilizing the algorithm described in Section 4.1, the IO throughput can be improved considerably. Using best-performing `8-1-16` layout results, throughput has improved by a factor of 2.5, 5.6, and 6.9 for *nlpkkt240*, *gap-web* and *gap-kron* respectively.

5.4 PETSc

To compare the distributed sparse matrix-vector multiplication (D-CSR5) implemented by `dim` to a state of the art BLAS library, an implementation of conjugate gradient solver using `PETSc` (v3.15.1) introduced in Section 1.4 was created.

`PETSc` supports multiple storage formats for sparse matrices such as compressed sparse row, block compressed sparse row, etc. The default for distributed sparse matrices, which is compressed sparse row (PETSc `MatType` `MATMPIAIJ` [12]) was used for benchmarking.

Furthermore, since `PETSc` is a general-purpose library, some tradeoffs had to be made in the benchmarking code. Even though `PETSc` has its own conjugate gradient implementation (`KSPCG`), the goal of the benchmark is to compare

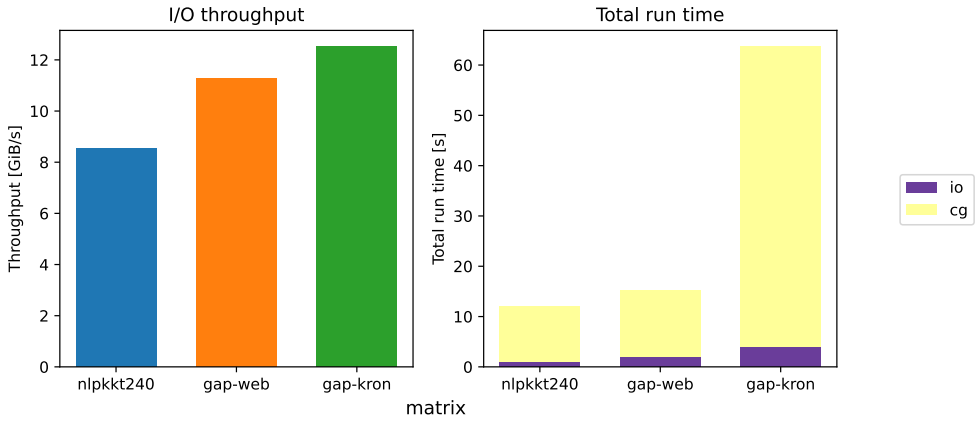

Figure 5.5: Time spent doing IO vs CG (8-1-16 layout)

the performance of distributed sparse matrix-vector multiplication, not the conjugate gradient implementation as a whole. To improve the granularity of benchmarked sections, an alternative implementation was produced to enable the timing of separate steps of the iteration instead of just timing the whole iteration step.

As the benchmark is using CSR as a storage format, each process of `PETSc` solver loads a sub-section of the input matrix' rows of the same size (except possibly the last process. Non-zero elements in the matrices used for benchmarking are distributed fairly evenly across these sections[12], so this approach yields good load distribution across all processes.

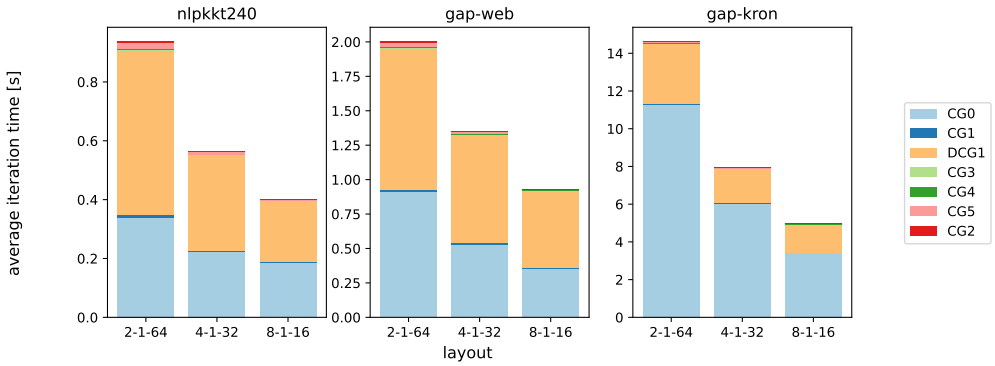

Figure 5.6: PETSc implementation benchmark results

[12]Heatmaps can be generated by running `dim_cli generate_heatmap <input_matrix>`.

The benchmark results show that a significant portion of time each iteration is spent in the `DCG1:distributing s` step. This issue partly stems from a design decision made in PETSc. The distributed vectors (such as `s`) have local size (size of vector part physically present in process) and global size (size of the whole vector). For vector-vector operations such as `axpy` or dot product, PETSc requires the participating vectors to have the same local size. Since each process may require any element from `s`[13] it must have whole `s` in its memory. To obtain a sub-vector of `s`, suitable for distributed `sAs` a process must call `VecGetSubVector` and then restore said sub-vector in `s` by calling `VecRestoreSubVector`.

However, `VecGetSubVector` may allocate new memory to store the sub-vector in, releasing said memory when `VecRestoreSubVector` is called, thus skewing the actual time needed for synchronization of `s` itself. For this reason, the `DCG1:distributing s` step will not be considered when comparing the two implementations.

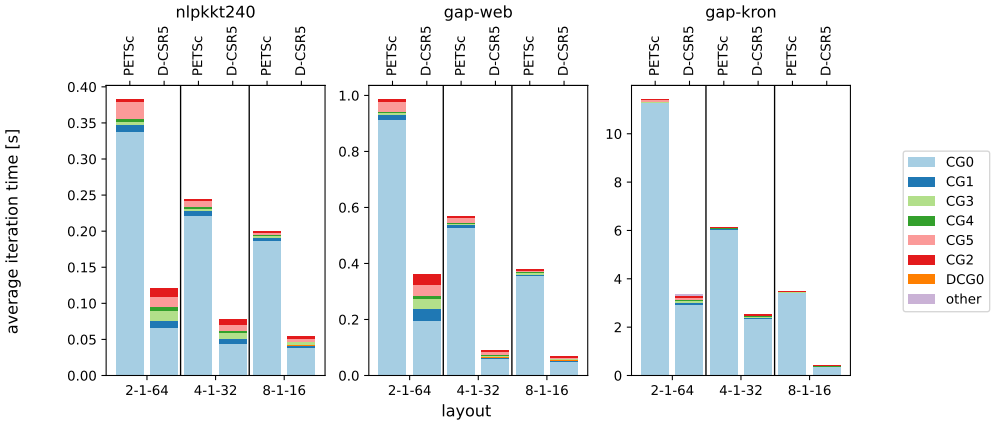

Figure 5.7: PETSc vs D-CSR5 benchmark results (no synchronization)

Even with the `distributing s` step removed, benchmark results (visualized in Figure 5.7) show that D-CSR5 implementation performs better on all the benchmarked configurations. The speedup of D-CSR5 based implementation compared to `PETSc` ranges from 2.4 and 2.57x for iteration and SpMV steps respectively up to 8.5 and 9.46x. Speedup over every layout and input matrix combination is shown in Figure 5.8.

[13] Because any element in a row of sparse matrix can be non-zero.

	nlpkkt240		**gap-web**		**gap-kron**	
layout	iteration	As	iteration	As	iteration	As
2-1-64	3.72	4.81	2.75	4.69	3.4	3.83
4-1-32	3.14	5.04	6.27	9.04	2.43	2.57
8-1-16	3.72	4.81	5.57	7.03	8.53	9.46

Figure 5.8: Speedup of iteration and SpMV step of iteration using D-CSR5 vs PETSc

Conclusion

This thesis aimed to research and evaluate the viability of distributing sparse matrix-vector multiplication among multiple computational nodes using MPI. To that end, coordinate (COO) and compressed sparse row (CSR) formats were reviewed, as well as Compressed Sparse Row 5 format introduced in [2].

On-disk storage formats for sparse matrices were reviewed. Namely Matrix Market Exchange Format and HDF5 backed format used by MATLAB v7.3. After reviewing existing formats, a storage scheme for CSR5 was devised using HDF5 as underlying format, nullifying the need for conversion from CSR to CSR5 each time a matrix is loaded.

The parallel SpMV algorithm outlined in [2] was implemented using C++20. Routines for loading and storing sparse matrices as well as converting between sparse matrix formats were also implemented. These were packaged into a library/toolkit named `dim`.

The building blocks from `dim` were then used to build distributed SpMV. Using MPI, SpMV was distributed among multiple processes, with the CSR5 SpMV re-implementation used to perform partial multiplication inside each process. Leveraging the MPI-IO capabilities of HDF5 library [1] to load only relevant parts of the matrix to minimize the time spent performing I/O operations. An algorithm for synchronizing and distributing the partial results was then outlined and implemented.

This distributed solution was then benchmarked, using conjugate gradient method implementation, against the single-process solution. These benchmarks clearly show that distributing the sparse matrix-vector multiplication for large sparse matrices can lead to significant speed-up even though the result needs to be synchronized and distributed amongst all processes each iteration.

To compare the performance of this solution against a battle-tested implementation conjugate gradient method was also implemented using `PETSc` toolkit and then benchmarked. Comparing the average per-iteration times suggests that the approach implemented in `dim` leads to non-negligible speed ups.

Future directions include researching algorithms other than conjugate

gradient in which SpMV is the most computationally intensive task and the viability of distributing it. The on-disk storage format introduced by this thesis may also benefit from more generalization, as it is currently quite specific to CPU-only implementation of SpMV and thus not suitable for GPU or CPU/GPU heterogeneous implementations.

Bibliography

[1] The HDF Group. Hierarchical Data Format, version 5. 1997-NNNN, https://www.hdfgroup.org/HDF5/.

[2] Liu, W.; Vinter, B. CSR5: An Efficient Storage Format for Cross-Platform Sparse Matrix-Vector Multiplication. 2015, 1503.05032.

[3] OpenMP Architecture Review Board. OpenMP Application Program Interface Version 5.0. Nov. 2018. Available from: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5.0.pdf

[4] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*. June 2021. Available from: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf

[5] Balay, S.; Abhyankar, S.; et al. PETSc Web page. https://petsc.org/, 2021. Available from: https://petsc.org/

[6] Boisvert, R.; Pozo, R.; et al. The Matrix Market Exchange Formats: Initial Design. 03 1997.

[7] Anton, H.; Rorres, C. *Elementary Linear Algebra: Applications Version*. Wiley, eleventh edition, 2014, ISBN 9781118434413 1118434412 9781118474228 1118474228.

[8] Wu, T.; Yan, D.; et al. An efficient sparse-dense matrix multiplication on a multicore system. 10 2017, pp. 1880–1883, doi:10.1109/ICCT.2017.8359956.

[9] Saad, Y. *Iterative Methods for Sparse Linear Systems*. Other Titles in Applied Mathematics, SIAM, second edition, 2003, ISBN 978-0-89871-534-7, doi:10.1137/1.9780898718003.

[10] Available from: https://math.nist.gov/MatrixMarket/mmio-c.html

[11] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 2.0*. June 1997. Available from: https://www.mpi-forum.org/docs/mpi-2.0/mpi20-report.pdf

[12] Balay, S.; Abhyankar, S.; et al. PETSc/TAO Users Manual. Technical report ANL-21/39 - Revision 3.16, Argonne National Laboratory, 2021.

[13] D'Azevedo, E.; Eijkhout, V.; et al. Lapack Working Note 56 Conjugate Gradient Algorithms with Reduced Synchronization Overhead on Distributed Memory Multiprocessors. 01 2000.

Appendix A

Conjugate gradient method

The conjugate gradient algorithm (abbr. CG) is one of the best known iterative methods which can be used to solve large symmetric positive definite linear systems [9]. The algorithm in its iterative form is outlined in Alogithm 12.

Algorithm 12 Iterative conjugate gradient

```
function CONJUGATE_GRADIENT(A, b, x)
    r ← b - Ax
    s ← r
    r_r ← r · r
    for i = 0; i < max_iters; ++i do
        As ← A · s
        alpha ← r_r / s'·As
        x ← x + alpha · s
        r ← r − alpha · As
        r_r_new ← r' · r
        if sqrt(r_r_new) < threshold then
            break
        end if
        beta ← r_r_new / r_r
        r_r ← r_r_new
        s ← r + beta · s
    end for
end function
```

A.1 Distributed conjugate gradient implementation

Each process first obtains a full copy of `s`. Then the steps of the algorithm above are performed as follows, with each step being labeled as `CG<num>` for the

steps of the actual algorithm and `DCG<num>` for the steps only the distributed version performs.

Step `CG0`: `As` $\leftarrow$ `A` $\cdot$ `s`. Each process can perform SpMV on its local part of `A`. This SpMV will result, in each process having part of `As`, with some overlapping output elements having only part of its value which needs to be synchronized.

Step `DCG0.1`: edge sync. After SpMV is done, the synchronization for overlapping edge elements described in Section 4.2 can begin.

Step `CG1`: `alpha` $\leftarrow \frac{\texttt{r\_r}}{\texttt{s}\cdot\texttt{As}}$. Since each process only has part of `As` and full `s`, it can do its part of the `s` $\cdot$ `As`. Dot product of two vectors is computed as $x \cdot y = \sum x_i \cdot y_i$, so `s` $\cdot$ `As` can be rewritten as $a \cdot As = \sum_{p=0}^{np} \sum_{i=os_p}^{oe_p} s_i \cdot As_{i-os_p}$ where np is number of processes, and os_p and oe_p denote start and end of output for process p respectively.The outer sum can be computed using `MPI_Allreduce`, summing the results of inner sum for each process.

For the elements that need to be synchronized (multiple processes are computing their sub-values), the resulting element is a sum of all of the sub-results thus $s \cdot As = \sum s_i \cdot As_i$ term for i which is computed by multiple processes can be rewritten as $s_i \cdot (As_{i_{p1}} + ... + As_{i_{pn}})$. As multiplication in R is distributive, this can be rewritten as $s_i \cdot As_{i_{p1}} + ... + s_i \cdot As_{i_{pn}}$ so there is no need to synchronize As_i as result will be synchronized when partial results for each process are summed.

Step `CG2`: `x` $\leftarrow$ `x` + `alpha` $\cdot$ `s` each process only has the part of `x`, coinciding with the output ranges into `As` it owns. Thus thus each process only modifies said part of `x`.

Step `CG3`: `r` $\leftarrow$ `r` $-$ `alpha` $\cdot$ `As` Similarly to previous step, each process only modifies the part of residual vector `r`. Synchronization of overlapping output elements for `As` must be done before this step, as process which owns the element uses it to modify `r` in this step.

Step `CG4`: `r_r_new` $\leftarrow$ `r'` $\cdot$ `r` Same as in alpha computation, the dot product can be divided into sub sums, which can then be summed across all processes, using `MPI_Allreduce`.

Step `DCG0.1`: edge sync. The second part of `DCG0`, which calls `MPI_Waitall` to wait until the sync is over as the owning processes need to have the correct values in `s`.

Step `CG5`: `s` $\leftarrow$ `r` + `beta` $\cdot$ `s` each process computes only part of new `s`, corresponding to its output range in `As`.

Step `DCG0.1`: distributing `s`. The resulting `s` is then distributed to all processes using the same algorithm as described in Section 4.2.2 using `MPI_Allgatherv`. This distribution step is the most intensive in the context of MPI communication.

Note that this algorithm may be further optimized as described in [13] but since it is only used as a test-bed for benchmarking implementation in `dim`, these are not implemented.

Appendix B

Building dim

This chapter describes the process of obtaining dependencies, configuring, and building `dim` as well as optional benchmarks.

B.1 Dependencies

`dim` uses `cmake` as its meta build-system. Only recent versions of CMake (3.20-3.22) were tested.

The code uses C++20 features and as such, needs a C++20 capable compiler. Tested compilers are:

- GCC 10.3.0 (available on RCI)
- GCC 11.1.0
- Clang 12.1.0 (available on RCI)
- Clang 13.0.1

`dim` is also built by GitHub Actions CI by clang 13 as well as linted by clang-tidy on each commit.

`conan` is used for package management. `conan` is implemented in Python and can be obtained from `pip`. The project will invoke `conan` automatically when `cmake` configure step is running, so it should be transparent to the user. If the code is built locally (meaning not on RCI cluster), some packages hosted on a private Artifactory instance of the author are needed. The following commands have to be executed to add the remote:

```
# the remote requires revisions to
# be enabled in local conan installation
conan config set general.revisions_enabled=1
# this hosts the PETSc package recipe as well
```

```
# as some prebuilt libraries.
conan remote add rurabori-conan \
    https://rurabori.jfrog.io/artifactory/api/conan/rurabori-conan
cmake . -B build # cmake arguments as usual ...
```

When building on the RCI cluster, it may be beneficial to use the provided libraries optimized for the nodes they will be running on. To enable this, pass `-Dsystem_scientific_libs="ON"` to `cmake` when configuring. There is a script at `scripts/rci/slurm_cmake` which will automatically load modules needed for compilation and offload the configuration/compilation step to a computational node.

B.2 Building the binaries

The project provides only two configuration options immediately relevant to this thesis. The first one is `enable_petsc_benchmark` which is disabled by default. If it is enabled, the PETSc implementation of the distributed conjugate gradient will be built, and the project will require PETSc as a dependency. The second option is `system_scientific_libs` which was explained in B.1. Both of these options can be enabled/disabled by passing `-D<option>="ON/OFF"` to `cmake` during configuration phase.

A typical RCI configuration might look like this:

```
cd <project_directory>
cmake . -B build \
        -Dsystem_scientific_libs="ON" \
        -Denable_petsc_benchmark="ON" \
        -DENABLE_TESTING="OFF" \
        -G "Unix Makefiles" \
        -DCMAKE_BUILD_TYPE=Release
cmake --build build
```

The resulting binaries will be in the build directory, in subfolders matching the structure of the repository i.e. `dim_cli` which is defined in folder `apps/dim_cli` will be found in `build/apps/dim_cli` folder. Alternatively the project can be installed as a SLURM module after it is built by running:

```
cmake --install build \
      --prefix <install_dir>/dim
MODULEPATH="$MODULEPATH:<install_dir>"
# from here on, dim should behave as any other module
ml dim
dim_cli --version
```

B.3 Available executables

`dim` build produces multiple executables. Benchmarks which were used to produce data for this thesis, as well as `dim_cli`, a command-line utility with multiple subcommands for downloading/converting and working with sparse matrices in general. Only the CLI will be explained as the benchmarks map well to graphs in this thesis, so they should be pretty self-explanatory.

B.3.1 dim_cli

`dim_cli` is a command-line utility which provides four subcommands [14]:

- `store_matrix`
- `compare_results`
- `download`
- `generate_heatmap`

`store_matrix` subcommand takes a path to matrix in MMEF or Matlab compatible HDF5 format, and stores the matrix in CSR or CSR5 format (provided to the `-f` flag) into a HDF5 file. The command also requires a configuration file, which describes properties for the HDF5 groups in which the data will be stored. A sample config is stored in `resources/config.yaml` and installed into `share` directory when `dim` is installed as a SLURM module.

```
# convert MMEF to CSR5.
dim_cli store_matrix matrix.mtx \
                     -f csr5 \
                     -c resources/config.yaml
# will produce matrix.csr5.h5
```

`compare_results` takes a path to HDF5 file (or two files) with result vectors and compares them. This command is mainly useful for validating that results are correct and stable across implementations.

```
# compare two result vectors.
dim_cli compare_results res.h5 -l "/Y1" -r "/Y2"
```

`download` subcommand retrieves a matrix from an URL provided as an argument and unpacks it while downloading, saving some time as the used matrices are fairly large.

[14] dim_cli supports `-h`/`--help` for the top-level executable as well as for each subcommand for more detailed argument listing.

```
# downlaod and unpack matrix from <URL> into
# resources/matrices directory.
dim_cli download <URL> -d resources/matrices
```

`generate_heatmap` expects a path to matrix in CSR5 format introduced in Section 3.5. It will produce a black and white PNG image, with the brightness reflecting the number of non-zero elements of the segment of the matrix the pixel represents.

```
dim_cli generate_heatmap mat.csr5.h5
# mat.png will be produced
```

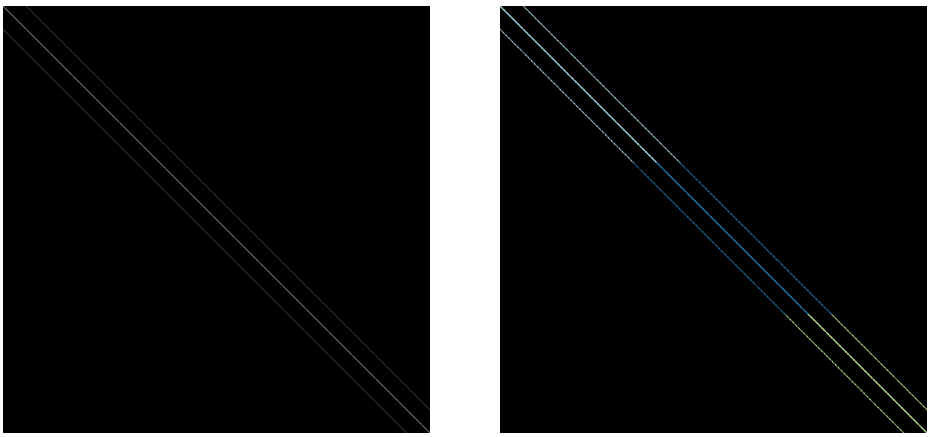

Figure B.1: Zoomed-in example of a generated heat/distribution maps.

However, this command will behave a bit differently when process count `-p` is specified. It will not reflect the density structure as precisely as the black and white output, coloring the pixels with a different color for each process instead. Resolution can also be specified by providing `-r <width> <height>` if a finer structure needs to be captured (or the matrix is large).

Appendix C

Acronyms

COO Coordinate Format

CSR Compressed Sparse Row Format

CSR5 Compressed Sparse Row 5 Format

CLI Command Line Interface

n_{nz} Number of non-zero elements in a sparse matrix

HDF5 Hierarchical Data Format v5

dim The library implemented during result for this thesis

PETSc Portable, Extensible Toolkit for Scientific computation

CG conjugate gradient method

SpMV Sparse Matrix-Vector multiplication

SSE Streaming SIMD Extensions

AVX Advanced Vector Extensions

Appendix D

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `src` ..... the directory of source code
  - `include` ..... Header files
  - `lib` ..... Libraries
  - `apps` ..... Applications
  - `benchmarks` ..... Benchmarks
  - `resources` ..... Auxiliary resources
    - `benchmark_data` ..... Data from benchmarks ran on RCI cluster
    - `config.yaml` ..... Default config for `dim_cli`
  - `docs/paper` ..... the thesis source in LaTeX format
  - `tests` ..... Unit tests
- `thesis.pdf` ..... the thesis text in PDF format