2016

Performance analysis and acceleration of nuclear physics application on high-performance computing platforms using GPGPUs and topology-aware mapping techniques

Dossay Oryspayev
Iowa State University

Performance analysis and acceleration of nuclear physics application on high-performance computing platforms using GPGPUs and topology-aware mapping techniques

by

Dossay Oryspayev

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY

Major: Computer Engineering

Iowa State University
Ames, Iowa
2016

TABLE OF CONTENTS

ACKNOWLEDGEMENTS

I would like to take this opportunity to express my sincere gratitude to those who helped me with various aspects of research and writing of this thesis. First and foremost, Dr. Pieter Maris for his guidance, patience, and support throughout this research and the writing of this thesis. His expert advice, careful attention to details, and feedback and suggestions on countless drafts of my thesis has been invaluable to me. I am deeply grateful to Dr. Joseph Zambreno for his guidance, support, encouraging conversations, and for generously sharing his time and expertise. I am thankful to Dr. James Vary for his support and allowing me to conduct research using his allocation times on supercomputers. I would also like to thank Dr. Diane Rover and Dr. Yong Guan for insightful comments, questions, and suggestions which improved my thesis. I would like to thank Dr. Masha Sosonkina for her great support and guidance during my initial years of graduate studies. I am also grateful to Dr. Glenn Luecke and Dr. Metin Aktulga for their support at various stages of my research. Finally, and most importantly, I would like to thank my family for their love and endless support.

A portion of the computational resources were provided by the National Energy Research Scientific Computing Center (NERSC), a DOE Office of Science User Facility supported by the US DOE Office of Science under Contract No. DE-AC02-05CH11231, and by the Innovative and Novel Computational Impact on Theory and Experiment (INCITE) program. This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725. This research also used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under contract DE-AC02-06CH11357.

ABSTRACT

The number of nodes on current generation of high performance computing (HPC) platforms increases with a steady rate, and nodes of these computing platforms support multiple/many core hardware designs. As the number of cores per node increase, either CPU or accelerator based, we need to make use of all those cores. Thus, one has to use the accelerators as much as possible inside scientific applications. Furthermore, with the increase of the number of nodes, the communication time between nodes is likely to increase, which necessitates application specific network topology-aware mapping techniques for efficient utilization of these platforms. In addition, one also needs to construct network models in order to study the benefits of specific network mapping. The specific topology-aware mapping techniques will help to distribute the computational tasks so that the communication patterns make optimal use of the underlying network hardware. This research will mainly focus on the Many Fermion Dynamics nuclear (MFDn) application developed at Iowa State University, a computational tool for low-energy nuclear physics, which utilizes the so-called Lanczos algorithm (LA), an algorithm for diagonalization of sparse matrices that is widely used in the scientific parallel computing domain. We present techniques applied to this application which enhance its performance with the utilization of general purpose graphics processing units (GPGPUs). Additionally, we compare the performance of the sparse matrix vector multiplication (SpMVM), the main computationally intensive kernel in the LA, with other efficient approaches presented in the literature. We compare results for the total HPC platforms' resources needed for different SpMVM implementations, present and analyze the implementation of communication and computation overlapping method, and extend a model for the analysis of network topology presented in the literature. Finally, we present network topology-aware mapping techniques, focused at the LA stage, for IBM Blue Gene/Q (BG/Q) supercomputers, which enhance the performance as compared to the default mapping, and validate the results of our test using the network model.

CHAPTER 1. INTRODUCTION

In this section we give a brief introduction to the research work of this thesis and summarize the contributions of the works performed. Specifically, section 1.1 presents the motivation of this research, section 1.2 presents the contributions of the research work done, and section 1.3 lays out the organization of the thesis.

1.1 Motivation

The advent of supercomputers, HPC platforms, has led to many advances in science by making it possible to tackle large scientific problems. For example, complex models for severe weather forecasting, advanced models for the spread of infectious diseases, and molecular dynamics modeling rely on supercomputers. In recent years we have witnessed steady growth both in the number of cores per node (either through accelerators or in many-core nodes) and in the number of nodes available on supercomputers, which gives a tremendous computing power to them. To this date, according to the latest list of TOP 500 supercomputers [3] in the world, November, 2015, the #1 supercomputer in the world, Tianhe-2 (Milkyway-2) features $16,000$ compute nodes, and a total of $3,120,000$ cores.

With so many cores available, when tackling a scientific application problems ported to these supercomputers, issues of data locality and data placement within a node, as well as communication between nodes needs to be carefully addressed. Detailed knowledge about the memory hierarchy within the node and the interconnect between the nodes becomes crucial for applications. So, one should consider both application and hardware specific information to utilize the full power of the resources available at hand.

One has to use hybrid programming paradigms on these supercomputers because of distributed and shared memory infrastructures. Multi-level parallelism on these supercomputers present the possibility of overlapping communication with computation. Moreover, shared memory designs, the arrival of accelerators such as GPGPUs and introduction of Intel[1]'s Many Integrated Core (MIC) architecture added more complexity to the programming and efficient utilization of current supercomputers. These new technological advances require detailed knowledge of both hardware and software.

In order to reduce communication overhead, one has to first understand the communication patterns in the application, and then map these communication patterns efficiently onto the physical network connecting the nodes of a specific supercomputer. With the help of a network model one can quantitatively analyze the performance of different mappings, and once validated, use the network model to choose suitable mappings for future calculations. The main goal of this research work is to address these issues and present solutions to improve the overall performance of scientific application–MFDn.

MFDn [66, 1] is a scientific application for performing large-scale *ab initio* calculations of atomic nuclei. It runs on several HPC platforms worldwide; within the U.S. more than 100 million CPU hours were used by this application in 2015 on the supercomputers at National Energy Research Scientific Computing Center[2] (NERSC), Oak Ridge National Laboratory[3] (ORNL), and Argonne National Laboraty[4] (ANL). MFDn has been developed at Iowa State University by Vary and collaborators for more than two decades [66]. Significant improvements in the performance of MFDn have been made under the U.S. Department of Energy Scientific Discovery through Advanced Computing (SciDAC) [64] project, Universal Nuclear Energy Density Functional (UNEDF, 2007-20012) [5], and ongoing under the SciDAC Nuclear Computational Low-Energy Initiative (NUCLEI) [2] project. The "SciDAC program was created to bring together many of the nation's top researchers to develop new computational methods for tackling some of the most challenging scientific problems" [64]. These efforts resulted in a

[1] http://www.intel.com
[2] https://www.nersc.gov
[3] https://www.ornl.gov
[4] https://www.anl.gov

hybrid Message Passing Interface (MPI) [44]/Open Multi Processing (OpenMP) [49] Fortran 90 code that uses MPI for communication among different nodes and OpenMP for multi-threaded execution within a node, which scales to thousands of nodes.

In MFDn the quantum many-body Schrödinger equation for atomic nuclei is represented as a sparse matrix problem. It uses LA [40, 59] to obtain the lowest eigenvalues and eigenvectors of this sparse symmetric many-body Hamiltonian matrix. Each Lanczos iteration consists of a SpMVM followed by an orthogonalization. By far, the most computationally intensive parts of the calculation are: (1) constructing the large sparse matrix and (2) obtaining the lowest eigenvalues and eigenvectors of this matrix. The dimension of this matrix can reach several billions, but it is a very sparse matrix. The symmetry of the matrix allows us to store only the half of the matrix (including the diagonal matrix elements). Optimal use of the limited amount of computing resources require efficient construction of the nuclear Hamiltonian matrix and efficient SpMVM, which is the computationally intensive kernel of the LA. A complicating factor is that only half the matrix is stored due to memory limitations; therefore, one also has to perform an efficient transpose SpMVM, SpMVM$^{\mathsf{T}}$, with the same data structure.

1.2 Contributions of the research work

The goal of this research is to be able to analyze the specific topology-aware mapping techniques and enhance the execution of MFDn application in the presence of new technological and network infrastructural advancements in the HPC domain. One of the main metrics of importance to us is the "aggregate CPU core hours" (or "Raw Machine hours"), which is an important indicator of supercomputer usage and measured by multiplying the total number of cores used with the total time of execution. In this thesis, we first show the porting of a specific part of the MFDn application onto GPGPUs, and present the speedups obtained. Next, we focus on the performance of the SpMVM used during the Lanczos iterations, for which we compare different data structures for storing the matrix and different implementations for the SpMVM. We also extend the network model to analyze the communication during the SpMVM. Finally, we propose topology-aware mappings which significantly improve the performance of

MFDn on IBM[5] BG/Q supercomputers. We then further extend the model to evaluate the proposed mappings and verify their efficiency compared to the default mapping on IBM BG/Q. The main contributions of this thesis can be summarized as follows:

1. **Introduction of GPGPU into MFDn:** In the work presented in [52], we describe initial steps of leveraging accelerators, such as GPGPUs, in *ab initio* nuclear physics calculations. Specifically, we outlined and implemented the necessary steps to make MFDn utilize GPGPUs during its matrix construction stage for runs with 2- and 3-body forces. (For such runs the matrix construction typically takes about 20% to 40% of the total runtime.) The experiments are presented comparing the multithreaded CPU-based version of this stage with the newly designed GPGPU-based version. Four- to ten-fold speedups were observed for the code executing on GPGPUs. Results are presented in Chapter 3 [6].

2. **Analysis of different SpMVM techniques:** In this work [50], we present the analysis of different distributed SpMVM techniques. Symmetric SpMVM, used in MFDn, is an important kernel that frequently arises in HPC scientific applications. Since SpMVM is a kernel with low arithmetic intensity, several approaches have been proposed in the literature to improve its scalability and efficiency in large scale computations. In this work, our target systems are high-end multi-core architectures and we use a hybrid programming model, MPI+OpenMP, for parallelism. We analyze the performance of an implementation of the distributed symmetric SpMVM, originally proposed for large sparse symmetric matrices arising in *ab initio* nuclear structure calculations, used in MFDn application.

   First, we study important features of this implementation and compare it with other SpMVM implementations that do not exploit the symmetry of the matrices. Our main comparison criteria is the "aggregate CPU core hours" metric. Second, since the communication overhead becomes crucial for large-scale runs on supercomputers, we analyze the mapping of logical process ranks onto the physical processors using a simplified network

[5] `http://www.ibm.com`

[6] Modified from a paper published in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International* [52].

load model. Third, we orchestrate an overlapping of communication and computation using the hybrid MPI+OpenMP paradigm. The SpMVM implementation that takes the matrix symmetry into account and hides communications yields the best value for the "aggregate CPU core hours" metric while incurring a reduced communication overhead in data movement. Results are presented in Chapter 4 [7].

3. **Performance improvement of LA on an IBM BG/Q supercomputer through topology-aware mappings:** We propose new mappings which utilize the network interconnect of Mira, an IBM BG/Q, supercomputer that are more efficient than the default mapping. The current mapping used in MFDn was based on the Cray[8]'s scheduler used on Hopper, Cray XE6, supercomputer, which ensured that physically nearby processing units are ranked consecutively [9]. This mapping seems to be effective for platforms with schedulers that have sparse allocations, with non-contiguous (blocks of) nodes. In such situations we do not know ahead of time where our MPI ranks will be placed on the network interconnect, except for the case when we use the entire machine. However, the architectural design of Mira at ANL differs in many factors. An essential difference for optimizing network mappings is that on Mira our jobs are allocated in contiguous blocks, which means that we know the coordinates of each of the MPI ranks before our run starts. Furthermore, the block is reserved exclusively for a single job or batch script (with minor negligible exceptions), and routing is zone based.

   We concentrate on LA stage of the MFDn as it is the most computationally intensive part. We present the detailed empirical results in Chapter 5. We run our experiments up to full machine, which is 48 racks, total of $786,432$ cores. The topological mapping techniques we propose take specifics and details of Mira supercomputer into account, and improve overall performance of MFDn significantly compared to the default mapping. We also explore the use of different MPI call routines, and find combinations that perform better than the original setup. Additionally, we have developed and validated a network

[7]Modified from a paper published in *The Journal of Concurrency and Computation: Practice and Experience, 2015* [50].

[8]http://www.cray.com

model for analyzing these mappings for the different MPI routines. A paper describing this work is in preparation [51].

1.3 Organization of the thesis

Chapter 1 contains a brief introduction to the application studied in this thesis and a summary of main contributions. In chapter 2 we give more detailed background on the MFDn application, the algorithms, and data structures used in it. Chapters 3–5 contain the main contributions of this work. Specifically, chapter 3 gives details on the initial steps taken to introduce the GPGPUs into MFDn application. In chapter 4, we present our work on performance analysis of distributed SpMVM techniques. In chapter 5, we present the results of mapping techniques we propose for IBM BG/Q supercomputer, and extended network model that validates the results. Finally, chapter 6 contains a summary of the results and conclusions.

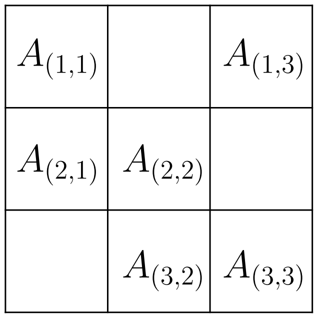
CHAPTER 2. BACKGROUND

In this chapter we briefly describe the main terminology and components used in the MFDn application. Specifically, in section 2.1 we give a general overview of the MFDn application, in section 2.2 we provide detailed information on LA, in section 2.3 we describe several commonly used compressed storage formats for storing sparse matrices along with the most recent storage format that is used in MFDn, and finally in section 2.4 we give more detailed information on the execution of LA process in MFDn application.

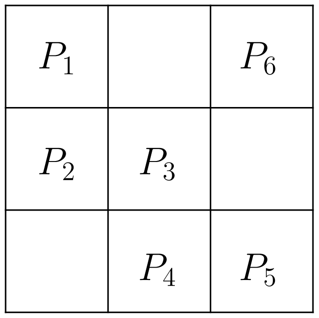
2.1 Brief overview of MFDn

The Configuration Interaction (CI) approach is a computational approach in quantum mechanics for solving the many-body Schrödinger equation as a matrix problem. The lowest eigenvalues of the matrix correspond to the energies of the low-lying states, and the eigenvectors correspond to wave functions which can be used to calculate additional observables. The basis is constructed from antisymmetrized Slater Determinants of single-particle states. In the limit of a complete (but infinite-dimensional) basis the calculations become exact; in practice the basis is truncated to a finite dimension. In general, the larger the basis, the higher the accuracy of a calculation for a given system. Thus, the biggest challenge in CI calculations is that it leads to very large (but very sparse) Hamiltonian matrices, of which one needs to calculate the lowest eigenvalues and eigenvectors. The dimension grows rapidly with the number of particles, and is often of the order of 10^9 or more [65, 42] and for codes that keep the nonzero matrix elements in-core, the number of nonzero matrix elements is only limited by the aggregate memory of the available HPC platforms.

MFDn [62, 8, 42, 41, 7] is a state-of-art software package that performs No-Core CI calculations for atomic nuclei, that is, for self-bound systems consisting of protons and neutrons. It obtains the lowest eigenvalues and eigenvectors of the sparse symmetric many-body Hamiltonian matrix A of size $m \times m$. It is a hybrid MPI/OpenMP parallel code, developed mainly at Iowa State University, and written in Fortran; it has been under continuous development for almost 25 years [66]. The MFDn implementation considered in this thesis, mainly, is the in-core implementation, i.e., all nonzero matrix elements are stored in-core memory. Although such implementation is limited by the memory available on the processors, it is faster compared to the out-of-core counterpart that uses the disk I/O for the matrix construction [67].



(a) Data distribution.



(b) Process mapping.

Figure 2.1: Distributed symmetric SpMVM data distribution and process mapping. Partitioning of the large symmetric matrix, A, among $n_d(n_d + 1)/2 = 6$ MPI processes with $n_d = 3$ diagonals.

Since the matrix is symmetric, and available memory is a bottleneck, only half of the nonzero matrix elements are stored, distributed over all MPI processes. The matrix is partitioned into n_d by n_d approximately square submatrices as shown in Figure 2.1a using $n_d(n_d+1)/2$ MPI ranks as shown in Figure 2.1b. The basis is distributed over n_d "diagonal MPI ranks" (referred to as "diagonals") in such a way that the resulting off-diagonal submatrices all contain approximately the same number of nonzero matrix elements. On the diagonals we only store half of a diagonal submatrix, which tends to have less nonzero matrix elements than the off-diagonals. However, for extremely large and extremely sparse matrices the diagonals may actually have more nonzero matrix elements than the off-diagonals, potentially leading to significant load-imbalance. One way to deal with this is by using one (or more) additional MPI process for each diagonal MPI rank, referred to as "extras" in MFDn. Alternatively, one can treat the diagonal part of the matrix separately, distributed over all MPI processes.

Once the basis is distributed, each MPI process locates and evaluates nonzero matrix elements in its submatrix. The distribution of the matrix over the MPI ranks is very well load-balanced: all off-diagonal MPI ranks have approximately the same workload and memory usage, but most of the (highly nontrivial) structure of the matrix is lost. For practical purposes, one can consider the submatrices to be unstructured sparse matrices. (Note that the submatrices themselves are not symmetric, with the exception of the diagonal submatrices.)

MFDn has several major stages:

- The initialization stage consists of reading and processing the input data, followed by the construction and distribution of the many-body basis. Most of this initialization, including the basis construction, is performed on the diagonal MPI processes only, and the information generated is broadcasted to the off-diagonal processes by these diagonal MPI processes.

- The construction of the matrix A itself, which includes identifying the location of the nonzero matrix elements and evaluating their values. Initially these nonzero matrix elements were stored in compressed sparse column (CSC) (also known as CCS), but in the current version of MFDn, these nonzero matrix elements are stored in the compressed sparse blocks coordinate (CSB_COO) [7] format. Determining each nonzero matrix element location and evaluating their values takes up a significant amount of time, up to about 40% of the entire calculation for representative cases [41].

- In the third stage, MFDn uses an iterative LA to obtain the low-lying eigenvalues and eigenvectors of the matrix A. Each iteration of the LA is composed of SpMVM followed by full reorthogonalization. Since only half of the symmetric matrix is stored in-core, one has to do both an SpMVM and a SpMVM$^{\mathsf{T}}$ with the same data structure. This is the most computationally intensive stage, because typically several hundred iterations are required for convergence, especially for the eigenvalues beyond the ground state. Efficient hybrid MPI/OpenMP multicore implementations have been proposed and implemented [8, 7];

the performance with respect to the non-uniform memory architecture (NUMA) inside a NUMA machine is studied in [61].

- The final stage in MFDn is composed of using eigenvectors to obtain a set of (user-defined) observables, which, once converged, can be compared with experimental nuclear structure data, and used for new predictions in nuclear physics.

2.2 Lanczos Algorithm (LA)

The LA [40, 59] is an iterative method useful for finding the eigenvalues and eigenvectors of a symmetric matrix. Let A be a large, unstructured, sparse, and symmetric n by n matrix. For such a matrix there is no easy way of finding eigenvalues. Using LA we can tridiagonalize it, i.e., find a tridiagonal matrix T such that $A = Q * T * Q^{\mathsf{T}}$, and then use various known efficient algorithms to find eigenvalues and eigenvectors of T. In case of MFDn, the DSTEVX subroutine provided by LAPACK library [12] is used. In exact arithmetic, the matrix T is similar to A, so eigenvalues of T are identical to eigenvalues of A. Further, if y is an eigenvector of T corresponding to eigenvalue λ, i.e., $T * y = \lambda * y$, then we get the following equality $A * (Q * y) = Q * (T * y) = Q * \lambda * y = \lambda * (Q * y)$. So $Q * y$ is the eigenvector of A corresponding to the same eigenvalue λ. Hence finding eigenvalues and eigenvectors of A will be easier, once tridiagonal matrix T is found. In floating point arithmetic, the eigenvalues of T will be approximate to egienvalues of A. There are various equivalent forms of the LA, MFDn uses the one similar to the classical algorithm presented in [40, 59] and shown in algorithm 1.

Algorithm 1: Classical Lanczos Algorithm

```
1: Set $||q_1|| = 1$ and $\beta_1 = 0$
2: for $i = 1, \dots, n$ do
3:     Compute the vector $A * q_i$
4:     Set $\alpha_i = q_i^\mathsf{T} * (A * q_i)$
5:     Set $z = A * q_i - \alpha_i * q_i - \beta_i * q_{i-1}$
6:     Set $\beta_{i+1} = ||z||$
7:     if $\beta_{i+1} == 0$ then
8:         BREAK
9:     else
10:        $q_{i+1} = z/\beta_{i+1}$
11:        CONTINUE
12:    end if
13: end for
```

$$T_k = \begin{bmatrix} \alpha_1 & \beta_2 & 0 & \cdots & 0 \\ \beta_2 & \alpha_2 & \beta_3 & & \vdots \\ 0 & \ddots & \ddots & \ddots & 0 \\ \vdots & & \beta_{k-1} & \alpha_{k-1} & \beta_k \\ 0 & \cdots & 0 & \beta_k & \alpha_k \end{bmatrix}$$

If the process stops at k-steps, the LA will give us a tridiagonal matrix T_k (as shown to the right of algorithm 1), and the orthonormal vectors $q_1, \dots, q_k$. The process will stop at no more than n steps, because we cannot have $n+1$ orthogonal vectors in n-dimensional vector space. Then we will have the equation $A * Q_k = Q_k * T_k$, where the matrix Q_k is built using vectors $q_1, \dots, q_k$ as columns. So the matrix Q_k is n by k and orthogonal, i.e., $Q^\mathsf{T} * Q = I_k$ and $Q * Q^\mathsf{T} = I_n$. Note that all of these hold in the exact arithmetic. However, in practice, as k gets larger, the matrix Q_k looses its orthogonality, since computations are done in finite floating precision arithmetic [23]. For this reason in MFDn during the LA stage full reorthogonalization at every iteration is done to preserve the orthogonality [23, 40]. The steps of the LA as implemented in MFDn are shown in Algorithm 2. Other variants of this algorithm include partial or selected reorthogonalization, which are currently under investigation for MFDn.

In Algorithm 2, the main loop executes until the algorithm is converged, thus the lines 10 and 11 were intentionally added to check the convergence of the algorithm in order to exit from do loop earlier. This extra checking step comes at almost no additional cost, but instead helps to avoid the unnecessary loops. (Note that in MFDn the convergence is checked every 10th iteration.)

Algorithm 2: Lanczos algorithm (LA) in MFDn.

```
1  procedure serial Lanczos ($A, \mathbf{x}_0$);
   Input  : Sparse symmetric matrix $A$ and initial vector $\mathbf{x}_0$;
   Output: Eigenvectors $Z$ and eigenvalues $D$ such that $\|A \times Z - Z \times D\|_F$ is small;
2  $\mathbf{x} \leftarrow \mathbf{x}_0/\|\mathbf{x}_0\|$;
3  $X \leftarrow (\mathbf{x})$;
4  while not converged do
5      $\mathbf{b} \leftarrow A \times \mathbf{x}$;
6      $\mathbf{b} \leftarrow \mathbf{b} - X \times X^\mathsf{T} \times \mathbf{b}$;
7      $\mathbf{x} \leftarrow \mathbf{b}/\|\mathbf{b}\|$;
8      $X \leftarrow (X, \mathbf{x})$;
9      $T \leftarrow X^\mathsf{T} \times A \times X$;
10     Diagonalize $T$ to obtain $(U, D)$;
11     Check convergence;
12 end
13 $Z \leftarrow X \times U$;
```

Depending on memory access patterns induced by the application, the SpMVM is typically the most expensive operation in LA. Many kernels used in this algorithm are shared by several other iterative methods in sparse linear algebra. For example, Hendrickson et al. [30] use the closely related Conjugate Gradient (CG) algorithm to evaluate the performance of their SpMVM implementation.

2.3 Storage formats for sparse matrices

There are a number of schemes proposed for compact storage of sparse matrices' nonzero elements. In this section we first briefly describe, in subsection 2.3.1, most common storage formats used, and also describe the currently used storage format in MFDn, in subsection 2.3.2.

2.3.1 Common Storage Formats

The Coordinate (COO) [59], Compressed Sparse Row (CSR) [15, 59], and the closely related CCS/CSC storage formats are among the most commonly used storage formats for storing nonzero matrix elements of sparse matrices. The CSR format allows a straight-forward SpMVM implementation in addition to providing a compact memory footprint, and nonzeros of a matrix row are stored consecutively as a list in memory (*val*).

Table 2.1: Data used by COO, CSR, and CCS/CSC formats.

$$M = \begin{bmatrix} 3 & 0 & 0 & 45 \\ 34.6 & 0 & 12 & 0 \\ 0 & 3.4 & 0 & 0 \\ 36 & 0 & 0 & 0 \end{bmatrix}$$

`val`	`rowind`	`colind`	`rowptr`	`colptr`
3	1	1	1	1
34.6	2	1	3	4
36	4	1	5	5
3.4	3	2	6	6
12	2	3	7	7
45	1	4		

One maintains an array of pointers (which are simply integer offsets) into the list of nonzeros in order to mark the beginning of each row (*rowptr*). An additional index array is used to keep the column indices of the nonzeros (*colind*). Nonzero values and column indices are stored in separate arrays of length equal to the number of nonzeros in the matrix, nnz. The length of the row pointers array is equal to one more than the matrix dimension, $n+1$. For single-precision floating-point matrices whose rows and columns can be addressed with 32-bit integers (i.e., $n \leq 2^{32}-1$), the storage cost for the CSR format is $8nnz + 4(n+1)$. Whereas CSR stores the values row by row, the CCS stores the them column by column and uses the *val*, *colptr*, and *rowind* arrays. Both CSR and CCS use array of pointers, but COO instead stores both row (*rowind*) and column (*colind*) indices of the values (*val*), with a storage cost of $12nnz$ using 32-bit integers for the indices, i.e., a memory footprint that is up to 50% larger than CSR or CSC/CCS. An example is given in Table 2.1 for a matrix with six nonzero elements. (Note that although the global matrix dimension in MFDn is often too large to be addressed by 32-bit integers, the local submatrices are always such that the rows and columns can easily be addressed by 32-bit integers.)

2.3.2 Compressed Sparse Block Format

There are a number of schemes proposed for compact storage of sparse matrices as described in section 2.3.1. In case of symmetric distributed memory SpMVM algorithm, as the one used by LA of MFDn, each process needs to perform the local SpMVM as well as the local SpMVM$^\mathsf{T}$ operations. One may reuse matrices stored in the CSR format for the SpMVM$^\mathsf{T}$ operation by reinterpreting row pointers and column indices as column pointers and row indices, respectively. Such an interpretation would correspond to the CCS/CSC representation in which one operates on columns rather than row sums to implement the SpMVM$^\mathsf{T}$ operation.

$$M = \left(\begin{array}{cc|cc} 3 & 0 & 0 & 45 \\ 34 & 0 & 12 & 0 \\ \hline 0 & 3.4 & 0 & 0 \\ 36 & 0 & 0 & 0 \end{array}\right)$$

$$\text{gval} = \{3,34,36,3.4,12,45\}$$

$$\text{grloc} = \{1,2,2,1,2,1\}$$

$$\text{gcloc} = \{1,1,1,2,1,2\}$$

Block #	lnnz	roffset	coffset	gptr
(1,1)	2	0	0	0
(2,1)	2	2	0	2
(1,2)	2	0	2	4
(2,2)	0	2	2	6

Figure 2.2: Example of CSB_COO format.

In a shared memory parallel environment, when performing SpMVM$^\mathsf{T}$ using the CSR or CCS/CSC storage, a possible race condition between threads has to be eliminated. The use of atomic updates or locks has serious negative effects on execution time. Another option is to keep a private copy of the output vector for each thread (see Figure 2.5a, which will be explained in section 2.4). This approach was implemented in the production version of MFDn prior to 2015, and works reasonably efficient with a small number of threads. However, it has a memory overhead of $O(nt)$, where t denotes the number of threads, which becomes problematic on many-core architectures; in addition, the reduction of the private output vectors over the number of threads becomes time consuming. Furthermore, keeping several private output vector copies may also adversely effect data reuse in cache and therefore result in worse performance compared to the SpMVM computation [50].

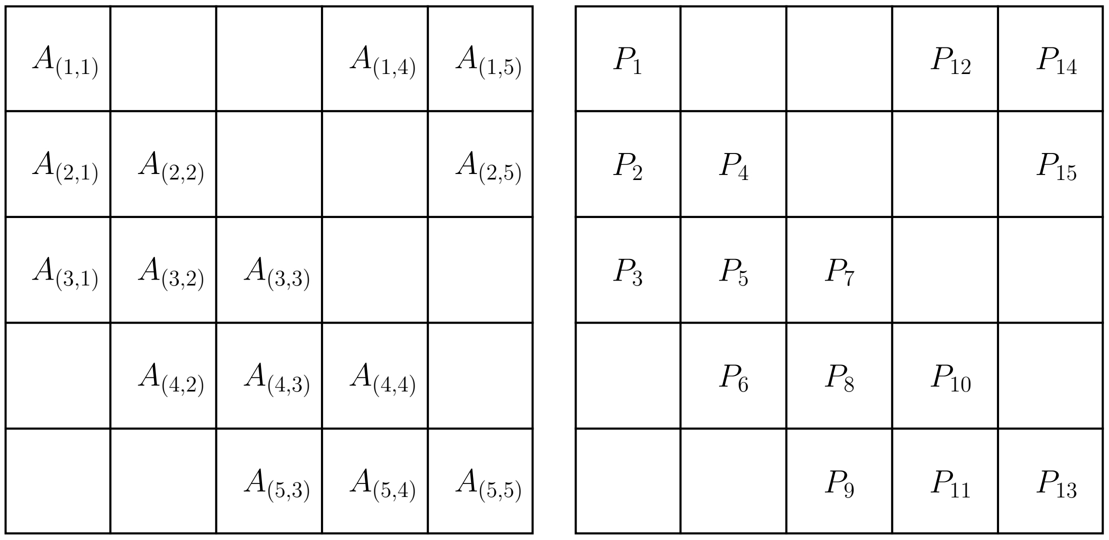
The CSB format [19] provides a solution to this problem. For a given block size parameter β, CSB nominally partitions an $n \times n$ matrix into $\beta \times \beta$ blocks. When β is on the order of

$\sqrt{n}$, we can address nonzeros within each block by using half the bits needed to index into the rows and columns of the full matrix (16 bits instead of 32 bits). For $\beta = \sqrt{n}$, the storage cost of CSB matches the storage cost of traditional formats such as CSR. Each $\beta \times \beta$ block is independently addressable through a 2D array of pointers.

In the works presented in chapters 4 and 5, the MFDn application uses a recently proposed variant of the CSB format, named CSB_COO [7]. In CSB_COO (see Figure 2.2), blocks of (sub)matrices are stored using the COO format. In this implementation the SpMVM operation is performed by processing the 2D pointer array in rows. Essentially, each thread is assigned a block row of dimensions $\beta \times n$ in the local sparse matrix vector multiply (the size of this block row is $\beta \times n$ because all of the blocks in that row are being processed by a single thread). β is chosen such that there are several more block rows than the number of threads t. To ensure load balancing among threads, the block rows are dynamically scheduled in an OpenMP parallel loop. Similarly, SpMVM$^{\mathsf{T}}$ is implemented by processing the 2D array of pointers in columns. In this case, the threads act on block columns of the matrix. Note that there are no race conditions between the outputs of different threads in SpMVM$^{\mathsf{T}}$. In our experiments (performed during the work presented in chapter 4) we have observed lower percentage difference between regular and transpose sparse matrix computation times for parallel symmetric SpMVM implementation of MFDn using CSB_COO than CSC. An example of a matrix M stored using the CSB_COO format is shown in Figure 2.2. Note that for illustration purposes only, the sparse matrix is divided into square blocks of size $\beta = 2$ which is much smaller than usual.

2.4 LA execution in MFDn

The parallel symmetric SpMVM process, which is the main part of the parallel LA implementation in MFDn, is described in [9]; we will refer to this implementation as “s1” throughout the thesis. In a distributed memory environment a symmetric SpMVM, $b = A * x$, can be accomplished by distributing the nonzero elements of the large sparse symmetric matrix A and the vector x as evenly as possible among participating compute nodes. The symmetry of the matrix A can be leveraged to save memory space by storing only half of the matrix (although doing so requires both the regular SpMVM and SpMVM$^{\mathsf{T}}$ operation to be performed by a single

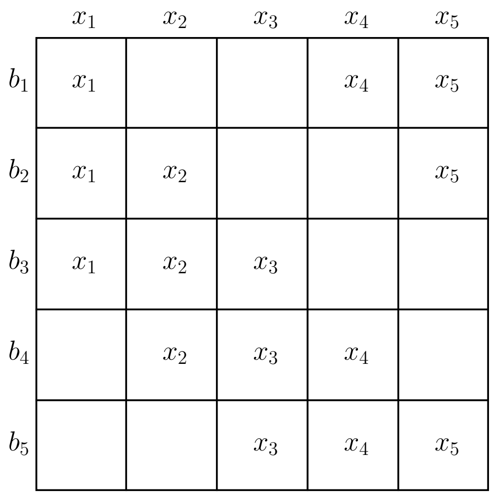


(a) Data distribution. (b) Process mapping.

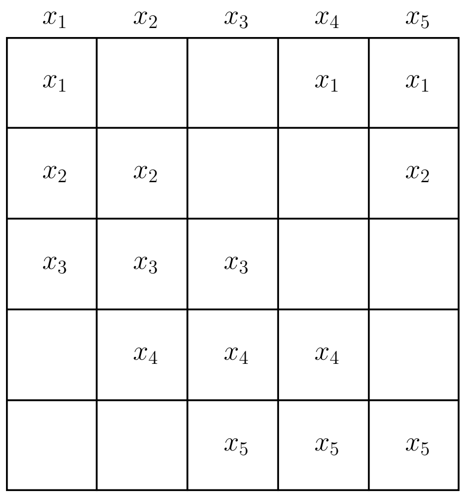
Figure 2.3: Distributed symmetric SpMVM data distribution and process mapping. Partitioning of the large symmetric matrix, A, among $n_d(n_d+1)/2 = 15$ MPI processes with $n_d = 3$ diagonals.

node). For this purpose, in MFDn the sparse matrix A is distributed on a half 2D processor grid, where only half of the sparse matrix is stored as shown in Figure 2.3a. Each process, see Figure 2.3b, stores a submatrix of A denoted by $A_{(r,c)}$, containing the nonzeros of that submatrix in a load-balanced way [41]. The total number of processes participating in SpMVM is $n_d(n_d+1)/2$, where n_d is an odd integer denoting the number of diagonal processes.

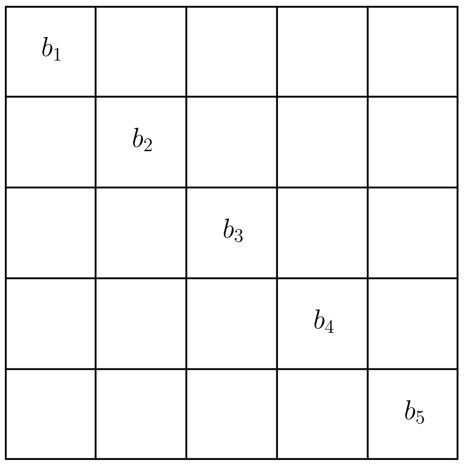
Row subcommunicator groups (RSg) are setup to facilitate communications along the rows of the 2D processor grid. Similarly, column subcommunicator groups (CSg) are used for communications along the columns. For both RSg and CSg diagonal MPI processes act as the roots of the subcommunicators. Additionally, there are diagonal (DSg) and off-diagonal (OSg) subcommunicator groups which are composed of diagonal and off-diagonal processes, respectively. The size of both RSg and CSg is equal to $(n_d+1)/2$, whereas the size of DSg is n_d, and of OSg is $n_d(n_d-1)/2$. For example, in Figure 2.3, $n_d = 5$ and the number of processors in each |RSg|=|CSg| = 3.



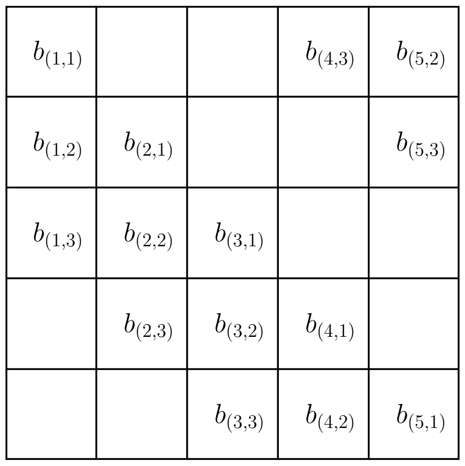
(a) Input vectors, x_i, are broadcast among CSg for SpMVM computations.



(b) Input vectors, x_i, are broadcast among RSg for SpMVM$^{\mathsf{T}}$ computations.



(c) Reduction of b_i vectors on diagonal processes.


$b_{(1,1)}$ $b_{(4,3)}$ $b_{(5,2)}$
$b_{(1,2)}$ $b_{(2,1)}$ $b_{(5,3)}$
$b_{(1,3)}$ $b_{(2,2)}$ $b_{(3,1)}$
$b_{(2,3)}$ $b_{(3,2)}$ $b_{(4,1)}$
$b_{(3,3)}$ $b_{(4,2)}$ $b_{(5,1)}$

(d) Partitions of b_i vectors stored on each process.

Figure 2.4: Distributed symmetric SpMVM process. Initially, x_i reside on diagonal processes only.

The vector x is partitioned according to the column/row partition of A into n_d parts, where $i = 1, 2, \ldots, n_d$ as shown in Figures 2.4a and 2.4b. Each x_i is initially generated on the i^{th} diagonal processor. Since we store only half of the symmetric sparse matrix A, the processor assigned to the (r, c) position of the grid needs to perform two operations on the submatrix $A_{(r,c)}$, the regular SpMVM and SpMVM$^{\mathsf{T}}$, see Figure 2.5. Therefore the processor at the (r, c) grid point needs two input vectors: x_r from the diagonal processor in its RSg and x_c from the diagonal processor in its CSg.

The parallel LA then proceeds as follows. Initially, diagonal processes generate the x vector, and then broadcast, MPI_BCAST, subsequent x_c portions of x along CSg (Figure 2.4a). This part is done only once before the LA iterations start. Once this x_c is received by the CSg members, all of the MPI processes can now perform the local SpMVM, i.e., $b_r = A_{(r,c)} * x_c$. During this stage we do overlapping of communication with computation, by devoting a single OpenMP thread to broadcast, MPI_BCAST, the x_r[1] along the RSg members. This part is shown in Figure 2.5a.

Next, once the SpMVM is done, every MPI process is ready to perform the SpMVM$^{\mathsf{T}}$,i.e., $b_c = (A_{(r,c)})^{\mathsf{T}} * x_r$. While doing this part diagonal MPI processes can collect the SpMVM results, using the MPI_REDUCE, along the RSg members. This part is also done by a single OpenMP thread and is overlapped with computation as shown in Figure 2.5b.

Once the SpMVM$^{\mathsf{T}}$ is done, we collect the results of SpMVM$^{\mathsf{T}}$ on diagonal MPI processes, and then scatter the resultant b_i further among the CSg members as shown in Figures 2.4c and 2.4d. These chunks $b_{i,j}$ form together a single vector b. This part is a two step operation, but we use an efficient MPI routine called MPI_REDUCE_SCATTER. The post-processing on the vector b (e.g., orthogonalization as required by Lanczos and CG algorithms [59]) is performed using these chunks $b_{i,j}$, evenly distributed over all available MPI processes, as shown in Figure 2.4d. Finally, a gather-operation, MPI_ALLGATHERV, after post-processing yields the distribution of Figure 2.4a for subsequent iterations.

[1]Note that on diagonal processes $x_c = x_r$.

```
!OMP Parallel Private(i,j,k,tid)
!OMP Default(Shared)
tid = OMP_GET_THREAD_NUM()
wiloc(:, tid) = 0
if (tid == 0) then
 MPI_BCAST(v_i,...,RSg)
endif
!OMP Do schedule(dynamic)
do j = 1, nloc
 do i = colptr(j), colptr(j+1)
   k = rowind(i)
   wiloc(k,tid) += A(i)*v_j(j)
 enddo
enddo
!OMP End Do
!OMP Do
do i=0, nloc
 w_i(i) = SUM(wiloc(i))
enddo
!OMP End Do
!OMP End Parallel
```

(a) $w_i = A_{ij}v_j$ that overlaps the broadcast of v_i with the SpMVM computations.

```
!OMP Parallel Private(i,j,k,tid)
!OMP Default(Shared)
tid = OMP_GET_THREAD_NUM()
if (tid == 0) then
 call MPI_REDUCE(w_i,...,RSg)
endif
!OMP Do schedule(dynamic)
do j = 1, nloc
 do i = colptr(j), colptr(j+1)
    k = rowind(i)
    w_j(j) = w_j(j) + A(i)*v_i(k)
 enddo
enddo
!OMP End Do
!OMP End Parallel
```

(b) $w_j = A_{ij}^{\mathsf{T}}v_i$ that overlaps the reduction of w_i with the SpMVM$^{\mathsf{T}}$ computations.

Figure 2.5: A hybrid OpenMP/MPI implementation of SpMVM and SpMVM$^{\mathsf{T}}$ with overlapping of computation and communication as done in MFDn. In this example submatrices are stored in CCS/CSC format.

In large-scale clusters, the mapping of the processes to the physical processors can significantly influence the communication overheads as we will discuss in chapter 5. Because in “s1” the collective communications along the RSg’s are already overlapped with computations, one needs to consider efficiently mapping the CSg, since it is not overlapped. In Figure 2.3b, we see a column-major mapping of the processes onto the half 2D processor grid, which is a heuristic aimed at optimizing the communications along CSg’s. For this reason and additionally because we have group sizes of both RSg and CSg equal (as well as almost perfectly equal number of submatrix nonzero elements on each of the off-diagonal MPI processes) this mapping is called load-balanced column-major (LBCM) [9].

The LBCM heuristic is effective for the Cray supercomputer architectures, because Cray's scheduler does a best effort in placing consecutively ranked MPI processes onto nearby physical processors [9]. However, as we demonstrate in chapter 5, this mapping is generally not efficient on Mira, which has a 5D torus interconnect and uses a block-based job scheduler.

Finally, as we have already explained in some cases we might have a situation where the maximum number of nonzero submatrix elements on diagonal processes is more than the maximum number of nonzero submatrix elements on any of the off-diagonal processes. In these cases MFDn introduces n_{extra} number of MPI helper processes for each of the diagonal processes to divide the local computation, and thus we will need a total of $n_d((n_d + 1)/2 + n_{extra})$ MPI processes. Also, in this case we will have |RSg|=$(n_d + 1)/2$ and |CSg|=$(n_d + 1)/2 + n_{extra}$. Alternatively, we can deal with the diagonal part of the matrix separately, and distribute the excess nonzero matrix elements evenly among the CSg members. The latter treatment has recently been developed, partly motivated by the communication overhead data presented in chapter 5. In either case we will explicitly mention in the text if any of these two approaches are under consideration.

CHAPTER 3. INTRODUCTION OF GPGPUs IN *AB INITIO* NUCLEAR PHYSICS CALCULATIONS

In this chapter we present the main results of our work [1] targeted for the acceleration of MFDn using GPGPUs [52]. Specifically, the chapter is organized as follows. Section 3.1 introduces the problem under consideration, and section 3.2 gives a brief overview of the GPGPU programming. Section 3.3 presents MFDn tasks to be executed on GPGPUs and the calculations needed to construct the matrix elements, which have been ported to GPGPUs. Section 3.4 discusses the main features adapted for better utilization of the GPGPUs. Finally, section 3.5 presents and discusses the experimental results, while section 3.6 concludes the findings.

3.1 Introduction

Determining each matrix element location and value in MFDn takes a significant amount of time, on the order of 20% to 40% of the entire CI calculation in representative cases [42]. Furthermore, this costly procedure has highly parallel steps, i.e., each matrix element may be generated independently, and thus should yield good performance on the "single instruction, multiple data" (SIMD)-type highly parallel accelerators, such as GPGPUs.

GPGPUs have been available for general-purpose programming for several years already, and exhibit performance increases in comparison with CPU-only codes, given intelligent GPGPU memory accesses and usage and a high degree of fine-grain parallelism. In the case of CI computations, such as the *ab initio* nuclear structure application MFDn, hundreds of GPGPU threads may participate in the generation of Hamiltonian matrix elements in parallel. However, to achieve and sustain a high degree of parallelism, the serial construction procedure

[1] Modified from a paper published in *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International* [52].

has to be redesigned for GPGPUs, and a multithreaded parallel programming model, such as Compute Unified Device Architecture (CUDA) from NVIDIA[2], used to implement the new design. This chapter outlines this design and implementation of the Hamiltonian matrix element construction and presents initial results as well as notes on the efficiency of GPGPU utilization.

3.2 CUDA Programming Model

One of the significant differences between GPGPU and CPU architectures is in the number of transistors dedicated to data computing [47]. GPGPUs contain hundreds of cores and support a highly multithreaded programming model, providing an inherently parallel computational environment. The CUDA framework provides easy access to usage of the GPGPUs and enables writing multithreaded programs without the knowledge of the architectural design, internal organizational specifics of the GPGPUs, and a low-level graphics API. With CUDA, it is now possible to program the GPGPUs using high-level programming languages, such as C. A CUDA *kernel* is a function that is executed on the GPGPU in a single-instruction multiple-thread manner using a large number of parallel threads. The main difference between the functions in high-level programming languages and the *kernels* is that the *kernels* are executed in groups of *blocks* in parallel across many cores of the GPGPU. In general, the top-down hierarchy for the thread is as follows: *Grids* are composed of *blocks*, which are themselves composed of threads. To arrange a GPGPU computation, the user specifies the number of threads per *block* and number of *blocks* per *grid*.

There are different types of memory spaces provided to threads in CUDA, such as read-write and read-only. Threads are able to access these memories throughout the execution. Every thread can read-write to per-thread local memory (as well as registers); each thread in a *block* is able to use the *shared* memory with the lifetime of the *block*; and all the threads can access the global memory, which can be as large as several gigabytes in the latest NVIDIA GPGPUs. On the other hand, the read-only memories are termed *constant* and *texture*, which can be accessed by all threads, i.e., per-grid. The constants and kernel arguments are stored in *constant* memory, whereas *texture* memory is optimized for 2D spatial access pattern.

[2]http://www.nvidia.com

3.3 MFDn Tasks to Be Executed on GPGPUs

MFDn's major steps have been presented in detail in chapter 2. The construction of the Hamiltonian matrix involves a large number of independent calculations for its elements, and as such is an obvious candidate for parallelization; the GPGPU acceleration of that stage is the subject of this chapter. The LA's iterations are computationally intensive and memory-bound, which cannot be broken down into independent calculations nearly as easily. The implementation of a block-algorithm as an alternative to the iterative LA is currently under investigation, and this may eventually also lead to a GPGPU implementation of the actual diagonalization phase. Also, with a highly efficient GPGPU implementation of the matrix construction it may be possible to perform the matrix vector multiplications 'on the fly', without explicitly storing all nonzero matrix elements in-core, thus reducing the memory footprint significantly.

3.3.1 Closer look at the Hamiltonian matrix element evaluation

To reduce the memory footprint in the beginning of the second stage, see chapter 2, of the MFDn execution, a new procedure, termed ME3M, has been elaborated and implemented recently [58, 57, 54, 43] for 3-body interactions. MFDn uses a basis called m-scheme, which is convenient to use but memory-intensive; the ME3M procedure stores the 3-body interactions in a more compressed basis by making use of special properties of adding angular momentum and isospin in quantum mechanics. A basis transformation algorithm is executed in order to generate the 3-body m-scheme matrix elements as needed for a given many-body matrix element. This compressed basis is denoted as coupled-JT [58, 57]. The ME3M procedure allows MFDn to request individual 3-body m-scheme matrix elements without storing the entire, possibly quite large, m-scheme 3-body matrix, to construct a many-particle Hamiltonian matrix element. For example, a 3-body matrix of size 100GB using the m-scheme basis would be equivalent to a matrix only around 2GB in size in the coupled-JT basis, amounting to significant memory savings.

The difference between the m-scheme and coupled-JT bases is in how they represent isospin (a quantity that has to do with whether a particle is a proton or a neutron) and angular momentum. The m-scheme basis specifies the exact angular momentum and isospin of each particle separately, while the coupled-JT basis adds all three angular momenta together and specifies the totals; likewise for isospin. The latter 3-body matrix representation is much more compact and, since the underlying 3-particle interaction depends only on total angular momentum and isospin, no information is lost in this "compression".

3.3.2 Matrix structure

Transforming a 3-body matrix between the coupled-JT scheme and the m-scheme is accomplished through a change-of-basis calculation. In such a change-of-basis calculation each element in the new basis is a linear combination of elements from the old basis, weighted by various projections between the old and the new basis vectors that specify the matrix row and column labels in the respective basis. These projections define transformation coefficients that go into the matrix elements of a matrix D used in the change-of-basis equation:

$$A' = D^T A D, \tag{3.1}$$

where each element of the D matrix is developed from a sequence of additions of angular momenta and isospins. In this case, the coupled-JT basis is much smaller than the resulting m-scheme basis, so the matrix multiplication would look like in Figure 3.1.

An m-scheme matrix element is specified by two m-scheme basis vectors, each of which corresponds to a set of coupled-JT basis vectors, per physics considerations of the angular momenta and isospins. Together, these sets specify a region of the coupled-JT matrix. This region is the one that contributes to the m-scheme element in question. An m-scheme element, then, can be calculated from a sum over coupled-JT elements, each multiplied by the appropriate projections as shown in Figure 3.2.

MFDn requests elements one-by-one from the generated m-scheme matrix in the process of constructing the many-body Hamiltonian matrix to be diagonalized. The generated 3-body m-scheme matrix is never actually stored as a full matrix since ME3M generates necessary

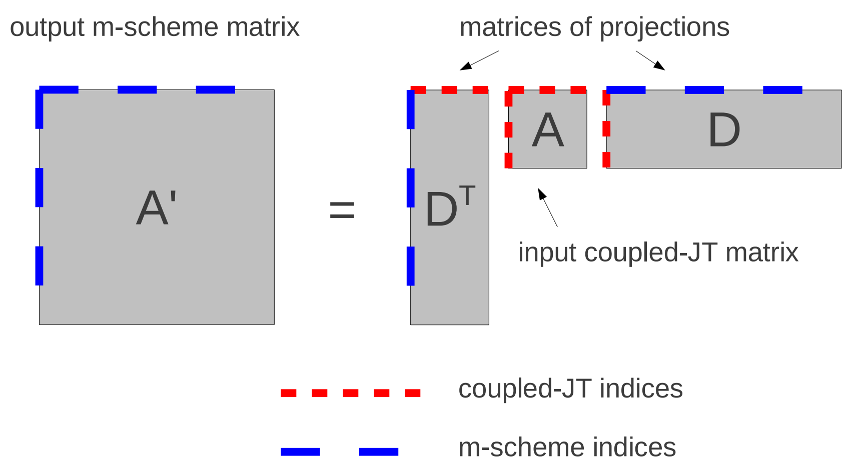


Figure 3.1: The coupled-JT to m-scheme transformation.

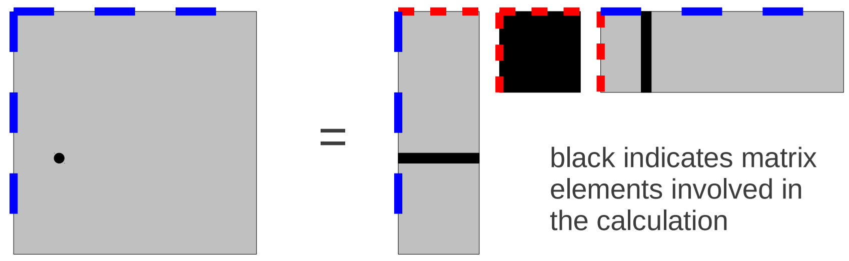


Figure 3.2: The m-scheme element calculation.

m-scheme matrix elements as needed. Note that only a small subset of coupled-JT matrix elements actually end up contributing to the resulting m-scheme matrix element. The CPU-based implementation of ME3M represents the coupled-JT matrix intelligently, so that the contributing elements can be easily accessed [57].

A 3-body matrix element can be indexed by two vectors from the basis in which the matrix is constructed. These vectors will be referred to as the row state and the column state. Each row or column state has three single-particle states (SPSs). Certain indices for each SPS remain fixed when transforming between the coupled-JT and the m-scheme bases. These fixed SPS identifiers (quantum numbers) are n (the radial quantum number), l (the orbital quantum number) and j (the total SPS angular momentum). The set nlj serves then as a useful identifier for the immutable aspects of each of the three SPSs within each row state or column state. Because of these immutable features, these two matrices (the coupled-JT and the one generated from the coupled-JT to m-scheme transformation) are both divided into blocks (see Figure 3.3),

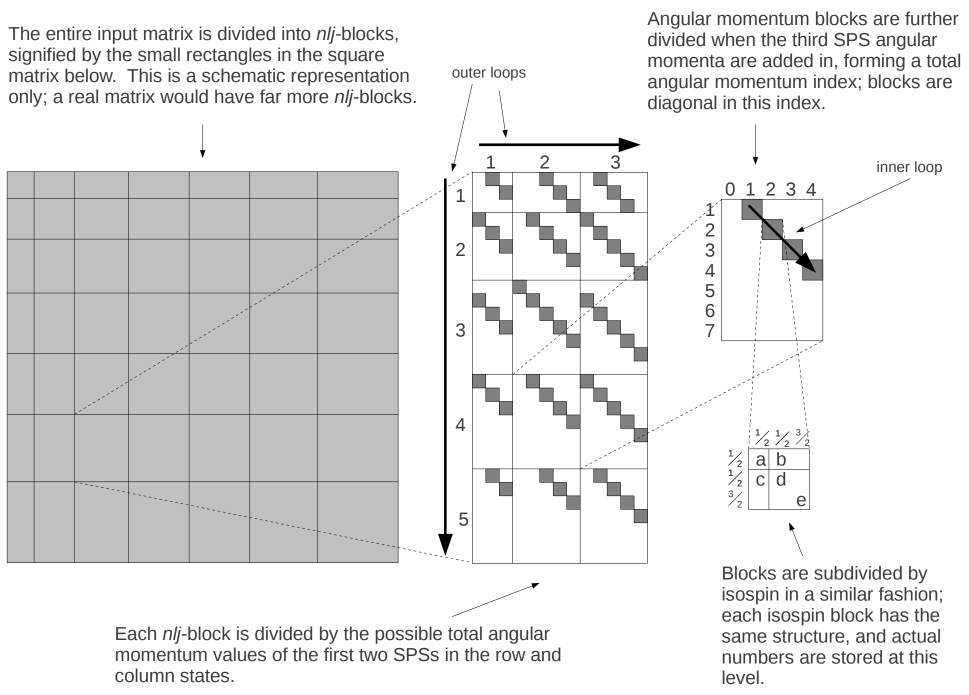

The entire input matrix is divided into *nlj*-blocks, signified by the small rectangles in the square matrix below. This is a schematic representation only; a real matrix would have far more *nlj*-blocks.
outer loops
Angular momentum blocks are further divided when the third SPS angular momenta are added in, forming a total angular momentum index; blocks are diagonal in this index.
inner loop
1 2 3
1 2 3 4 5
0 1 2 3 4
1 2 3 4 5 6 7
½ ½ 3/2
a b
c d
e
Each *nlj*-block is divided by the possible total angular momentum values of the first two SPSs in the row and column states.
Blocks are subdivided by isospin in a similar fashion; each isospin block has the same structure, and actual numbers are stored at this level.

Figure 3.3: Matrix structure.

such that each block is defined by three nlj sets for its row states and three nlj sets for its column states. Each block of this type is referred to as an nlj-block. The transformations do not mix blocks, i.e, a block of the A' matrix from equation 3.1 can be constructed entirely from the elements of the block in the coupled-JT matrix A having the same three row state nlj indices and the same three column state nlj indices as the resulting block in matrix A'. Since the j indices of a row state specify the angular momentum magnitudes of its SPSs, and likewise for a column state, each nlj block has a definite total angular momentum for each of the three row SPSs and each of the three column SPSs.

Inside each nlj block for the first two column-state SPSs, the total angular momentum j-values are added to form a new column index, and likewise for the row state. Note that adding two angular momenta produces a *set* of total angular momenta. This new index divides the nlj block into subblocks, each of which has a definite total angular momentum for the first two column SPSs and for the first two row SPSs. These subblocks are subdivided by another angular momentum index; the column state angular momentum of the subblock is added to the third column state SPS total angular momentum to produce a final total angular momentum index, and likewise for the row state. Thus, each block at this level has a definite final total angular momentum for the column state and for the row state. Note that coupled-JT matrix elements are zero if their column and row states have different final total angular momenta; this means that nonzero elements occur in diagonal regions within the nlj block.

Finally, each block with definite total angular momenta is subdivided twice by adding isospin the same way angular momentum was added in previous divisions. Isospin works mathematically the same way as angular momentum does. Here, only protons and neutrons are considered, i.e., particles with isospin 1/2, and isospin projection $+1/2$ and $-1/2$, respectively. Therefore, each isospin block has the same form, with zero elements in the same locations. There are only five nonzero elements per isospin block. To implement the isospin basis transformations, it is more efficient to unroll the corresponding loops and hard-code which matrix elements to add.

3.3.3 Basis Transformation Algorithm

The entire coupled-JT matrix is stored as a flat series of sets of five floating point values, as shown in Figure 3.3. Looping over the block indices as described above references the nonzero sets in a predictable sequence, so if the code accesses them in that sequence it is sufficient to store only the nonzero sets. Since one A' matrix element is constructed entirely from a single nlj block, the algorithm only needs to iterate over one block; the locations of nlj blocks within the flattened coupled-JT matrix are stored in an array.

The heart of the coupled-JT code, as implemented in the ME3M procedure, is a set of three nested for-loops. The bounds of the first and second loops are determined by the m-scheme element requested. The innermost loop bounds, however, depend on the position in the first two loops. Inside the inner for-loop there is a linear combination of five coupled-JT matrix elements, each weighted by the product of several so called Clebsch-Gordan coefficients (CG-coefficients). These CG-coefficients are real-valued quantities whose products form the matrix elements of the D-matrix, shown in Figure 3.1. A lookup table for the CG-coefficients is generated at initialization.

Algorithm 3: To transform coupled-JT to m-scheme

```
$L1$ ← determineFirstLoopBound
$L2$ ← determineSecondLoopBound
for 1 to L1 do
    for 1 to L2 do
        $L3$ ← determineInnerLoopBound
        for 1 to L3 do
            add input matrix elements weighted by CGs
        end for
        accumulate inner loop sum weighted by a CG
    end for
    accumulate second loop sum weighted by a CG
end for
return accumulated sum
```

The loop structure iterates over all the input coupled-JT total angular momentum arrangements, and the linear combination inside the innermost loop is a sum over the coupled-JT total isospin arrangements. The output of the algorithm is, thus, a linear combination of all the contributing coupled-JT matrix elements. Contributions are summed within each loop and then added to the sum in the next-higher loop. This implementation allows an entire loop to be efficiently multiplied by the same CG-coefficient. Algorithm 3 shows the pseudocode for the transformation process. Taken as a whole, the loop structure can be thought of as a set of J-loops over total angular momentum and a set of unrolled T-loops over total isospin, with one J-loop and one T-loop for each m-scheme basis vector, and several sub-loops over component couplings leading to the total coupling.

The coupled-JT matrix is stored in such a way that its elements will always be accessed sequentially for a given generated element and its respective for-loops. Thus, once the first coupled-JT matrix element is located, one may efficiently access all the needed elements for that generated matrix element. The first coupled-JT element is determined from a set of conditional statements and a look-up table that relates indices known in m-scheme to a flat coupled-JT matrix index.

3.4 GPGPU Implementation

The MFDn many-body Hamiltonian is partitioned across multiple processors, as it is shown in Figure 2.3, where every processor needs more or less random access to the basis transformation results (i.e., nonzeros of matrix A' in equation 3.1). The full set of m-scheme 3-body matrix elements may quickly grow over 100GB, which is not currently practical to process without basis compression considerations, such as those provided by the ME3M procedure.

CUDA was used for generating the m-scheme 3-body matrix elements with GPGPUs, NVIDIA Tesla C2050, code-named Fermi, from the information provided by the coupled-JT 3-body matrix elements. This procedure, as seen in Algorithm 3, is easily parallelizable in a multithreaded fashion: Each m-scheme 3-body matrix element is calculated by its own CUDA thread, and the CUDA *kernel* is invoked over many elements at once. The entire Algorithm 3 was put into the *kernel*, without modifying it significantly, and given a wrapper function to transfer chunks of matrix element requests in the form of sets of SPS indices to the GPGPU and retrieve a corresponding group of calculated 3-body m-scheme matrix elements from the device. The m-scheme SPSs are specified by the nlj set of indices as well as by the m and t values, which are the angular momentum and isospin projection quantum numbers. These five indices are all flattened into one linear SPS index. An m-scheme 3-body matrix element can then be specified by a set of six linear SPS indices. The uniformly-sized chunks of these six-index sets are transferred to the GPGPU. The size of each chunk will influence the performance of the GPGPU implementation as will be shown in Section 3.5.

Most of the multidimensional arrays were flattened to be referenced only by a single index because of the efficiency and convenience while transferring these data to and from GPGPU. The six-dimensional array relating the six nlj indices of an nlj block to the flattened index of the first coupled-JT 3-body matrix element in that block is not straightforward to flatten since the orderings of the six linear nlj indices are not strongly restricted, and the value of one can control the range of another (referred to as "jagged" linear nlj indices). Also, each linear SPS index may run as high as $10,000$ or more in realistic applications. Thus, this array may be flattened only with a significant computation overhead resulting from the multiple nested summations needed to map a single index to a multidimensional jagged-array index. Furthermore, transferring this jagged array to the GPGPU involves an exorbitant number of very small and inefficient memory copies. In the initial GPGPU implementation, this six-dimensional array was not transferred to the GPGPU but, instead, was processed on the CPU to precompute only the required part to be transferred to the GPGPU along with the sets of the 3-body matrix element requests. Subsequent implementations transferred the entire array to the GPGPU and referenced it with six indices, as in the CPU code. This modification produced better results by a factor of about 4.5x than in the initial attempt without considering the jagged array transfer time, which occurs only once, during ME3M initialization. The CG-coefficients, index structures, and the associated matrix element values are computed using the (mostly unmodified) original implementation of the ME3M procedure and then transferred to the GPGPU during the initialization step.

The minimization of the number of requests to the *global* GPGPU memory on a per-thread basis was performed as much as possible. Also, the use of the *constant* GPGPU memory was considered for some arrays that are accessed by the GPGPU threads in a read-only fashion and that may fit in this memory. Unfortunately, this attempt did not improve significantly the overall runtime of the kernel. The GPGPU consists of multiple streaming multiprocessor (SM), specifically in our case we had 14 of them, each with 32 cores. Each SM of the GPGPU used has 64KB of on-chip memory (*shared memory* + L1 cache). These 64KB can be configured as $48 + 16$KB or $16 + 48$KB between *shared* memory and L1 cache. All of the threads in a thread *block* will run on a single SM, which will schedule the threads in groups of 32 parallel threads called *warps*. When fine-tuning the kernel performance, preference for a larger L1 cache size for the kernel improved the GPGPU execution time on average by about 5.15% for the smaller test case considered and by about 8.46% for the larger one.

To summarize, the experimental results shown in Section 3.5 do not use the constant GPGPU memory but set the preference to use larger L1 cache for all the GPGPU runs. As per the CUDA programming model, the space of requested elements is divided into blocks, which are subdivided into threads, with one thread per requested element. Using the CUDA occupancy calculator[3], the highest occupancy was attained for several block sizes, and almost all the possible combinations of width and height for these block sizes were tested. For the experiments presented in Section 3.5, the block size is taken as having 64 threads (64 in width and 1 in height), which has given the best timings, and a grid of blocks has as many columns as are required to accommodate the number of elements requested. Several different variations of grid dimensions were tested as well, but these did not show significant decrease in the execution time.

[3] https://developer.nvidia.com/category/zone/cuda-zone

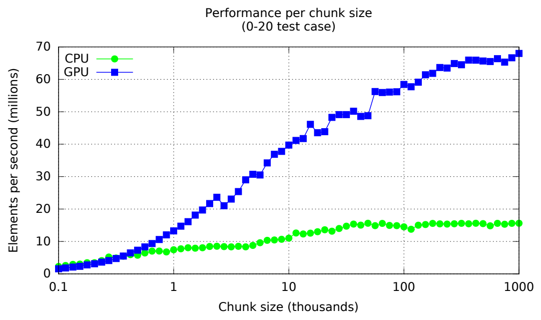
3.5 Experimental Results

The GPGPU node cluster Dirac at NERSC was used for all the test results. Specifically, an NVIDIA Tesla C2050, code-named Fermi, with 3GB memory and 448 parallel CUDA cores was used. Each of the nodes has two Intel 5530 2.4 GHz QPI quad-core Nehalem processors. The Dirac computing platform was used both for the GPGPU code timings and for the comparison with the CPU multithreaded version.

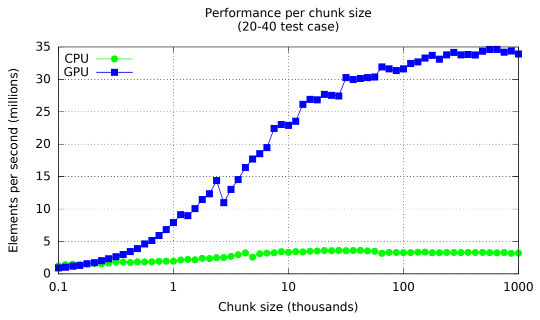
The experiments reflect only a particular part of the MFDn calculation, namely the transformation from the input coupled-JT 3-body matrix elements to the m-scheme 3-body matrix elements needed in the construction of the many-body Hamiltonian matrix. The algorithm used for the GPGPU code was taken directly from the CPU version; and, for a fair comparison, the GPGPU performance optimizations, such as flattening of various arrays, were retrofitted to the CPU version. In other words, the GPGPU and CPU versions compared here differ only in the particulars of the programming model used to execute on multithreaded CPU and GPGPU versions on Dirac.

Depending on the desired accuracy of the MFDn calculation, 3-body matrices of different sizes are used. The largest test cases considered here employ 3-body matrices with about $456,000$ elements requiring about 1.7MB of main memory. Note that the present tests were conducted in “stand-alone” mode, i.e., without their integration into the actual MFDn code. Instead, the proposed GPGPU implementation of the basis transformation algorithm was compared against a CPU-based multicore implementation of the ME3M procedure. Such stand-alone testing allows a better investigation of the effects of the GPGPU parallelization in timing comparisons with the CPU implementation. In order to integrate the stand-alone test code into MFDn additional issues had to be resolved [55].

The initial test case contains a 3-body matrix consisting of 1 million elements, which were generated from the beginning of the m-scheme SPS basis, and uses elements with SPS indices in the 0–20 range, inclusive. (Recall that each m-scheme element is indexed by six linear SPS indices.) This range generally involves very short decoupling loops and a correspondingly small amount of computation. Next, the tests were done with the single-particle indices in the 20–40 range, inclusive, which yielded a more intensive computation, without increasing the memory transfer amount between CPU and GPGPU. This larger test case showed better performance, as expected.



(a) 0-20 test case



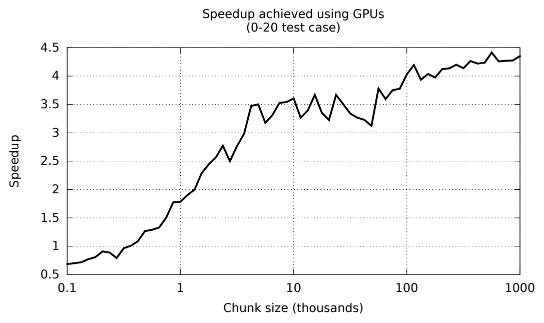
(b) 20-40 test case

Figure 3.4: Performance comparison between GPGPU and CPU runs.

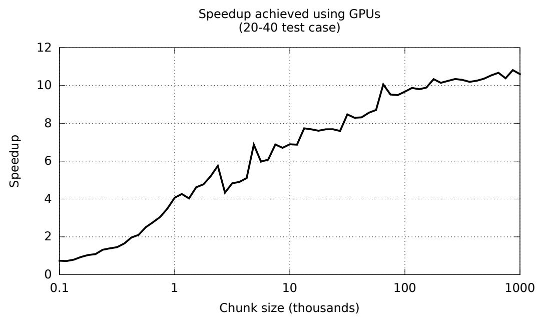
Since the allocation, initialization, transfer, and cleanup stages for the index structures and coupled-JT matrix element values happen only once, they were not timed and are not shown here. On the other hand, the results do include the times of sending the m-scheme matrix element request to GPGPU and of transferring the calculated matrix elements back to CPU.

For each test case and different granularity of matrix element processing, the rate of matrix element generation, as matrix elements-per-second, and its speedup as compared with the CPU-based code are shown in Figures 3.4 and 3.5, respectively. The results presented are for chunk sizes in the range from 100 to 1 million and are the averages of five runs. The CPU timings shown in Figures 3.4 and 3.5 are with eight OpenMP threads, which is the maximum number of CPU threads allowed without oversubscription. It was observed experimentally that the runs using eight OpenMP threads have achieved a 2x speedups as compared with the four-threaded ones already at the chunk sizes of $36,517$ and $8,659$ for the 0–20 and 20–40 test cases, respectively, and continued to grow with a slightly superlinear speedup tendency as the chunk sizes approach 1 million.

The GPGPU code gave a performance improvement of almost 4.4x over the CPU code for the 0–20 test case (Figure 3.5a) and almost 10.8x for the more computationally intensive 20–40 test case (Figure 3.5b). Notice that the speedup achieved by GPGPU increases quite slowly up



(a) 0-20 test case



(b) 20-40 test case

Figure 3.5: Speedup with respect to the CPU execution when different chunk sizes are used.

to a certain chunk size (about 365 and 177 elements per chunk, for the 0–20 and 20–40 cases, respectively). In conjunction with the timing data, hardware counters provided an estimate of the degree to which the code is memory-bottlenecked. Disregarding the transfers of element requests to the GPGPU and calculated elements back from it, which take time but do not involve significant calculation, the GPGPU ran at approximately 3% of its peak FLOPS for the 0–20 test case and approximately 6% of its peak FLOPS for the 20–40 test case.

3.6 Conclusions

This chapter tackles a challenge currently faced by the MFDn code: The matrix sizes of the input 3-body matrix elements become unmanageable, so new approaches to obtain the m-scheme 3-body matrix elements are required. One approach is to read the 3-body matrix elements in coupled-JT format, which enables MFDn to obtain the same amount of information as in the (memory-intensive) m-scheme but in a much more memory-efficient manner. This approach is well-parallelizable and has been adapted for GPGPUs. Initial experiences of porting the coupled-JT to m-scheme transformation to GPGPU using CUDA have been presented in this chapter and already show promising results in the range of four-to-ten fold improvements. Further improvements and analysis are needed, however, for both the CPU and GPGPU implementations. For example, the GPGPU code may be able to take advantage of the texture memory and multiple streams.

We have incorporated enhanced GPGPU implementation [52] described in this chpater into MFDn, and full production runs on Titan have been made. Titan is a, Cray XK7, supercomputer located at ORNL, with a hybrid architecture: Each compute node contains both a conventional CPU and a GPGPU accelerator. Each CPU processor of Titan is a 16-core 2.2GHz AMD Opteron 6274 (Interlagos) processor with 32 GB of RAM. The nodes are connected with Gemini interconnect. In addition, each of the physical compute nodes of Titan contain an NVIDIA Tesla K20 GPGPU accelerator (which is based on NVIDIA's Kepler architecture) with 6GB of DDR5 memory. We have obtained speedup of approximately $2.2x - 2.7x$ for the matrix construction stage and $1.2x - 1.4x$ for the entire run [55]. Physics results using the GPGPU-enhanced version of MFDn for large-scale production runs on Titan have also been published in [54], and a follow-up paper to [54] is in preparation.

CHAPTER 4. PERFORMANCE ANALYSIS OF DISTRIBUTED SPMVM ALGORITHMS FOR MULTI-CORE ARCHITECTURES

This chapter presents results of our work [1] on performance analysis of both symmetric and non-symmetric parallel SpMVM implementations [50]. Here, in chapter 4, we give a brief introduction to the work performed in section 4.1 and in section 4.2 we discuss related literature. Next, in section 4.3, we give brief details of the non-symmetric parallel (general distributed) SpMVM methods presented in literature. In Section 4.4 we outline the network load model that is used to assess the load incurred on the interconnection network by the SpMVM implementations. Finally, experimental results and their analyses are presented in Section 4.5 and Section 4.6 concludes the chapter.

4.1 Introduction

Sparse matrices arise in various contexts in scientific computing. SpMVM forms the main kernel in iterative methods used for solving large systems of linear equations or eigenvalue problems. The focus of this chapter is on analyzing the performance of distributed memory SpMVM algorithms for *symmetric* matrices with *irregular* sparsity patterns. Such matrices arise in quantum many-body calculations and graph analytics among others. For example, iterative eigensolvers are used to extract the few lowest eigenpairs of extremely large symmetric sparse matrices that correspond to the ground state and low-lying excited states of nuclei [41]. Spectral techniques are also widely used in the analysis of large-scale graphs [25, 45, 11]. The results presented may also be useful in other problems where the effort to identify and exploit the underlying sparsity structure is prohibitively high.

[1]Modified from a paper published in *The Journal of Concurrency and Computation: Practice and Experience, 2015* [50].

Since our focus is on matrices with irregular sparsity patterns, we require a good load-balance through simple matrix decomposition schemes, such as 1D or 2D partitioning of the matrix. Ogielski and Aiello [48] have shown that, by randomly permuting the rows and columns of sufficiently large matrices with bounded number of nonzeros in each row and column, one can achieve good load balance with high probability. Therefore, throughout the chapter, we assume that the matrix has already been constructed and distributed in a load-balanced way among different processing units. As described in the literature review section (Section 4.2), several algorithms have been reported for scalable SpMVM on massively parallel architectures under similar assumptions [10, 27, 36, 39, 38, 30, 60]. Our target platforms are distributed memory multi-core machines, in which compute nodes are connected via a high-speed communication network. Within each node, several processing units (or cores) share a common pool of memory. The ideas we discuss are also applicable to more sophisticated multi-core systems.

The contributions of this chapter can be summarized as follows:

- Efficient implementations of previously reported SpMVM algorithms on multi-core architectures by leveraging the hybrid MPI+OpenMP paradigm;
- Performance analysis of a recently proposed distributed symmetric SpMVM algorithm [9] against these efficient implementations [39, 38] under various conditions;
- Comparison of the different SpMVM implementations in terms of the incurred data movement overheads using the network load model presented in [9];
- Extension of a network load model to a Dragonfly network topology and evaluation of the effectiveness of a topology-aware mapping heuristic on clusters with 3D Torus and Dragonfly interconnects.

As we move towards the exascale era, the main concerns for high-end computing systems can be listed as: 1. the increasing gap between the computational power and the communication capabilities [13]; 2. the decreasing amount of memory space per core [34]; and 3. energy consumption of which data movement overheads account for a significant fraction [13]. In this study, we show the detailed performance analysis of a distributed SpMVM algorithm presented in [9] which can achieve excellent scalability through effective overlapping of expensive collective communication operations on multi-core architectures to address the concern number 1 stated above. In addition, this algorithm exploits the symmetry of the underlying matrices to save valuable memory space by storing only half of the sparse matrix in distributed memory [62] and, thus, addresses the concern number 2. We also show that this approach is able to achieve better utilization of resources and that total data movement overheads are significantly reduced, which, in turn, may contribute to energy savings; thereby addressing the concern number 3 of computing at the exascale. For detailed implementation of this algorithm, called “s1”, please refer to section 2.4.

4.2 Literature review

Aliaga and Hernandez [10] present implementations of different parallel algorithms for matrix vector multiplication for both symmetric and non-symmetric cases of sparse and dense matrices. The authors study the load balance among processors using different methods based on ordering, distribution, and adjustment. The data is distributed among processors for both symmetric and non-symmetric cases in block rows or columns. Geus and Rölin [27] present fast parallel SpMVM on symmetric matrices, too. These authors describe optimization techniques, such as software pipelining, register blocking, and matrix reordering on a single node to accelerate SpMVM computations. They present the implementation of these techniques (with the exception of register blocking) in a parallel environment using message passing. Exploiting the matrix symmetry, only the lower triangles of the matrices were stored, and the data were distributed among the processes by block-rows, i.e., 1D decomposition. Actual parallel multiplication was performed using three different choices, one without and two with latency hiding. Another parallel symmetric SpMVM algorithm was presented by Krotkiewski and Dabrowski [36] with several optimizations based on matrix reordering, manual prefetching, and blocking as well as overlapping communication and computation using MPI and POSIX threads.

Lewis and van de Geijn [39] evaluate several distributed memory matrix-vector multiplication algorithms for a parallel implementation of the CG method involving sparse matrices from unstructured grid computations. They show that 1D decomposition schemes scale poorly in general and present a 2D decomposition that scales better than other 2D methods on a hypercube architecture. Lewis *et al.* [38] extend this 2D decomposition to mesh architectures and mention a possibility of overlapping communication and computation for better scalability but provide no details about implementation or its performance. Hendrickson *et al.* [30] have developed an efficient parallel algorithm for matrix-vector multiplication on hypercube architectures with a focus on dense or unstructured sparse matrices using overlapping of communication and computation. Schubert *et al.* [60] present explicit communication overlap in hybrid MPI+OpenMP parallel sparse matrix vector multiplication, which they extended [35] to MPI and GPGPUs. Popovyan [53] has described a parallelization of Lanczos type algorithms on modern architectures using the same technique as in [30], but the author reports no performance improvement as a result of communication and computation overlapping on a 3D Torus architecture.

4.3 Distributed SpMVM Methods for Multi-core Architectures

In this section, we present the different distributed memory SpMVM implementations considered throughout the chapter. Since, we have briefly described the distributed symmetric SpMVM algorithm [9] in chapter 2, specifically in section 2.4, in this section we present efficient implementations on modern multi-core architectures for some general (non-symmetric) SpMVM methods reported in the literature. Throughout this section, we denote the SpMVM operation by $b = A * x$, where A is a large symmetric sparse matrix; x, and b are (dense) vectors.

Specifically, in this section, we briefly describe some previously reported distributed SpMVM approaches [39, 38, 26, 30, 56] and present the techniques we used for achieving their efficient implementations on modern multi-core architectures. The total communication volume of 1D partitioning schemes (such as those reported in [39, 56]) scale linearly with the problem dimension and the number of processors. Therefore, we consider only 2D partitioning based methods. So the total number of processes needed is n_d^2. All approaches described here are based on the data distribution shown in Figure 4.1a and use column-major ordering of processes on the 2D grid (see Figure 4.1b) as discussed in section 2.4. Again two different subcommunicators, as explained in chapter 2, RSg and CSg, are created to facilitate communications along row and columns of the 2D processor grid.

$A_{(1,1)}$	$(A_{(2,1)})^T$	$(A_{(3,1)})^T$	$A_{(1,4)}$	$A_{(1,5)}$
$A_{(2,1)}$	$A_{(2,2)}$	$(A_{(3,2)})^T$	$(A_{(4,2)})^T$	$A_{(2,5)}$
$A_{(3,1)}$	$A_{(3,2)}$	$A_{(3,3)}$	$(A_{(4,3)})^T$	$(A_{(5,3)})^T$
$(A_{(1,4)})^T$	$A_{(4,2)}$	$A_{(4,3)}$	$A_{(4,4)}$	$(A_{(5,4)})^T$
$(A_{(1,5)})^T$	$(A_{(2,5)})^T$	$A_{(5,3)}$	$A_{(5,4)}$	$A_{(5,5)}$

(a) Data distribution.

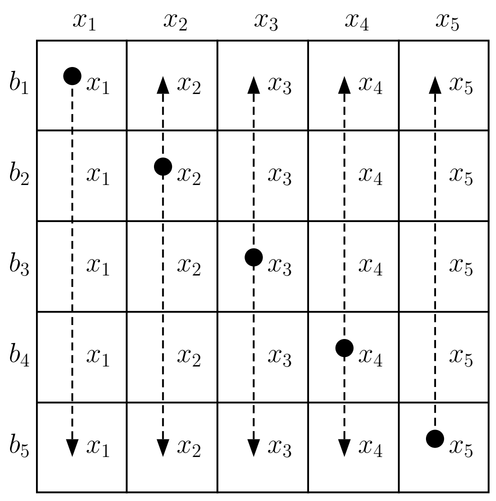
P_1	P_6	P_{11}	P_{16}	P_{21}
P_2	P_7	P_{12}	P_{17}	P_{22}
P_3	P_8	P_{13}	P_{18}	P_{23}
P_4	P_9	P_{14}	P_{19}	P_{24}
P_5	P_{10}	P_{15}	P_{20}	P_{25}

(b) Process mapping.

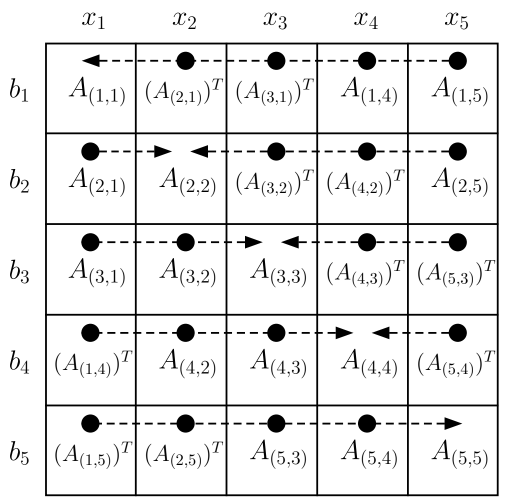
Figure 4.1: General distributed SpMVM data distribution and process mapping.

4.3.1 Approach a1:

Our base implementation is the "Row-Fan-In, Column-Fan-Out" method described in [39]. In this method, the initial x_i vectors reside on the diagonal processes and they are broadcast among the CSg's as shown in Figure 4.2a.


x_1 x_2 x_3 x_4 x_5
b_1 b_2 b_3 b_4 b_5
x_1 x_2 x_3 x_4 x_5

(a) x_i distribution.


x_1 x_2 x_3 x_4 x_5
b_1 b_2 b_3 b_4 b_5
$A_{(1,1)}$ $(A_{(2,1)})^T$ $(A_{(3,1)})^T$ $A_{(1,4)}$ $A_{(1,5)}$
$A_{(2,1)}$ $A_{(2,2)}$ $(A_{(3,2)})^T$ $(A_{(4,2)})^T$ $A_{(2,5)}$
$A_{(3,1)}$ $A_{(3,2)}$ $A_{(3,3)}$ $(A_{(4,3)})^T$ $(A_{(5,3)})^T$
$(A_{(1,4)})^T$ $A_{(4,2)}$ $A_{(4,3)}$ $A_{(4,4)}$ $(A_{(5,4)})^T$
$(A_{(1,5)})^T$ $(A_{(2,5)})^T$ $A_{(5,3)}$ $A_{(5,4)}$ $A_{(5,5)}$

(b) Reduction of the local SpMVM results at the diagonal processor along each of the RSg.

Figure 4.2: Approach "a1": A simple distributed memory SpMVM method.

Once the input vectors are ready, each process performs the local SpMVM computations on its submatrix $A_{(r,c)}$ (Figure 4.2b). Then the partial results are reduced at the diagonal processes in each RSg to produce the final b_i vectors as shown in Figure 2.4c. Finally, the b_i's are scattered (using MPI_SCATTER) evenly among the n_d processes in each CSg for the orthogonalization procedure. Figure 4.3 shows the distribution of the b vector before orthogonalization.

As in Section 2.4, communications along CSg are also optimized by combining the MPI_GATHER after the orthogonalization and the MPI_BCAST of the next iteration into a single MPI_ALLGATHER call. This optimization is also applied to the subsequent approaches to be discussed below which are essentially variants of "a1". However, note in this case that the MPI_REDUCE_SCATTER optimization (as used in "s1" approach) is not possible because reductions and scatters take place in different subcommunicators.

$b_{(1,1)}$	$b_{(2,2)}$	$b_{(3,2)}$	$b_{(4,2)}$	$b_{(5,2)}$
$b_{(1,2)}$	$b_{(2,1)}$	$b_{(3,3)}$	$b_{(4,3)}$	$b_{(5,3)}$
$b_{(1,3)}$	$b_{(2,3)}$	$b_{(3,1)}$	$b_{(4,4)}$	$b_{(5,4)}$
$b_{(1,4)}$	$b_{(2,4)}$	$b_{(3,4)}$	$b_{(4,1)}$	$b_{(5,5)}$
$b_{(1,5)}$	$b_{(2,5)}$	$b_{(3,5)}$	$b_{(4,5)}$	$b_{(5,1)}$

Figure 4.3: Data distribution for the b vector after MPI_SCATTER among CSg's.

4.3.2 Approach a2:

A hybrid parallel implementation of the approach "a1" on multi-core architectures opens possibilities for efficient overlapping of communication and computations [38]. As mentioned in Section 2.4, the column-major ordering of processes on the 2D processor grid yields a topology-optimized CSg. Therefore the main focus should be on overlapping the potentially expensive communications over RSg's.

The main observation behind the approach "a2" is that we propose the commonly used (serial) SpMVM algorithms using the sparse matrix storage formats (described in Section 2.3.2), the elements of the result vector b_i can be generated incrementally. Therefore to start the reduction of b_i, there is no need to wait until all local computations are finished as in "a1". Instead, we designate a communication thread responsible for the reduction of "ready" b_i elements, while other threads continue working on SpMVM computations. We perform reductions in small chunks to reduce messaging overheads.

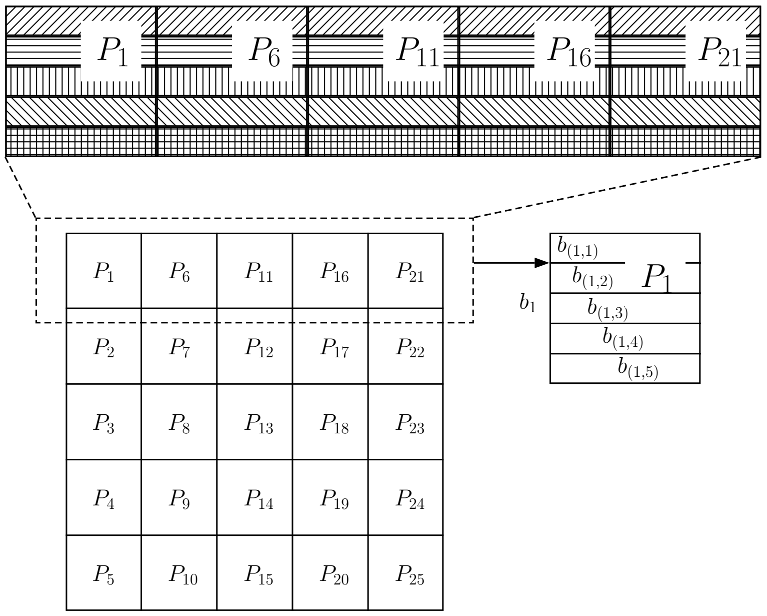


Figure 4.4: Approach "a2": Example of block-row subdivision among processes of the first RSg and reduction on first diagonal process.

We illustrate this idea in Figure 4.4. Here the first RSg consists of processes P_1, P_6, P_{11}, P_{16}, and P_{21}. In this approach, local SpMVM computations are performed in n_d phases. In phase j, the j^{th} chunk of the result vector $b_{(1,j)}$, $j = 1, \ldots, n_d$, is computed. The j^{th} chunk is reduced over the RSg in the phase $j + 1$.

```
do p=1, n_d inside parallel region
 Get/compute bounds for partial SpMVM
 if (p.eq.1) then
  Do partial SpMVM in parallel
 else
  Do partial SpMVM in parallel
  Use one thread for communication
 end if
end do
Do the last communication
```

Figure 4.5: Pseudocode for overlapping of communication and computation in SpMVM using local work subdivision and multi-threading in a 2D grid process.

For example, after the first phase is completed, the designated communication thread performs the reduction of $b_{(1,1)}$, while other threads start the SpMVM computations of the second phase to compute $b_{(1,2)}$. The final $b_{(1,5)}$ chunk is reduced without any overlaps. As in "a1", b_i result vectors are collected at the diagonal processes of their respective RSg's (which is P_1 in Figure 4.4). The post-processing phase initiated by an MPI_SCATTER over the CSg's is identical to that of approach "a2".

In Figure 4.5, we give the pseudocode for overlapping communication and computation (following the idea presented in [38]) in algorithm "a2" using parallel environment. All available threads participate in the partial SpMVM of the first phase (line 4). For the following phases, only single thread is responsible for the communication (lines 7 and 10), while the remaining threads are busy with SpMVM computations (line 6). Synchronizations are necessary during the iterations to prevent race conditions. The result of the last phase is communicated outside the parallel region (line 10). To our knowledge, the implementation of communication and computation overlapping strategy based on the idea presented in [38] and its performance analysis that will follow in Section 4.5 have not been reported in the literature before.

4.3.3 Approach a3:

This method is the implementation of the "Standard Method" described in [39, 38] using the communication/computation overlapping idea of "a2". One potential problem with the approach "a2" is that the diagonal processes which act as the sink for all MPI_REDUCE calls and the source for the MPI_SCATTER call that follows may potentially be communication "hot spots". To prevent this, in approach "a3" the n_dth phase is designated to be the n_dth process of that RSg. In Figure 4.6, we show the distribution of the $b_{(1,j)}$ vectors for the first RSg in approach "a3".

In "a3", the point-to-point send/receive operations (*exchanges*) between processes at grid points $A_{(r,c)}$ and $A_{(c,r)}$ are initiated after SpMVM in order to start the orthogonalization computations. By distributing the communication load over all processes, the underlying interconnection network can be used more effectively. Lewis et al. [39, 38] propose another method called "Redistribution-Free Method" to eliminate the exchanges of method "a3" at the expense

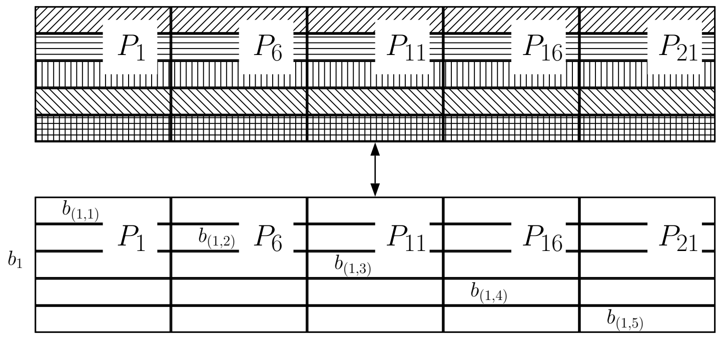


Figure 4.6: Approach “a3”: Example of block-row subdivision and reduction among processes of the first RSg.

of initial swaps between block subrows[3] . In our tests, performance results for this method were not much different than for “a3”. Hence, results for the “Redistribution-Free Method” are not included in our experimental results.

4.4 Network Load Model

Topology-aware mapping of processes to physical processors in large-scale computations can significantly reduce the communication overheads and improve overall execution time [31]. To analyze the data movement and communication overheads incurred by different mappings, we use the network load model for distributed memory SpMVM computations developed in [8]. We briefly summarize this model here for convenience, and evaluate the performance of the heuristic described in Section 4.3 in the next section, i.e., Section 4.5. Our evaluation includes the new network architecture called the Dragonfly topology (see Section 4.5.1), in addition to evaluation of generic distributed SpMVM algorithms on 3D Torus topology.

We model the communication graph of an SpMVM implementation by $G = (V_G, E_G)$. G is a directed graph, where the vertex set V_G is the set of processes and an edge $e = (u, v) \in E_G$ denotes a message sent from u to v. We define two communication graphs G_{col} and G_{row} associated with the CSg's and RSg's on a 2D (half) grid. Likewise, we model the physical interconnection network by a weighted, directed graph $\mathcal{H} = (V_\mathcal{H}, E_\mathcal{H}, c_\mathcal{H})$. The vertex set $V_\mathcal{H}$ is the set of physical nodes, $E_\mathcal{H}$ is the set of links between nodes, and $c_\mathcal{H}(e)$ is the capacity of

[3]For most of the scientific applications this part is just changing the initial matrix generation process, so at zero expense overall.

the link $e = (u, v)$. We assume that the messages are routed between nodes constituting the shortest path. We denote such a path by $p(u, v)$, the set of links connecting nodes u and v. Let $\mathcal{P}(u, v)$ be the set of all shortest paths between u and v. We assume that each path in $\mathcal{P}(u, v)$ is used with equal probability for sending a message. We also assume static routing, i.e., the messages are not redirected when congestion is detected on certain parts of the network.

Under this model, the topology mapping is specified by a function $\Gamma : V_G \to V_{\mathcal{H}}$. Three quality measures can be defined for this mapping: (average) dilation $\mathcal{D}(\Gamma)$, (average) traffic $\mathcal{T}(\Gamma)$, and (maximum) congestion $\mathcal{X}(\Gamma)$. $\mathcal{D}(\Gamma) = (1/|E_G|) \sum_{uv \in E_G} |p(\Gamma(u), \Gamma(v))|$, which is the average number of links traveled by a message. Hence, it is a measure of the total communication work performed by the interconnection network. The total traffic crossing a link $e \in E_{\mathcal{H}}$ is

$$\mathcal{T}_\Gamma(e) = \sum_{uv \in E_G} (|\{p : \ p \in \mathcal{P}(\Gamma(u), \Gamma(v)) \wedge e \in p\}|)/(\mathcal{P}(\Gamma(u), \Gamma(v))),$$

and the average traffic is $\mathcal{T}(\Gamma) = (1/|E_{\mathcal{H}}|) \sum_{e \in E_{\mathcal{H}}} \mathcal{T}_\Gamma(e)$. The congestion of a link can be modeled by $\mathcal{X}(e) = \mathcal{T}_\Gamma(e)/c_{\mathcal{H}}(e)$ and the network congestion by $\mathcal{X}(\Gamma) = \max_{e \in E_{\mathcal{H}}} \mathcal{X}(e)$.

Since the communication graph G does not contain any time-stamp information, $\mathcal{X}(\Gamma)$ may not be a good approximation to the actual network congestion, if G represents a large time window. However, both G_{col} and G_{row} capture the communication happening over a small time window for a number of collective calls. Therefore, $\mathcal{X}(\Gamma)$ is a good approximation to the network congestion during the column and row communications of the LA described in Section 2.2.

4.5 Experimental Results

This section reports on the performance analysis results obtained for approaches described in Section 4.3. We use the LA (section 2.2) as a proxy to an *ab initio* nuclear physics application. We used CSB_COO sparse matrix storage format (Section 2.3.2) in all methods for improved performance.

4.5.1 Description of the Computing Platforms

We performed all our experiments on Hopper, a Cray XE6 system, and Edison, a Cray XC30 system, at the NERSC.

Hopper - Cray XE6:

Each compute node of Hopper has two 12-core AMD 'MagnyCours' at 2.1 GHz processors (i.e., 24 cores per node) with 32 GBs of main memory. Each processor is composed of two sockets. Although the entire 32 GBs of memory is available to all cores on a node, it is physically partitioned into four memory banks. There are three levels of cache available per core. Each core has its own L1 and L2 caches with sizes 64KB and 512KB, respectively. One 6MB L3 cache is shared between 6 cores. It is possible to run a single MPI process on a Hopper node with 24 OpenMP threads. However, in this case, the overall performance is significantly degraded due to the NUMA issues. Based on the conclusions of Srinivasa and Sosonkina [61], we use one MPI process per NUMA domain with six OpenMP threads.

Hopper has a Cray Gemini based interconnection network that has a 3D Torus topology of dimensions $17\times8\times24$. Two Hopper nodes are connected by a single Gemini Application-Specific Integrated Circuit (ASIC) with two network interface controllers (NICs) [46]. Each NIC has two links in $+X$, $-X$, $+Z$, $-Z$, but one link in $+Y$ and $-Y$ directions [46]. The job scheduler on this machine ensures that physically nearby processing units are ranked consecutively [9].

Edison - Cray XC30:

Each compute node of Edison has two 12-core Intel 'Ivy Bridge' processors at 2.4 GHz (i.e., 24 cores per node) with 64 GBs of main memory. Each node is composed of two sockets and 32 GBs of memory per socket. Similar to Hopper's compute node, Edison compute node has also three levels of cache. First two are per core with sizes 64KB (L1) and 256KB (L2), and a 30MB L3 cache is shared between 12 cores. To match Hopper's setup, we use four MPI processes with six OpenMP threads on an Edison node.

Edison employs the Dragonfly [33, 24] topology for its interconnection network. Dragonfly topology is a hierarchical network with 3 different layers. On Edison, top level groups consist of two cabinets. Top level groups of locally interconnected routers are connected to other groups in an all-to-all manner using high speed global links (Rank3 network). This way, communications between groups are routed through a single global link only. Each top-level group contains two separate networks in it (Rank2 and Rank1 networks). Every cabinet on Edison has 3 chassis, so there are a total of 6 chassis in a Rank2 network. Rank2 network connects a compute blade in a chassis to the compute blades with identical slot numbers on the other 5 chassis in the group. Rank1 network is an all-to-all network of compute blades within a chassis. Rank1 network uses inter-Aries connections in the backplane. Every chassis deploys 16 compute blades (or 64 compute nodes). The adaptive routing deployed in this topology selects between minimal and non-minimal routes based on the load of the network. In a minimal route, there are only two hops between any two nodes in a group, whereas up to four hops might be required in case of a non-minimal route. A minimal route for inter-group communications uses only a single Rank3 link, however, 2 such links are needed in case of a non-minimal route.

Network model application:

If the physical coordinates of two processing units are known then we can compute the distance between two processes mapped onto them. Cray Linux Environment [21, 22] provides `xtprocadmin` and `xtdb2proc` utilities to obtain this information. The collective operations MPI_BCAST and MPI_REDUCE in the Cray MPI library of both Hopper and Edison are implemented using a binomial tree algorithm [63]. The construction of $E_{G_{row}}$ and $E_{G_{col}}$ is done by identifying the binomial trees associated with the row and column communication groups of the grid. In our case, four MPI processes would be mapped to a given physical node on both Hopper and Edison. Thus, Γ is a many-to-one function. In our model, the edges corresponding to intranode communications are not included because intranode communications do not incur load on the physical network. Edison's adaptive routing policies may result in non-minimal routes in case of congestion, but we still evaluate our network load model assuming minimal routes. So our load estimates can be described as (tight) lower bounds for the actual load on the network.

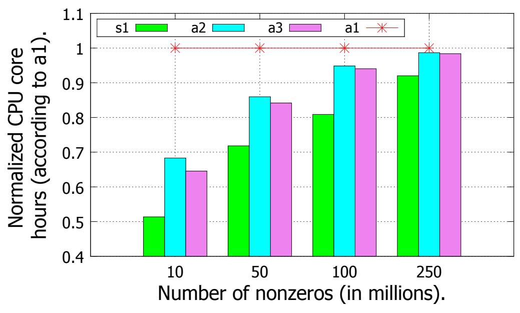
4.5.2 Analysis of the Obtained Results

Different test cases presented in this section are denoted by the number of diagonal processes used in the computation, i.e., n_d. We performed test runs using three different n_d values, i.e., 23, 45, and 65. The number of MPI processes used by our test runs is 529, 2025, and 4225 for the "a1/a2/a3" approaches, and 276, 1035, and 2145 for the symmetric approach "s1". The actual number of cores used in our test runs is 6 times the number of MPI processes due to MPI/OpenMP hybrid parallelization. In all runs, 100 Lanczos iterations were performed and performance results for a particular n_d is averaged over three runs. Also, the specific values are the averages of times among all the participating MPI processes. We implemented the distributed memory SpMVM approaches in Fortran using double-precision arithmetic. In all tests, the dimensions of the square submatrix on each MPI process is $1,600,000$. We used sparse matrices of varying degrees of nonzero densities to analyze the performance under different computation/communication loads. Local number of nonzeros are 10, 50, 100, and 250 million. Row and column indices of the nonzeros in local matrices are generated randomly on each MPI process[3] .

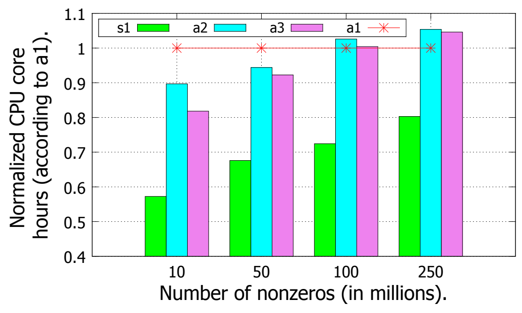
4.5.2.1 "CPU core hours" comparison:

Figure 4.7 shows the "CPU core hours" consumed by different distributed memory SpMVM approaches. "CPU core hours" are computed based on the total execution time for 100 Lanczos iterations (*tlanc*) and the number of CPU cores used by the specific approach. Results are normalized by the execution time of our base case implementation "a1".

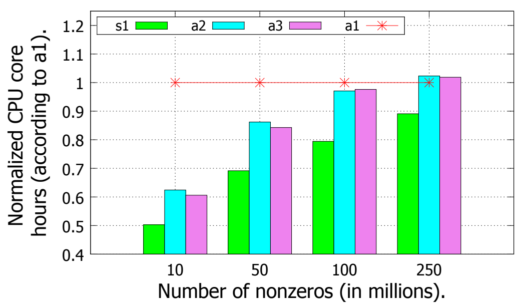
[3] `http://www.cs.ucdavis.edu/~bai/NEP/mvm/matran.f`



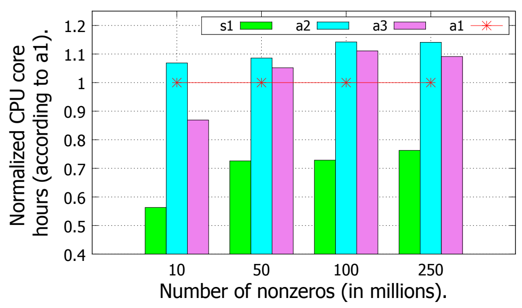
(a) Hopper run with $n_d = 23$.



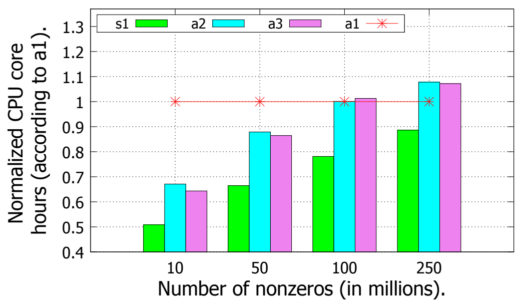
(b) Edison run with $n_d = 23$.



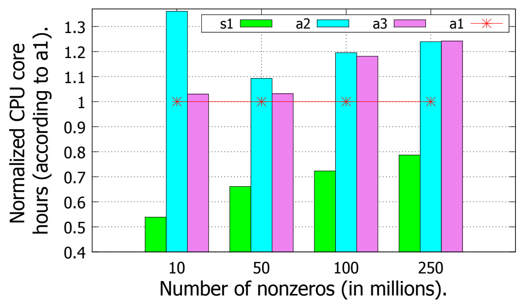
(c) Hopper run with $n_d = 45$.



(d) Edison run with $n_d = 45$.



(e) Hopper run with $n_d = 65$.



(f) Edison run with $n_d = 65$.

Figure 4.7: CPU core hours used by the tested distributed SpMVM implementations on Hopper and Edison. Results are normalized by the CPU core hours consumed in the base case implementation "a1".

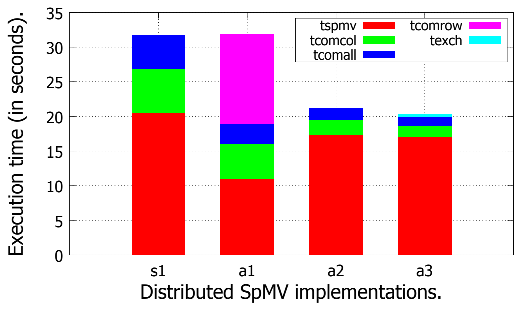
Although the execution time *tlanc* for “s1” is the highest, Figure 4.8, among all implementations, this method exploits the matrix symmetry and, thus, uses about half the number of cores used by other approaches (for n_d diagonal processes, “s1” uses $n_d * (n_d + 1)/2$, while other approaches use n_d^2). As Figure 4.7 shows, the total “CPU core hours” consumed by “s1” is substantially lower. Note that the computational complexity and communication patterns of the orthogonalization part is the same for all approaches. Therefore, gains in “s1” are due to its better performance in distributed-memory SpMVM computations.

On Hopper, overlapping communication with computation yields significant impact in the “a2” and “a3” implementations for small number of nonzeros (10 and 50), with “a3” generally being more efficient than “a2” independent of the sparsity. However, with large number of nonzeros (100 and 250 millions), gains are either small or non-existent. This can be explained by two factors. First, when overlapping communication with computation, a thread is designated as the communication thread and so there are only 5 threads instead of 6 threads for computations (i.e., a reduction of 16.6%). As the number of nonzeros increases, the load on computation threads also increases, but communication overheads decrease relatively. Secondly, as the number of processes (and hence n_d) increases, reduction of the result vectors requires more phases. Several small MPI reductions and OpenMP synchronizations in between phases adversely impact the performance.

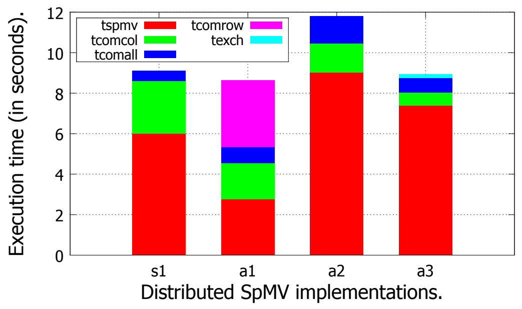
On Edison, the overall execution time is significantly improved over Hopper (up to 4×, see Figure 4.8). The “CPU core hour” cost of “s1” relative to “a1” is also improved over Hopper results for large number of nonzeros (100 and 250 million nonzero cases in Figure 4.7). However, overlapping communication with computation in “a2” and “a3” tends to improve the performance only with the smallest number of nonzeros for $n_d = 23$ and $n_d = 45$.

We note that the Dragonfly topology of Edison represents a major change over Hopper’s 3D Torus interconnect, and we used the PGI compiler on Hopper, which produces well-optimized executables for Fortran codes. On the other hand on Edison PGI compiler was not available and we used the GNU compiler. The difference in interconnect technology, compiler, L3 and L2 cache sizes, and the fact that we had exclusive L3 cache per MPI process on Hopper and not on Edison might have contributed to the poor performance of “a2” and “a3” on Edison.

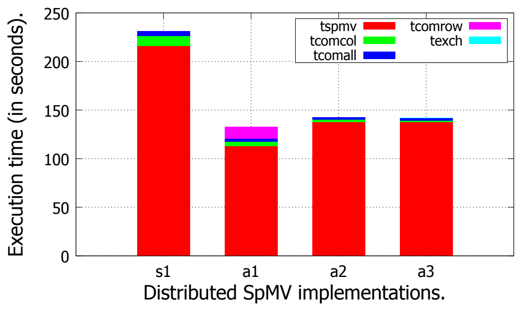
4.5.2.2 Execution time comparisons within Lanczos iteration:



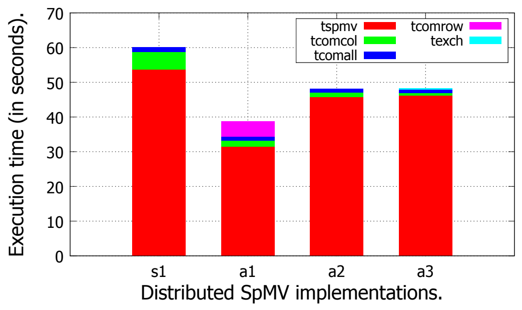
(a) Hopper run with $n_d = 65$ and $nnz = 10$ (million nonzeros per process (submatrix)).



(b) Edison run with $n_d = 65$ and $nnz = 10$ (million nonzeros per process (submatrix)).



(c) Hopper run with $n_d = 65$ and $nnz = 250$ (million nonzeros per process (submatrix)).



(d) Edison run with $n_d = 65$ and $nnz = 250$ (million nonzeros per process (submatrix)).

Figure 4.8: The *tspmv* is the execution time of the SpMVM process, *tcomcol* is the execution time of the communication along the column communicators, *tcomall* is the execution time of the MPI calls to MPI_ALLREDUCE inside the orthogonalization process, *tcomrow* is the execution time of the communication along the row communicators, and *texch* is the time spent for exchanging values between off-diagonal processes (this is only applicable to "a3"). Shown values are averages among all processes.

Figure 4.8 shows the breakdown of total execution times (*tlanc*) into computation time (*tspmv*) and various communication times. Timings in our largest runs ($n_d = 65$) for the matrix with lowest sparsity[3] (10 million nonzeros) and the one with the highest sparsity (250 million nonzeros) is shown. From Figure 4.8, one can see that the bulk of the *tlanc* time is spent performing SpMVM. Note that for "s1" SpMVM time consists of an SpMVM and an SpMVM$^\mathsf{T}$. Timings of overlapped reductions over RSg's are included in *tspmv* times for "s1", "a2", and "a3". Method "a1" spends a considerable amount of time for row communication compared to its SpMVM time for smaller number of nonzeros (Figure 4.8a and 4.8b), which is

[3]Here, we define sparsity as number of nonzero matrix elements divided by square of matrix dimension.

not the case when the number of nonzeros is increased to 250 (Figure 4.8c and 4.8d). Also, for both "a2" and "a3", the SpMVM times are larger than those for "a1", most likely because in both cases timings for communication along the RSg is included.

In Figure 4.8, the *tcomcol* time for "s1" is significantly larger than that of "a1/a2/a3" methods. This is because "s1" needs to perform two collective communications over CSg's, whereas other methods need only a single communication over CSg's. These collective communication operations are tree algorithms, so they have a time complexity of order $\log(p)$ for p processes. Methods "a1/a2/a3" have n_d processes in each group, and the time required for *tcomcol* is $\log(n_d)$. However, "s1" has $(n_d+1)/2$ processes in each group, so the total time for *tcomcol* in this case is $2\log((n_d+1)/2) = 2\log(n_d+1)-2$. This difference explains the increased *tcomcol* time for "s1".

4.5.2.3 Evaluation of the network load model:

We outlined the network load model, in section 4.4, originally developed in [9], that was extended to analyze the communication overhead of the SpMVM implementations. In Tables 4.1 and 4.2, the average dilation $\mathcal{D}$, average traffic $\mathcal{T}$, and maximum congestion $\mathcal{X}$ estimates are shown for the communication groups CSg and RSg on Hopper and Edison, respectively. As discussed above, "s1" needs to perform two communications over CSg and RSg groups. Therefore, we report double the traffic values $\mathcal{T}$ from our analysis of the "s1" method. Note that communicating twice does not affect the average dilation $\mathcal{D}$ or congestion $\mathcal{X}$ (because the two communications are happening at distinct times).

Table 4.1: Communication analysis for distributed SpMVM implementations on Hopper.

		"s1"		"a{1/2/3}"	
# of diagonals	Stats	CSg	RSg	CSg	RSg
23	$\{\mathcal{D},\mathcal{T},\mathcal{X}\}$	{0.4, 6, 6}	{4.5, 10, 30}	{0.6, 3, 8}	{5.7, 9, 74}
45	$\{\mathcal{D},\mathcal{T},\mathcal{X}\}$	{0.5, 6, 12}	{4.2, 12, 30}	{0.5, 3, 10}	{5.1, 14, 112}
65	$\{\mathcal{D},\mathcal{T},\mathcal{X}\}$	{0.5, 6, 10}	{5.2, 12, 40}	{0.6, 3, 12}	{7.2, 14, 82}

Table 4.2: Communication analysis for distributed SpMVM implementations on Edison.

		"s1"		" a{1/2/3}"	
# of diagonals	Stats	CSg	RSg	CSg	RSg
23	$\{\mathcal{D}, \mathcal{T}, \mathcal{X}\}$	$\{0.2, 3, 2\}$	$\{1.5, 30, 22\}$	$\{0.2, 0.5, 0\}$	$\{1.6, 16, 24\}$
45	$\{\mathcal{D}, \mathcal{T}, \mathcal{X}\}$	$\{0.5, 3, 4\}$	$\{2.7, 7, 28\}$	$\{0.4, 1.4, 5\}$	$\{2.8, 10, 84\}$
65	$\{\mathcal{D}, \mathcal{T}, \mathcal{X}\}$	$\{0.3, 3, 2\}$	$\{2.0, 12, 55\}$	$\{0.3, 1.5, 4\}$	$\{2.6, 20, 92\}$

Communication links on Hopper constitute a (relatively) homogenous 3D Torus network. However, Edison's hierarchical interconnection network is composed of three separate networks (Rank1, Rank2 and Rank3) with very different characteristics. The point-to-point communication benchmarks on Edison yield similar latencies and bandwidths for all 3 network types [14]. However, the Rank3 network is 2-3× slower than Rank2 and Rank1 networks under multi-point communication tests [14]. Therefore, we use the $\mathcal{T}$ and $\mathcal{X}$ estimates from the Rank3 network in our evaluations, while treating the links in all three networks the same to compute $\mathcal{D}$.

Lower values of $\{\mathcal{D}, \mathcal{T}, \mathcal{X}\}$ indicate a lower load on the network and therefore reduced communication overheads would be expected in actual computations. For approaches "s1" and "a{1/2/3}", the network load on CSg's are estimated to be significantly less than those of RSg's according to our network load model, as shown in Tables 4.1 and 4.2. This estimation is confirmed by the actual communication times observed in our tests on both Hopper and Edison (see the *tcomcol* and *tcomrow* timings in Figure 4.8 as an example). Our analyses show that placing consecutively ranked processes into the same CSg's indeed leads to a topology optimized communication groups on the 3D Torus interconnect of Hopper, as well as the Dragonfly interconnect of Edison.

To compare the data movement incurred by approaches "s1", "a1", "a2", and "a3", we analyze the total distance (number of hops) traveled by all messages in each computation. On Edison, we estimate that in a single iteration of the n_d = 23, 45, 65 test cases, the total distance for messages in "a1/a2/a3" is about 942, 6433, and 12175, respectively. In comparison, for the "s1" method our estimates are about 868, 6284, and 9556, indicating data movement reductions of 8%, 3%, and 21%, respectively. So in addition to lowering the cost of "CPU core hour" usage, the recently proposed "s1" method can be effective in reducing the total data movement, too.

4.6 Conclusions

We presented performance analysis of efficient parallel SpMVM approaches "s1", "a1", "a2", and "a3" presented in literature on distributed memory multi-core architectures. We analyzed the performance of these implementations on large scale sparse matrices using the LA. When the same number of diagonals, n_d, is used the time to completion with approaches "a1", "a2", and "a3" is typically lower than "s1". However, in terms of "CPU core hours" consumed, in all examined cases "s1" is more efficient than "a1", "a2", and "a3". In particular, the combination of a unique half 2D processor grid, communication hiding techniques, and topology aware mapping heuristics used in "s1" results in highly efficient distributed symmetric SpMVM computations. The approaches "a1", "a2", and "a3" do not make use of the underlying symmetry of the sparse matrix, and thus they consume more memory as well as CPU resources overall.

We observed that the communication and computation overlapping technique that we have implemented in "a2" (based on the idea provided in [38]) outperforms the basic approach "a1" [39] in the sparser cases on Hopper, but generally not on Edison. We also observed that the optimization suggested in approach "a3" [38] is generally more efficient than "a2". Furthermore, on Edison "a3" is competitive with or better than "a1" for the sparser cases, in contrast to "a2". This suggests that the benefits of overlapping communication and computation depends on several factors such as the sparsity (amount of computation vs. communication), system architecture, and type of interconnect.

Finally, we outline a simple network model to define measures such as the dilation, network traffic, and congestion. We have extended this model for the analysis of algorithms discussed in this chapter and additionally for the Dragonfly network. The lower estimate for total number of hops traveled by messages in "s1" indicate that this approach reduce data movement overheads compared to traditional approaches. The reduced memory footprint and low data movement overhead properties possessed by "s1" are especially important as we move towards the exascale era where these considerations become of paramount importance. These results hold for both interconnect topologies tested, i.e., 3D Torus and Dragonfly, which are very different in their designs, but both use the same job scheduler which places consecutive MPI ranks close to each other on the hardware [9]. The approach "s1" has been adopted in the solution of extremely large-scale eigenvalue problems for nuclear structure computations [41] and has been shown to achieve good scalability [9].

CHAPTER 5. TOPOLOGY-AWARE MAPPING FOR A LARGE-SCALE EIGENSOLVER ON A TORUS NETWORK

In this chapter we present new mappings and application setups that improve the performance of the parallel SpMVM during the parallel LA part of MFDn compared to the default settings. Specifically, section 5.1 introduces the motivation behind our work in this chapter, and gives literature survey. In section 5.2 we give information on the supercomputer considered and present the mappings that we propose. We apply these mappings and obtain empirical results, which are discussed in section 5.3 and further analyzed using the network model presented in section 5.4. Finally, the section 5.5 concludes the overall contributions.

5.1 Introduction

Large-scale real world scientific applications typically need supercomputers to perform their simulations, not only because of the necessary compute-power for large-scale calculations, but also because of the aggregate memory needed for the simulations. In order to process them one needs to store the data in RAM, which requires the use of a large number of nodes. Scaling such applications to larger production runs increases the number of nodes, which increases the importance of communication among tasks located on different nodes. According to the latest TOP 500 [3] listing, November 2015, top ranked supercomputers feature millions of cores with more than ten thousand nodes. This steep rise in the number of nodes requires a careful analysis of the communication pattern of specific applications and the topology of the underlying interconnect. Assigning tasks to nodes without taking into account the communication pattern and the topology of the interconnect often produces inefficient execution. In order to achieve a better performance one has to carefully map the tasks to the nodes, considering the load balance of the data distribution and necessary communication between nodes. Task mapping is the process of assigning application tasks to the nodes of a parallel computer so that it reduces the communication overhead and subsequently the execution time. The general task mapping is an NP-hard problem [18].

In this chapter we investigate the communication patterns during the parallel LA process of MFDn and tune the performance on Mira, an IBM BG/Q, supercomputer located at ANL. As it was previously mentioned in Section 2.4, MFDn currently uses LBCM ordering heuristic for mapping MPI tasks for a non-contiguously, sparse, allocated space [8, 9]. However there has not been any optimization of the task mapping for MFDn on a system with contiguous block-based node allocation scheme, where one knows ahead of time exactly how the nodes are connected; nor has any effort been made to exploit the torus based interconnects. Top-performing supercomputers frequently use the torus networks. Four of the top five supercomputers according to the 2015 TOP 500 use torus based interconnects, and two of them are IBM BG/Q machines. Also many Cray based supercomputers are using a torus interconnect, e.g., Cray XK7 machines such as Titan at ORNL. In Japan, the K computer uses the Tofu [6] interconnect which is a 6D mesh/torus. Additionally, eight of top ten supercomputers according to the latest listing of GRAPH 500 [4], November 2015, are also based on IBM BG/Q machines. Therefore, in this chapter, we focus on mapping the tasks of the large-scale eigensolver on contiguous blocks of the 5D Torus network used by IBM BG/Q supercomputers.

Aktulga et al. [8, 9] proposed a heuristic for mapping tasks on sparse allocations for MFDn. For those test cases the MFDn application was run on Hopper, a Cray XE6 supercomputer, as well as on Edison, a Cray XC30 supercomputer, both at NERSC. The main idea for selection of LBCM mapping is explained in Section 2.4, and it seems to work reasonably efficient for moderately large jobs, as also discussed in the Chapter 4. This heuristic is currently also used by MFDn for (extremely) large production runs on IBM BG/Q block-based partitions. Here, we propose new heuristics for efficient mapping for large-scale production runs with MFDn on these block-based partitions, which outperform the currently used default mapping.

Bhatele et al. [16, 17] developed a tool to generate mappings on torus network for applications. While using the tool one can generate a large set of mappings, it does not necessarily generate a mapping that will perform efficiently, and hence it requires a human expert to specify and evaluate the large number of mappings it generates. In fact, the mapping that performs well for one core count can perform poorly on another core count of the same application as it was shown in their study. They further automated the process [29] without the need for human experts. However, they focus on point-to-point communication, and do not examine the collective communications. The mappings we propose improve the performance for all partition sizes when placing a single MPI rank per node. However, for some partition sizes we actually get even better performance by placing two MPI ranks per node, although for those cases the *Default* mapping typically performs better than our proposed mappings.

As the problem size gets larger, the time spent on communication increases. This is because the computational load per node stays approximately fixed as the problem size increases, but the communication load increases both with the message size (length of the vectors) and with the number of MPI ranks. Therefore, our main focus is on larger runs, however we also consider relatively small runs (which go as small as 512 nodes) for completeness. Note however, that in terms of problem size (total number of nonzero matrix elements), the smallest runs considered in this chapter is approximately the same size as the largest runs considered in Chapter 4. All of the runs performed in this chapter were done as part of physics production runs and the core count for each of the problem size is the minimum required, i.e., we cannot place the same problem size on a smaller partition.

The main contributions of this chapter are as follows:

1. We present task mapping techniques that reduce the communication time and hence the execution time of a single LA iteration by up to 39%.

2. We describe a network model with various metrics for evaluating mappings on an IBM BG/Q 5D Torus interconnect.

3. We further suggest a change in MPI calls during LA process, which can produce additional reduction of up to 10% in execution time, depending on both the message size and the partition size.

4. Significant work was done previously for MFDn to overlap more than half of the communication with computation. However, as both the problem size and the number of MPI ranks increase, the communication load increases but the computational load per node remains approximately constant. Hence it is inevitable that at some point the communication time will dominate the execution time. Our study suggests that for the most sparse matrices, with less than one in a million matrix elements being nonzero, we can no longer achieve complete overlapping of the RSg communication, and the row communication time can become a significant part of the total execution time. With a careful mapping strategy we can however hide most of this communication time, which emphasizes the importance of the studying task mappings and opens up new questions and challenges for future work.

5.2 Mappings proposed

In this section we give information on the IBM BG/Q supercomputer, and on the proposed mappings. We propose mappings that reduce the communication time of the LA and can potentially reduce the communication time for other Lanczos type iterative algorithms on a torus interconnect with contiguous block-based allocations for submitted jobs.

5.2.1 Mira, an IBM BG/Q, supercomputer

In this study we run all of our test cases on Mira, an IBM BG/Q, supercomputer installed at ANL. (The only exception is the 2 rack cases of dataset #1, see section 5.3.1. These 2 rack cases were run on Cetus, an IBM BG/Q, test bed machine with total of 4 racks, also, installed at ANL. Specifically, this is the first 2 rack dimension listed in Table 5.1.) The Mira supercomputer has a total of 48 racks, where each rack contains 1024 compute nodes. One rack contains two midplanes, each with 512 nodes, and each midplane consists of 16 nodeboards, each containing 32 nodes. A single node of Mira contains 16GB of RAM and 17 PowerPC A2 cores. The 17th core is reserved by the system, so the applications can use up to 16 cores, where each core is capable of running up to four hardware threads. Nodes of Mira are connected via 5D Torus interconnect. The source code used, for MFDn, is written in Fortran (MPI+OpenMP), and is complied with "mpixlf90_r" compiler and "-g -O3" options set.

Table 5.1: Geometry of the partitions allocated for test cases on Mira, IBM BG/Q, for various node counts. Except for first 2 rack geometry, which is for Cetus, IBM BG/Q.

Racks	**Nodes**	**A**	**B**	**C**	**D**	**E**
.5	512	4	4	4	4	2
01	1024	4	4	4	8	2
02	2048	4	4	8	8	2
02	2048	4	4	4	16	2
04	4096	8	4	4	16	2
08	8192	4	4	16	16	2
12	12288	8	4	12	16	2
16	16384	4	8	16	16	2
24	24576	4	12	16	16	2
32	32768	8	8	16	16	2
48	49152	8	12	16	16	2

Each job allocated on Mira gets a block-based contiguous partition. The location of MPI process on the torus network is described using ABCDET coordinates, where the first five coordinates correspond to the five dimensions of the torus interconnect and the last coordinate T specifies the number of MPI processes sharing a single node. For example, for the first set of test runs, see section 5.3.1, we have one MPI process per node and 64 threads per MPI process, i.e., T = 1. The size of the dimension E is always 2, and all other dimensions have size at least 4, so every node is directly connected to each of its nine (+/- direction for each dimension) neighbors by a network link. Once the job starts each MPI process is assigned a ABCDET coordinate on a 5D torus. The default mapping in BG/Q is the assignment of MPI processes to nodes of the system in ABCDET order where the coordinates increment from the rightmost dimension to the leftmost. In other words, first the dimension T increments until the maximum value is reached, then the dimension E increments, and so on. One can permute the letters of the geometry to specify different mappings. In case the mapping cannot be obtained with just permuting, one can provide a file, which should contain a coordinate for each of the MPI ranks in a separate line of the file. Each coordinate in this file should consist of six numbers separated by space, e.g., 0 0 0 1 1 0. For a sample file see Appendix A.2. Both of these ways, either a mapping obtained by permuting or through a file can be specified with the command line option, "RUNJOB_MAPPING", as part of "qsub" statement, e.g., "RUNJOB_MAPPING=DCBAET" or "RUNJOB_MAPPING=data.txt". All of the geometries of the partitions used for our test cases are listed in Table 5.1, and all of them have a torus connectivity in all dimensions.

5.2.2 Heuristics for task mapping

Recall that the are n_d CSg's each containing $(n_d + 1)/2 + n_{extra}$ MPI processes and n_d RSg's each containing $(n_d + 1)/2$ MPI processes in MFDn as described in section 2.4. In order to reduce the communication load on network, we first focus on CSg, as the column communication is not overlapping with any computation, and can take a significant fraction of the overall run time of the LA. Our strategy is to place the MPI processes of each CSg on the torus network so that the nodes under the mapping are placed as close as possible within each CSg in terms of maximum hop count.

We first propose *Base* mapping described in Algorithm 4. In the current geometries on Mira, the dimension E is always 2, the dimension D is the largest; whereas in the proposed *Base* mapping the geometry $\ddot{g}$ is ordered from largest to smallest and equal sized dimensions are ordered in descending order. Note that none of the geometries on Mira (including the ones listed in Table 5.1) is ordered this way.

Algorithm 4: *Base* mapping steps.

```
1 procedure Base ($g$);
  Input  : Geometry $g$ of the partition;
  Output: New geometry $\ddot{g}$;
2 $\ddot{g} = g$;
3 Permute the dimensions of $\ddot{g}$ highest to lowest;
4 Permute the equal sized dimensions of $\ddot{g}$ in descending order;
5 return $\ddot{g}$;
```

Notice that if there is a communication between every pair of tasks then the ordering from highest to lowest is the best among all permutations of dimensions. For example, consider the 12-rack partition with the geometry <8,4,12,16,2>. The size of each CSg in this case is 79. The *Default* mapping gives the largest hop count 13 among CSg members in the physical interconnection network, while the ordering we propose (i.e., *Base* mapping) reduces it to 9. Although there are pairs of tasks with no communication between them. This method is aimed at decreasing so-called maximum dilation metrics (defined later in section 5.4). The *Base* mapping is meant to collect all CSg ranks into compact subgroups as much as possible.

Algorithm 5: *BaseAlt* mapping.

```
procedure BaseAlt ($g$);
Input  : Geometry $g$ of the partition;
Output: List $l$ of torus locations;
$\ddot{g} = Base(g)$;
for $i_1 \leftarrow 0$ to $size(\ddot{g}_1) - 1$ do
    $var_1 = i_1$;
    for $i_2 \leftarrow 0$ to $size(\ddot{g}_2) - 1$ do
        if $i_1$ is even then
            $var_2 = i_2$;
        else
            $var_2 = size(\ddot{g}_2) - i_2 - 1$;
        end
        for $i_3 \leftarrow 0$ to $size(\ddot{g}_3) - 1$ do
            if $(i_1 * size(\ddot{g}_2) + i_2)$ is even then
                $var_3 = i_3$;
            else
                $var_3 = size(\ddot{g}_3) - i_3 - 1$;
            end
            Add $var_{1,2,3}$ to $l$;
        end
    end
end
return $l$;
```

We next modify the *Base* mapping by alternating the increments during index assignment between increasing and decreasing orders for each dimension based on even and odd iterations, respectively. This can further place the RSg ranks of *Base* mapping closer; we call this mapping *BaseAlt*. The steps of this mapping are described in Algorithm 5 for 3D torus network (to save space), this can easily be extended to any higher dimension of torus. A sample *BaseAlt* mapping file is shown in Appendix A.2.

Placing the MPI processes using the *BaseAlt* mapping further decreases the largest hop count among CSg members in the physical interconnection network. E.g., for the above example the largest hop count that any CSg could have will decrease further to 8 with *BaseAlt* mapping.

5.3 Experimental results

This section contains empirical results with detailed descriptions of each of the runs. We separate our runs into three datasets: first – for quick check (section 5.3.1); second – for more detailed/rigorous test (section 5.3.2); and, finally, last – for comparison and analysis of best setups with the original source code setup, which is presented in Table A.3.1. We start the section by describing the terminology and abbreviations[1] that will help us explain the further details of the experimental setup as shown in Table 5.2.

5.3.1 Results using dataset #1

Tables 5.3 and 5.4 list the runs of first dataset and contain information on the matrices used and application parameters, including the number of MPI processes used for each test case based on the number of racks. Notice that the LNNZ is approximately 1 billion for all of the cases, except for test runs 08a and 48a. For this dataset single MPI process is placed in one node, and each MPI node runs with 64 threads, i.e., M. $= c1$ and Nth. $= 64$. Letters "a" and "b" at the end of the run "Name", in these tables, are used to differentiate between different runs of the same rack size.

[1] In addition to these abbreviations we also give information on message size of data exchanged among the CSg members in Appendix A.1, which was not shown in this table since it is not used explicitly throughout the chapter.

Table 5.2: Notations used throughout the chapter.

Abbreviation	**Description**
Nucleus	Physics case, e.g., A10Z5N5(12) where A - total nucleon number, Z - proton number, N - neutron number, and finally Nmax (maximum number of total oscillator quanta allowed in the many-body basis space above the minimum for that nucleus) value is in parenthesis.
Dim.	Dimension of the large matrix that needs to be tridiagonalized.
NNZ	Total number of nonzero matrix elements.
LDim.	Maximum local dimension of the submatrix in any given MPI process.
LNNZ	Maximum number of nonzero matrix elements in any given MPI process.
Procs	Total number of MPI processes used.
Nth.	The number of threads used by a single MPI process. Maximum number of hardware threads per MPI process is 64.
M.	The mode of the current setup that it is running on defines how many MPI processes will be sharing a single node. In our case we have two cases either a single MPI process or two MPI processes will be running on a single node, represented by $c1$ and $c2$, respectively.
S.	Sparseness of the local submatrix, i.e., $(\text{LNNZ}/\text{LDim.}^2) * 10^6$, in "parts-per-million".
R.msg	The size of the data (sub-vector), in MiB, in send buffer during the broadcast, MPI_BCAST, and reduce, MPI_REDUCE, MPI routines among the RSg members.

Table 5.3: Information on matrices of dataset #1.

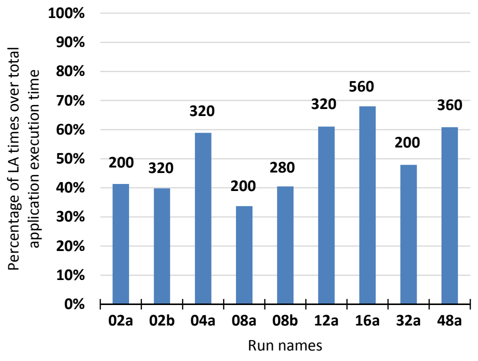
Name	**Nucleus**	**Dim.**	**NNZ**	**LDim.**	**LNNZ**	**S.**
02a	A06Z2N4(16)	595,922,646	1.97E+12	9,459,090	999,004,146	11.165
02b	A10Z5N5(10)	1,698,922,480	2.10E+12	26,967,023	1,055,417,797	1.451
04a	A09Z3N6(12)	2,614,910,212	4.30E+12	29,381,014	1,088,625,868	1.261
08a	A08Z2N6(14)	2,433,601,898	5.27E+12	19,162,220	657,303,251	1.790
08b	A09Z4N5(12)	4,232,122,420	7.49E+12	33,323,799	931,257,614	0.839
12a	A08Z3N5(14)	5,155,975,309	1.25E+13	33,264,357	1,043,829,669	0.943
16a	A08Z4N4(14)	6,770,401,490	1.71E+13	37,823,472	1,075,912,865	0.752
32a	A08Z2N6(16)	12,020,598,212	3.77E+13	47,512,246	1,184,591,877	0.525
48a	A09Z3N6(14)	16,469,097,856	3.98E+13	52,955,299	828,266,086	0.295

Table 5.4: Application parameters used for dataset #1.

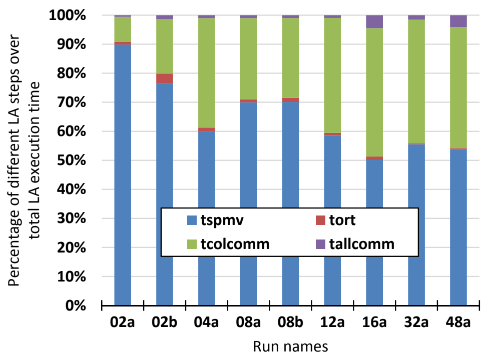
Name	**Procs.**	n_d	n_{extra}	**|RSg|**	**|CSg|**	**R.msg**
02a	2016	63	0	32	32	37
02b	2016	63	0	32	32	103
04a	4094	89	1	45	46	113
08a	8128	127	0	64	64	74
08b	8128	127	0	64	64	128
12a	12245	155	1	78	79	127
16a	16289	179	1	90	91	145
32a	32637	253	2	127	129	182
48a	49138	311	2	156	158	203

5.3.1.1 Baseline performance analysis

In order to establish the baseline performance, we run the application with the *Default*, ABCDET, mapping using up to 48 racks (full machine). The results are presented in Figure 5.1. The *tspmv* is the maximum execution time of the SpMVM process (this includes both SpMVM and SpMVM$^{\mathsf{T}}$ together with the overlapped row communication time), *tcolcomm* is the maximum execution time of the communication among the CSg ranks, which includes the communication during both the MPI_REDUCE_SCATTER and MPI_ALLGATHERV routines, *tort* and *tallcomm* are the average computation and communication times of the reorthogonalization parts, respectively.



(a) The numbers on bars indicate the number of iterations set or needed for convergence.



(b) Percentage breakdown of different LA steps.

Figure 5.1: Percentage breakdown of execution times of all dataset #1 runs with the *Default* mapping.

In Figure 5.1a we can see that the more the number of iterations in LA the more the percentage it takes of the overall application run time. Notice that, in Figure 5.1b, the column communication time, *tcolcomm*, grows with the matrix dimension and the number of nodes; while it is only 8% for the smallest test case, it increases up to 44% of the overall LA time for the largest runs. The communication during the reorthogonalization stage is on MPI_COMM_WORLD and it stays under 5% of the total LA execution time. This part occurs right after the MPI_REDUCE_SCATTER operation without synchronization at the starting stage, so it may include time from waiting on different CSg ranks to complete their MPI_REDUCE_SCATTER routines, therefore we take the average time for this part, whereas for all others we have taken the maximum values. Both *tort* and *tallcomm* parts together have very low percentage of the overall LA time, but increases with the number of iterations. Indeed the *tallcomm* is largest for the run with the most iterations, 560.

The row communication occurs within the computation intensive SpMVM procedure, which takes up a large percentage for smaller cases and less for larger ones. The row communication overlaps with the SpMVM computation, and is completely hidden in some cases and is not in others which we further investigate in section 5.3.2.

5.3.1.2 Empirical results and conclusions

In Table 5.5 we give the information on the number of diagonals, n_d, the number of extras per diagonal, n_{extra}, |CSg|, the geometry, and *Base* mapping for each of the runs of dataset #1. The last two columns of this table show for which mappings (if any of) the |CSg|'s evenly fit into the subtorus of partition, and whether or not it is equal to the subtorus. For example, the |CSg| = 16 (with M. = $c1$) evenly fits the following permutation of mappings <L,L,L,8,2>, <L,L,L,L,16>, <L,8,2,2,2> (where L is any of A, B, C, D, or E), i.e., any subtorus of multiple 16, whereas only first two are equal to the |CSg|. Note the following: Whenever *Base* mapping maps the CSg's evenly or exactly onto a subtorus, then so does the *BaseAlt* mapping, and whenever |CSg| is a power of 2, the CSg's can be evenly mapped on the underlying network, even if it does not actually fit a subtorus, except for 12, 24, and 48 racks.

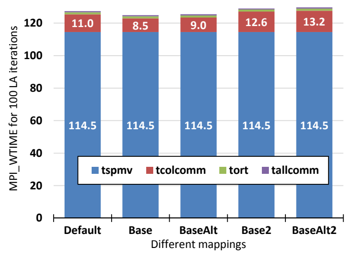
Table 5.5: Geometry and *Base* mapping information for dataset #1 runs.

Name	n_d, n_{extra}	|**CSg**|	**A,B,C,D,E,T**	***Base***	|**CSg**| maps |**subtorus**| evenly?	|**CSg**| equals |**subtorus**|?
02a/b	63,0	32	4,4,8,8,2,1	DCBAET	*Default* *Base*/*BaseAlt*	*Base*/*BaseAlt*
04a	89,1	46	8,4,4,16,2,1	DACBET	—	—
08a/b	127,0	64	4,4,16,16,2,1	DCBAET	*Default* *Base*/*BaseAlt*	—
12a	155,1	79	8,4,12,16,2,1	DCABET	—	—
16a	179,1	91	4,8,16,16,2,1	DCBAET	—	—
32a	253,2	129	8,8,16,16,2,1	DCBAET	—	—
48a	311,2	156	8,12,16,16,2,1	DCBAET	—	—

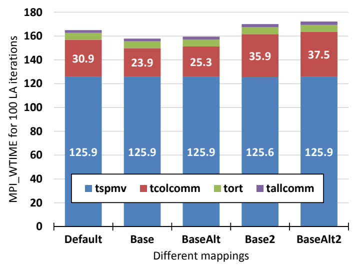
The detailed LA execution timings are presented in Figure 5.2. All of the subfigures have the maximum execution time in seconds (measured with MPI_WTIME) for *tspmv* and *tcolcomm*, and average execution times for *tort* and *tallcomm*. All results are normalized to 100 LA iterations for easy comparison.

For this set of tests both *Base* and *BaseAlt* mappings give better performance compared to the *Default* mapping, with *BaseAlt* performing better than *Base* for 6 of the 9 cases. The proposed mappings reduce the column communication for all test cases, with maximum reduction of 44%. Moreover, for most of the cases it can be seen that these mappings also reduce *tspmv*, which includes the row communication. That is, the proposed mappings reduce the row communication time, such that more of the row communication is overlapping with the (local) SpMVM and SpMVM$^{\mathsf{T}}$ computation as compared to the *Default* mapping. The only cases whose row communication increased with the *Base* mapping are 8 rack runs, i.e., 08*a* and 08*b*, however the *BaseAlt* mapping reduces it significantly for both runs.

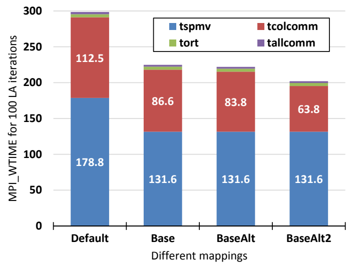
We also tried a change in MPI calls of LA stage, specifically for the column communication, which produces additional reduction of up to 10% in execution time. Recall from subsection 2.4 that the CSg communication consists of MPI_REDUCE_SCATTER and MPI_ALLGATHERV MPI routines. Using MPIX (MPI extensions) library provided for IBM BG/Q we have found that the MPI_ALLGATHERV operation is based on the optimal MPI_ALLREDUCE routine.



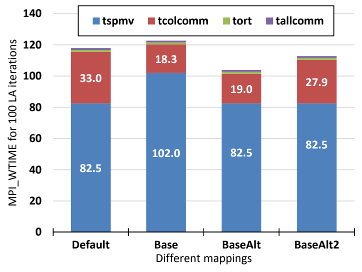
(a) 02a.



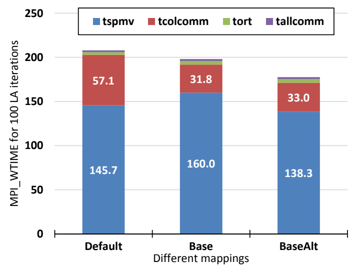
(b) 02b.



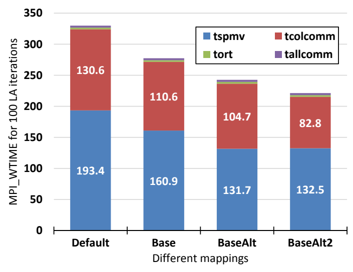
(c) 04a.



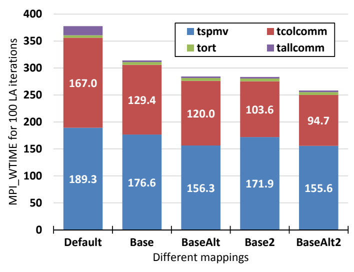
(d) 08a.



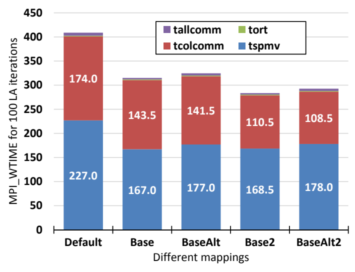
(e) 08b.



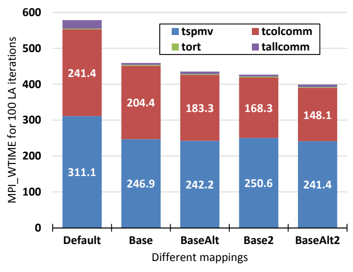
(f) 12a.



(g) 16a.



(h) 32a.



(i) 48a.

Figure 5.2: Detailed execution times for each test run of dataset #1. All timings are normalized to 100 LA iterations.

We have tested switching MPI_ALLGATHERV operation with MPI_ALLREDUCE and obtained better performance for some of the rack sizes, specifically for the larger ones. We call the versions with this change *Base2* and *BaseAlt2*, which were run with *Base* and *BaseAlt* mappings, respectively. The reason why this switch helps for some runs but does not for other runs needs further careful investigation, which is outside the scope of this work.

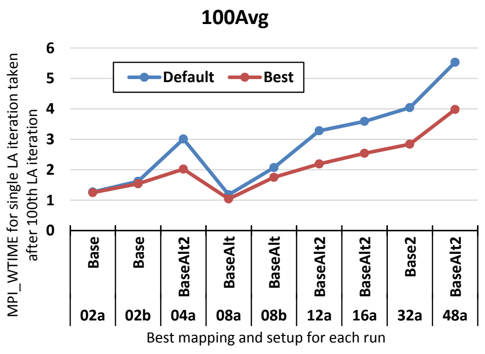
In all of our tests this switch was implemented as a separate case, i.e., hardcoded, but it can easily be incorporated dynamically as follows: Since we have several hundreds of iterations in LA, we can devote several iterations to be performed by each of the MPI routines of interest and then use MPI_REDUCE routine to settle on one of them. Here, we did this MPI routine swap only for the MPI_ALLGATHERV part, in our next more detailed dataset run in section 5.3.2 we also explore a similar alternative for MPI_REDUCE_SCATTER, and time the two column communications separately.

Table 5.6: Gain in percentage from the mappings proposed and call switches applied compared to the *Default* mapping, which uses MPI_REDUCE_SCATTER and MPI_ALLGATHERV. In every column bold numbers indicate the maximum percentage achieved by proposed mappings only.

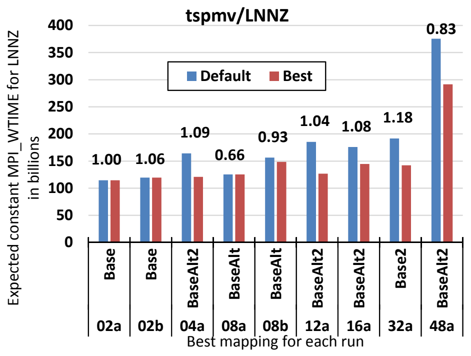
Name	**Mapping**	**tspmv**	**tcolcomm**	**100Avg**
02*a*	Base	0	23	2
02*b*	Base	0	23	5
04*a*	BaseAlt	26	26	**28**
04*a*	BaseAlt2	26	43	33
08*a*	BaseAlt	0	**42**	12
08*b*	BaseAlt	5	**42**	15
12*a*	BaseAlt	**32**	20	27
12*a*	BaseAlt2	32	37	33
16*a*	BaseAlt	17	28	22
16*a*	BaseAlt2	18	43	29
32*a*	Base	26	18	23
32*a*	Base2	26	36	30
48*a*	BaseAlt	22	24	22
48*a*	BaseAlt2	22	39	28

Table 5.6 summarizes the percentage gains obtained for different stages of the LA. The last column of this table shows the percentage gain obtained per single LA iteration time taken after the 100th LA iteration, *100Avg*. We were able to obtain up to 42% gain in *tcolcomm*

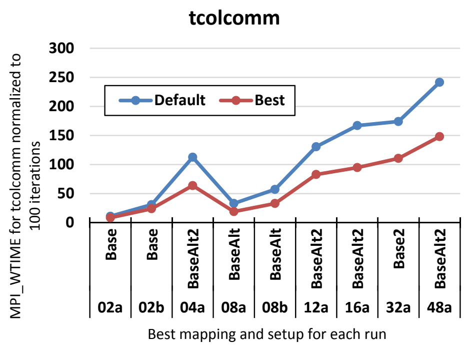
time with new mappings only. On top of the performance improvements obtained with new mappings in some cases we also obtained around 10% gain when the MPI call switch is applied. In general, we conclude that for 02*a*, 02*b*, and 32*a* runs we should use the *Base* mapping, and for all other cases *BaseAlt* mapping. Furthermore, we can also use the MPI_ALLREDUCE routine instead of MPI_ALLGATHERV. Overall, the proposed *BaseAlt* mapping improves the run time in all cases and is best in most of the cases.



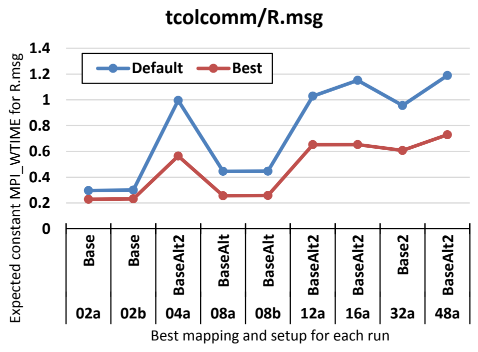
(a) Results for 100*Avg* times.



(b) Results for *tspmv*/LNNZ (in billions).



(c) Results for *tcolcomm* times.



(d) Results for *tcolcomm*/R.msg size (in MiB).

Figure 5.3: Comparison of *Default* mapping and the best mapping combination for dataset #1 runs.

Figure 5.3a summarizes the *100Avg* timings for *Default* and best mapping (combined with MPI routine swap). In this figure we can see the dips in the cases of 2 and 8 rack runs. For first test of 8 rack run it is more prevalent than the second one. We also provide two more Figures 5.3c and 5.3d, which show *tcolcomm* and *tcolcomm*/R.msg values, respectively. These figures taken together clearly show the dips for these two partition sizes. Furthermore, Figure 5.3d suggests weak scaling of the column communication, but at two levels: with the optimal mapping $tcolcomm$/R.msg ≈ 0.2 for 2 and 8 racks, and $tcolcomm$/R.msg ≈ 0.6 for 4, 12, 16, and 32 racks. This weak scaling breaks down for 48 racks. Recall that for 2 and 8 racks, but not for the other cases in this data set, the |CSg| is power of 2, i.e., it is 32 for 2 rack runs and 64 for 8 rack runs, see Tables 5.4 and 5.5. As shown in Table 5.5, in both of these rack sizes we have |CSg| mapping the subtorus evenly. This interesting observation led us to exploit the topology explicitly, using |CSg| sizes that are powers of 2 so it maps the network, which set the ground for our next set of runs that we perform in section 5.3.2 in much more details.

Finally, in Figure 5.3b we show the results of *tspmv* (which is the maximum of 100 iterations) divided by LNNZ in billions, this gives us the approximate time of the *tspmv* proportional to LNNZ. The numbers above the bars show the LNNZ in billions. According to our separate measures of only pure SpMVM+SpMVM$^{\mathsf{T}}$ (i.e., without row communication) time for 02*a* and 02*b* have shown around 114.2 and 124.34 seconds, respectively. The $tspmv/LNNZ$ for these two cases are 114.3426042 and 117.8119006, respectively. From these results in general we can conclude that the best mapping combination/setup shows almost complete overlapping of the row communication with the local SpMVM + SpMVM$^{\mathsf{T}}$, except for the largest case, 48 racks. In an ideal situation with completely overlapping row communication *tspmv* would be proportional to LNNZ, and thus one would expect to have about the same bar (i.e., for these runs it should be somewhere between 114.3426042 and 117.8119006) length for all of the cases, possibly with some minor fluctuations due to different cache performance depending on the sparsity. Clearly, the row communication is not fully overlapped with the *Default* mapping starting at 4 racks. Thus, this figure shows that the proposed mappings also reduce the row communication, which eventually helps hiding it behind the computation. The Figure 5.3b gives also information on weak scalability of the application's LA part, from these results we can clearly see that the best mapping combinations try harder to straighten the overall execution in order to achieve better weak scalability. Notice that the deviation, in Figure 5.3b, at full rack size, 48, is very high from all other rack sizes. We leave out this case in our second dataset experiments, and concentrate on rack sizes of up to 32, since for these cases the LA part seems to show in general good weak scaling.

5.3.2 Results using dataset #2

In this section we present the results for more rigorous runs based on our initial experiments discussed in section 5.3.1.2. Moreover, we verify the differences between mappings using the network model described in section 5.4. The information on the matrices used is given in Table 5.7. Further information on the application parameters is given in Table 5.8. Although we are more interested in optimizing the large costly cases, we have also included the small cases (half and 1 rack) in our runs for completeness.

Table 5.7: Information on matrices of dataset #2.

Name	Dim.	NNZ	LDim.	LNNZ	M.	D.	S.
r005a	530,189,304	5.20592E+11	17,103,317	1,077,170,909	c1	F	3.682
r01_A09a	574,827,349	6.68289E+11	13,369,180	722,279,401	c1	F	4.041
r01_A09b	574,827,349	6.68289E+11	9,125,498	337,952,291	c2	F	4.058
r01_A14b	1,056,962,719	8.67762E+11	16,782,710	434,123,420	c2	T	1.541
r02_A09a	1,597,056,996	2.30478E+12	25,351,680	1,161,556,620	c1	F	1.807
r02_A09b	1,597,056,996	2.30478E+12	17,945,343	584,246,152	c2	F	1.814
r02_A10a	1,675,942,566	2.28531E+12	26,605,737	1,150,923,937	c1	T	1.626
r04a	2,614,910,212	4.3005E+12	29,383,221	1,089,914,846	c1	F	1.262
r04b	2,614,910,212	4.3005E+12	20,592,675	535,897,640	c2	F	1.264
r08a	4,232,122,420	7.49068E+12	33,325,233	933,263,372	c1	T	0.840
r12a	5,155,975,309	1.24753E+13	33,272,515	1,043,795,926	c1	F	0.943
r16a	9,183,646,229	1.43838E+13	51,308,182	901,414,818	c1	T	0.342
r16b	9,183,646,229	1.43838E+13	36,014,299	440,681,284	c2	T	0.340
r24a	13,622,481,722	2.51012E+13	61,642,528	1,032,398,385	c1	T	0.272
r32a	23,709,299,558	2.8917E+13	92,977,645	885,937,823	c1	T	0.102

All test cases in this section were run with a modified source code, in order to better exploit the torus structure of the network. In particular, our initial results for 2 and 8 (and somewhat for 32) racks suggested that it may be beneficial to map each CSg onto a subtorus. For the interconnect of Mira this can only be done if |CSg| is a power of 2. The code used in section 5.3.1.2 had a possibility of having extra (helper) MPI processes for the diagonal MPI processes, but with one or two of these extra processors, |CSg| is generally not a power of 2 for any of the rack sizes, see Table 5.5. We therefore changed the code so that instead of devoting extra MPI processes, the modified source code distributes the excess of nonzero matrix elements on diagonal MPI processes (whenever diagonal MPI processes have more LNNZ than any of the

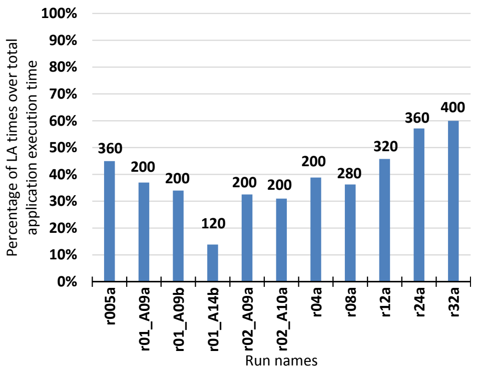
Table 5.8: Application parameters used for dataset #2.

Name	**Procs.**	n_d	**M.**	**|RSg|**	**R.msg**	**Nucleus**
r005a	496	31	c1	16	66	A10Z5N5(09)
r01_A09a	946	43	c1	22	51	A09Z5N4(10)
r01_A09b	2016	63	c2	32	35	A09Z5N4(10)
r01_A14b	2016	63	c2	32	65	A14Z7N7(08)
r02_A09a	2016	63	c1	32	97	A09Z4N5(11)
r02_A09b	4005	89	c2	45	69	A09Z4N5(11)
r02_A10a	2016	63	c1	32	102	A10Z2N8(12)
r04a	4005	89	c1	45	113	A09Z6N3(12)
r04b	8128	127	c2	64	79	A09Z6N3(12)
r08a	8128	127	c1	64	128	A09Z5N4(12)
r12a	12090	155	c1	78	127	A08Z3N5(14)
r16a	16110	179	c1	90	196	A11Z3N8(12)
r16b	32640	255	c2	128	138	A11Z3N8(12)
r24a	24531	221	c1	111	236	A10Z5N5(12)
r32a	32640	255	c1	128	355	A16Z8N8(10)

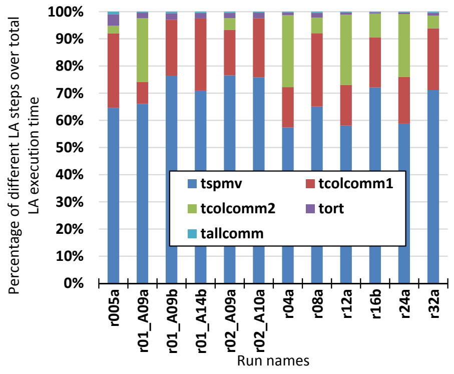
off-diagonal MPI processes) evenly among its CSg members without additional communication overhead. This tends to add less than a percent or so to the LNNZ of the CSg members, but it allows us to better exploit the network structure of Mira. So, in this modified version we do not have the n_{extra} term, and |RSg|=|CSg| for all test runs. Instead, we introduce the term "D." to identify whether the distribution was performed or not, in essence its role is similar to that of n_{extra} when $n_{extra} \neq 0$. So, if it has a value of "T" then it is distributed, and "F" otherwise. To differentiate these two versions of the source code we call the one with n_{extra} the *old* code, and the one with "D." the *new* code. Note that $n_{extra} = 0$ and D. = F corresponds to essentially the same source code.

5.3.2.1 Baseline performance analysis

Initial results from work on dataset # 1 suggest new approaches to apply. First, when possible we should try to fit torus geometry, these are the .5, 2, 8, and 32 rack cases with one MPI process per node, i.e., M. = $c1$. For the cases of 1, 4, and 16 racks the groups can fit the torus geometry with 2 MPI processes per node. See Table 5.8 for details of group sizes. Another approach we take from our previous study is the change in MPI routines for both of the col-



(a) The numbers on bars indicate the number of iterations needed set or needed for convergence.

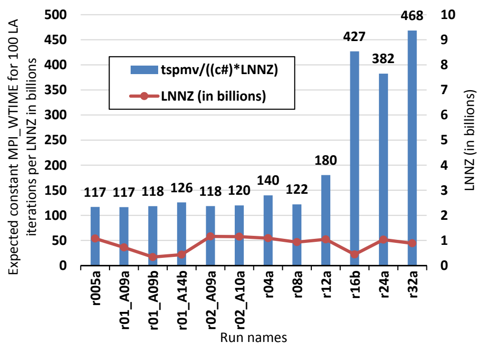


(b) Percentages breakdown of different LA steps.

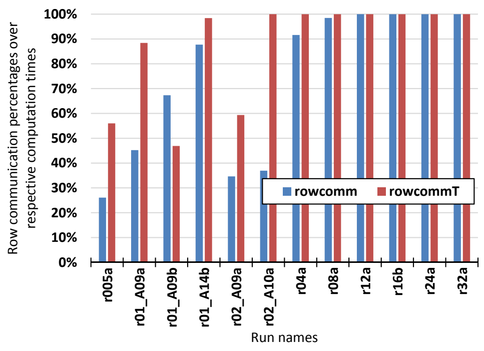
Figure 5.4: Percentage breakdown of execution times of dataset #2 runs with the *Default* mapping (if available).

umn MPI communication routines. Recall from section 5.3.1.2 that using MPI_ALLREDUCE operation instead of MPI_ALLGATHERV operation can further reduce the run time. We investigate this for all the test runs. Additionally, we also investigate a similar swap between MPI_REDUCE_SCATTER and MPI_ALLREDUCE. The baseline performance of the *new* code with *Default* mapping is given in Figure 5.4. We include only the ones for which we have the results with the *Default* mapping and original combination of column MPI routine calls, i.e., MPI_REDUCE_SCATTER and MPI_ALLGATHERV. The numbers above the bars in this figure represent the number of iterations required for convergence of LA. Note that the more the iterations the higher the percentage of LA, Figure 5.4a.

Figure 5.4b additionally gives percentage of times spent on each LA part separately. Similar to the first dataset, the column communication takes significant proportion of overall time, at least 30% in most of the cases and up to 40% in some cases. Here all the times are maximum values. Since we have explicit synchronization points before each of the communications start happening, the times shown are accurate communication times, i.e., no wait time is included. Additionally, while we had a single *tcolcomm* indicating the column communication with our previous study, here we have *tcolcomm1* and *tcolcomm2*, which give separate times for two column communications. This will help us identify the best MPI routine to use. Note that in here some of the runs seem to have no values for *tcolcomm2*, actually these runs did not separate the two column communications as such on those we have only *tcolcomm* values which were put as *tcolcomm1* time.



(a) The numbers on bars indicate the actual values of *tspmv*/LNNZ.



(b) *rowcomm* and *rowcommT* percentages over computation times for SpMVM and SpMVM$^{\mathsf{T}}$, respectively.

Figure 5.5: Times for *tspmv*/LNNZ and row communication (in percentages) with *Default* mapping for dataset #2 runs.

In Figure 5.5a we show the times for *tspmv*/((c#)*LNNZ) (in billions) on left y-axis, and right y-axis shows the LNNZ (in billions). Here, in calculation of *tspmv*/((c#)*LNNZ), (c#) means whether to use 1 or 2 depending on the mode *c*1 and *c*2, respectively. The actual *tspmv*/((c#)*LNNZ) times are shown on top of the bars. As mentioned with previous dataset this shows the approximate *tspmv* execution time for all the cases. Next, in Figure 5.5b we show the percentages of the maximum row communications with regard to maximum SpMVM and SpMVM$^{\mathsf{T}}$ times. From all these information we can identify the cases that do not completely overlap, which mainly include 4 rack and above runs. As will be shown later, with the proposed mappings we achieve full overlap in some of cases which were not completely overlapped with the *Default* mapping shown here.

In Table 5.9 we show the actual/pure *tspmv* execution times, i.e., SpMVM+SpMVM$^{\mathsf{T}}$ only, without any row communication for some of the runs. In last two columns of Table 5.9 we also put *tspmv* times with row communication, and *tspmv*/(c#)*LNNZ values for the dataset #2 runs (similar to the ones we show in Figure 5.5a). In last column of Table 5.9 we show the values in bold for those cases which were not fully overlapped. These hold (except for *r*08*a*) with the results we obtained regarding the runs that do not fully overlap from Figure 5.5.

Table 5.9: Pure SpMVM times for some of the cases of dataset #2.

Name	**M.**	**Nth.**	**LNNZ** (in billions)	**Pure *tspmv* Exec. Time** (in sec.)	$\frac{\textbf{(Pure } \textit{tspmv}\textbf{)}}{\textbf{(c\#)*LNNZ}}$	***tspmv*** (in sec.)	$\frac{\textit{tspmv}}{\textbf{(c\#)*LNNZ}}$
r005a	c1	64	1.08	124.90	115.96	125.88	116.86
r005a	c1	63	1.08	125.84	116.83	—	—
r01_A09a	c1	64	0.72	83.57	115.70	84.15	116.50
r01_A09a	c1	63	0.72	84.19	116.56	—	—
r01_A09b	c2	32	0.34	78.19	115.67	80.00	118.36
r01_A09b	c2	31	0.34	79.40	117.47	—	—
r01_A14b	c2	32	0.43	105.87	121.93	109.17	125.73
r01_A14b	c2	31	0.43	107.43	123.73	—	—
r02_A09a	c1	64	1.16	136.05	117.13	137.57	118.43
r02_A09b	c2	32	0.58	139.11	119.05	—	—
r02_A10a	c1	64	1.15	136.67	118.75	138.00	119.90
r04a	c1	64	1.09	—	—	152.49	**139.91**
r08a	c1	64	0.93	—	—	113.78	121.91
r12a	c1	64	1.04	—	—	188.13	**180.23**
r16b	c2	32	0.44	—	—	376.35	**427.01**
r24a	c1	64	1.03	—	—	394.85	**382.46**
r32a	c1	64	0.89	—	—	415.00	**468.43**

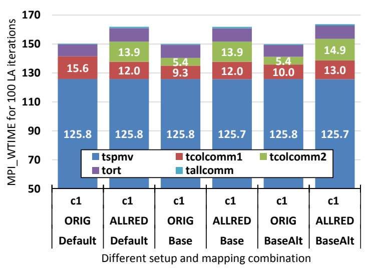
Finally, in Table 5.10 we give the similar information as one given in Table 5.5, but for dataset #2. Note for all cases where |CSg| is a power of two, except the 1/2 and 8 rack cases, the CSg's map exactly onto a subtorus of the physical network either with the *Default* or the *Base* and *BaseAlt* mappings (or with all three of them). In order to test the importance of mapping the CSg's exactly onto a subtorus we introduced for 8 racks *Colfit1* (= CAEDBT) mapping for which the |CSg| fits a DB $= 16 \times 4$ subtorus; in addition we also considered *Colfit2* (= CADBET) mapping for which two |CSg|'s fit a DBE $= 16 \times 4 \times 2$ subtorus.

Table 5.10: Geometry and *Base* mapping information for dataset #2 runs.

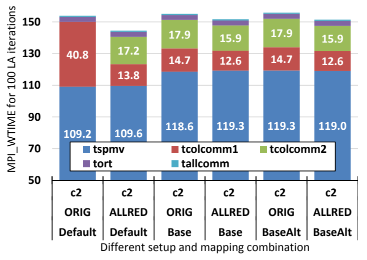
Name	n_d	**|CSg|**	**A,B,C,D,E,T**	***Base***	**|CSg|** maps **|subtorus|** evenly?	**|CSg|** equals **|subtorus|**?
r005a	31	16	4,4,4,4,2,1	DCBAET	*Default* *Base/BaseAlt*	—
r01_A09a	43	22	4,4,4,8,2,1	DCBAET	—	—
r01_A09b r01_A14b	63	32	4,4,4,8,2,2	DCBAET	*Default* *Base/BaseAlt*	*Default*
r02_A09a r02_A10a	63	32	4,4,4,16,2,1	DCBAET	*Default* *Base/BaseAlt*	*Default* *Base/BaseAlt*
r02_A09b	89	45	4,4,4,16,2,2	DCBAET	—	—
r04a	89	45	8,4,4,16,2,1	DACBET	—	—
r04b	127	64	8,4,4,16,2,2	DACBET	*Default* *Base/BaseAlt*	*Default* *Base/BaseAlt*
r08a	127	64	4,4,16,16,2,1	DCBAET	*Default* *Base/BaseAlt* *Colfit1* *Colfit2*	*Colfit1*
r12a	155	78	8,4,12,16,2,1	DCABET	—	—
r16a	179	90	4,8,16,16,2,1	DCBAET	—	—
r16b	255	128	4,8,16,16,2,2	DCBAET	*Default* *Base/BaseAlt*	*Base/BaseAlt*
r24a	221	111	4,12,16,16,2,1	DCBAET	—	—
r32a	255	128	8,8,16,16,2,1	DCBAET	*Default* *Base/BaseAlt*	*Base/BaseAlt*

5.3.2.2 Empirical results and conclusions

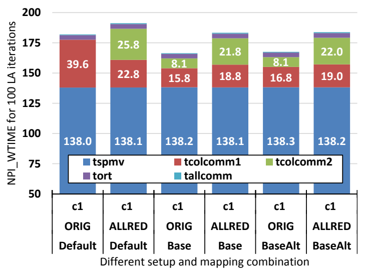
Detailed timings for dataset #2 are given in Figures 5.6 and 5.7 with the execution time in seconds on the y-axis normalized to 100 iterations. We analyze each rack case separately. Here we also introduce new terms "ORIG", "ALLRED", and "1ALLRED". Later on we also use "2ALLRED" name. These names are only meant to differentiate between two different MPI routine calls used during the column communication stages. "ORIG" means that we use MPI_REDUCE_SCATTER routine for first column communication, *tcolcomm1*, and MPI_ALLGATHERV for the second part, *tcolcomm2*. The name "ORIG" already suggests that this is the setup with the original source code, i.e., how it was used to date. Next, we name "ALLRED" the one where we have MPI_ALLREDUCE for both column communications; "1ALLRED" for the one where we use MPI_ALLREDUCE for the first column communication and MPI_ALLGATHERV for the second column communication; and, finally, "2ALLRED" for the one where we have MPI_REDUCE_SCATTER for first column communication and MPI_ALLREDUCE for the second column communication.



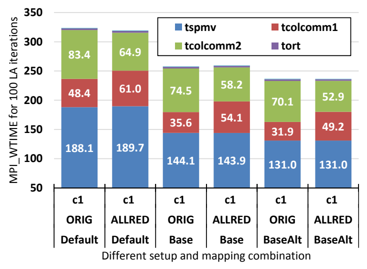
(a) r005a.



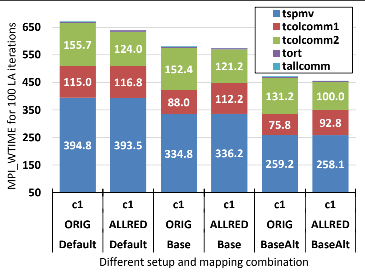
(b) r01_A14b.



(c) r02_A10a.

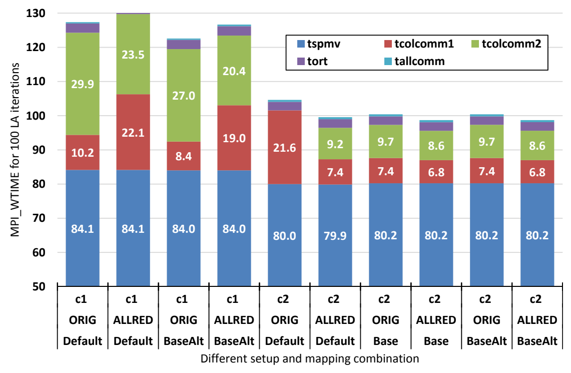


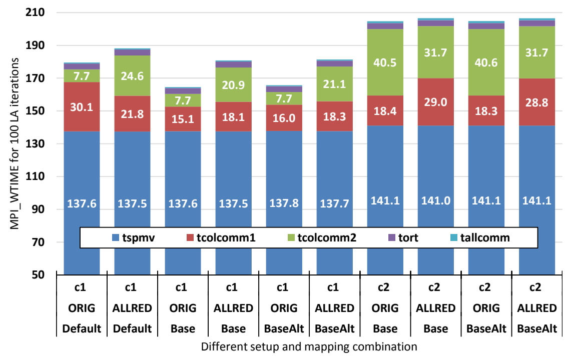
(d) r12a.

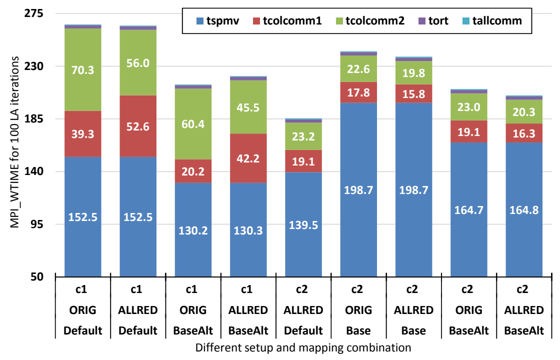


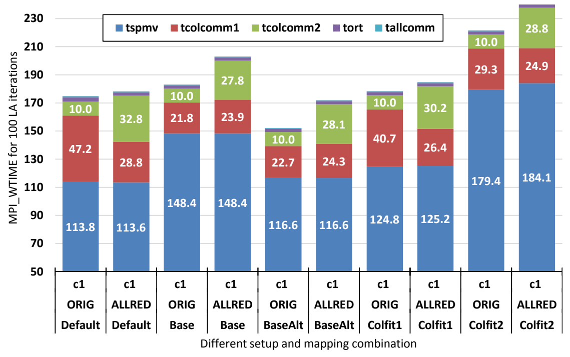
(e) r24a.

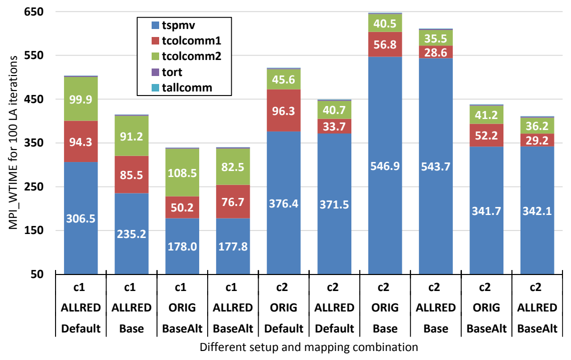
Figure 5.6: Detailed execution times for each of the cases of dataset #2.


(a) r01_A09 (a=$c1$ and b=$c2$).


(b) r02_A09 (a=$c1$ and b=$c2$).


(c) r04 (a=$c1$ and b=$c2$).


(d) r08a.


(e) r16 (a=$c1$ and b=$c2$).

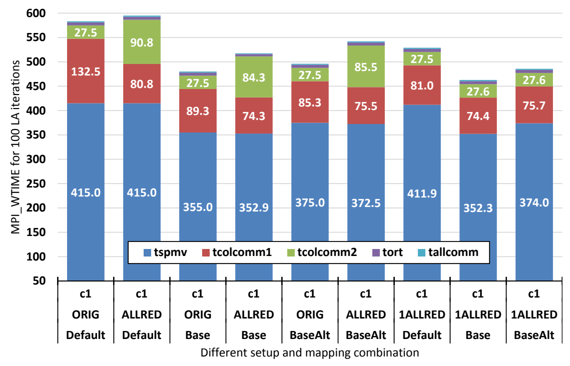

(f) r32a.

Figure 5.7: Detailed execution times for each of the cases of dataset #2.

The .5 and 1 rack cases are not costly and so we focus more on larger cases as mentioned before. For the .5 rack case, the "ORIG" setup is more efficient, while for 1 rack case it is slightly better to use the "ALLRED" setup. The 1 rack case is not affected much by a mapping, but it is important to run it in $c2$ mode, so that the column communication maps better onto the subtorus geometry.

From Figures 5.6c and 5.7b (2 rack runs) it can be seen that the mode should be set to $c1$ and "ORIG" setup should be used. As for the mapping, both *Base* and *BaseAlt* perform almost equally and reduce the time by 9% compared to *Default* mapping. Notice that the row communication is fully overlapped in these cases.

Results for the 4 rack case are given in Figure 5.7c. Interestingly, in $c1$ mode with *BaseAlt* mapping the row communication seems to be completely overlapping with computation. However, the *Default* mapping in $c2$ mode fits the torus geometry and with "ALLRED" setup gives better performance. This emphasizes the importance of selecting the correct mode and set of MPI operations to use for column communications. The 8 rack case results are given in Figure 5.7d. Here, at rack size 8, both of the newly introduced mappings, *Colfit1* and *Colfit2*, did not perform as expected, although they fit the torus geometry nicely. This case is run in $c1$ mode and the *BaseAlt* mapping performs better than all mappings with "ORIG" setup and gives reduction of up to 12% compared to the *Default* mapping.

The results for 12 and 24 racks are similar, Figures 5.6d and 5.6e. Both run in $c1$ mode with not much difference in MPI operation changes for column communication. The run with *BaseAlt* mapping, with "ALLRED" setup, outperforms the *Default* mapping by 33% and 34% for these cases by reducing both column and row communication in both cases.

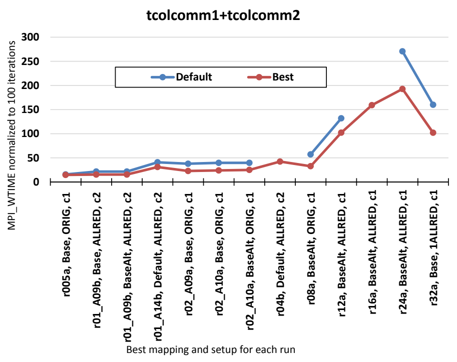
The 16 rack case result from Figure 5.7e suggests the $c1$ mode and "ALLRED" setup to be used. In this case *BaseAlt* mapping gives the best performance with reduction of 36% and 40% for two test cases. It seems that the row communication is mostly overlapped in this case. Note that for 16 racks in $c2$ mode the column communication time is significantly reduced compared to $c1$ mode: in $c2$ mode the $|\text{CSg}| = 128$ (vs. 90 in $c1$ mode, for example see Table 5.10), and maps well onto the underlying network resulting in efficient column communication. Unfortunately, the row communication in $c2$ mode is extremely slow, and therefore $c1$ actually outperforms $c2$ for this particular case. However, we expect that for less sparse matrices, with a higher computational load (i.e., increasing LNNZ) and small communication load (i.e., decrease in LDim.) $c2$ might actually be more efficient than $c1$ on this rack size.

Lastly, the results for 32 rack case are given in Figure 5.7f. While the *BaseAlt* mapping reduces the run time, the *Base* mapping outperforms all mappings reducing run time by 21%. The $c1$ mode and “1ALLRED” setup should be used in order to achieve this performance.

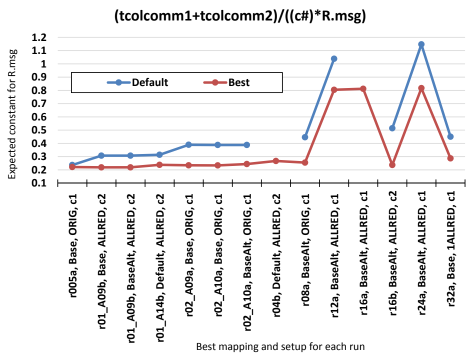
Overall, for all rack sizes 8 and below we seem to have full overlap, with the exception of $r04a$, which seems to be mostly overlapping. Beyond 8 rack sizes we seem to have some overlap, but certainly not full. Notice that most of the results obtained in this section are consistent with the ones obtained in dataset #1 with regard to the best performing mappings and column MPI routine setup with regard to single MPI rank per node. The only exception to this is 32 rack case for which in dataset #1 we used “2ALLRED” setup, but from results of dataset # 2 we found that we need to use the “1ALLRED” setup for the column communication. Also, since in this part we have also tested with $c2$ mode, we found that we should use $c2$ for 1 and 4 rack sizes; depending on the specific run details and sparsity, $c2$ mode may also be beneficial for 16 rack runs. In Table 5.11 we show the *100Avg* and *tspmv*/(c#)*LNNZ (in billions) results for dataset #2. From these results we can clearly see the effect of best mapping and setup combination for the larger runs, specifically starting from 4 rack. At 12 rack size we achieved full overlap, but at racks higher than 12 we achieved significant reduction of row communication though not full overlap of communication with computation.

Table 5.11: Comparison of *Default* and best mapping results for dataset #2 runs.

	Best setup			**$\frac{tspmv}{(c\#)*LNNZ}$**			**100Avg**	
Run	**Mapping**	**Setup**	**M.**	**Default**	**Best**	**LNNZ** $(*10^9)$	**Default**	**Best**
r005a	Base	ORIG	c1	116.82	116.77	1.08	1.47	1.45
r01_A09b	Base BaseAlt	ALLRED	c2	118.36	118.70	0.34	1.05	0.99
r01_A14b	Default	ALLRED	c2	125.73	126.18	0.43	1.57	1.47
r02_A09a	Base	ORIG	c1	118.43	118.48	1.16	1.80	1.64
r02_A10a	Base BaseAlt	ORIG	c1	119.90	120.10	1.15	1.83	1.67
r04b	Default	ALLRED	c2	—	130.13	0.54	—	1.86
r08a	BaseAlt	ORIG	c1	121.91	124.97	0.93	1.75	1.52
r12a	BaseAlt	ALLRED	c1	180.23	125.55	1.04	3.28	2.35
r16a	BaseAlt	ALLRED	c1	—	197.29	0.90	—	3.40
r24a	BaseAlt	ALLRED	c1	382.46	249.96	1.03	6.74	4.55
r32a	Base	1ALLRED	c1	464.99	397.61	0.89	5.23	4.58



(a) Results for *tcolcomm* times.



(b) Results for *tcolcomm* time divided by R.msg (in MiB).

Figure 5.8: Comparison of *Default* mapping and the best mapping combination for dataset #2 runs.

Finally, in Figures 5.8a and 5.8b we show the similar results as of that shown for dataset # 1 in Figures 5.3c and 5.3d but for dataset # 2. In Figure 5.8b we show the *tcolcomm*/((*c*#)*R.msg), where *c*# is replaced by 1 or 2 depending on the mode, i.e., *c*1 or *c*2, respectively. Note that ((*c*#)*R.msg) is a measure of the communication load per node, rather than per MPI processor. For most partition sizes we only show the optimal mode, i.e., *c*1 or *c*2, but for 16 racks we show both *c*1 and *c*2: in *c*2 mode the CSg's fit the underlying network topology, leading to a very efficient column communication, but unfortunately the row communication is very inefficient for this case, and the overall performance is worse than in *c*1 mode, see Figure 5.7e.

The Figure 5.8b clearly shows weak scaling of the column communication: the value obtained for *tcolcomm*/((*c*#)*R.msg) remains approximately constant at about 0.2 to 0.3 as we go from 1/2 rack to 32 racks, provided that we use a suitable mapping and fit the underlying network topology. In order to do so, |CSg| has to be a power of two, and we have to use *c*2 mode on 1, 4, and 16 racks. If the CSg's do not fit the underlying network, we also observe weak scaling, but much less efficient: for 12, 16, and 24 racks (all in *c*1 mode) the CSg's do not map well onto the underlying network, and *tcolcomm*/((*c*#)*R.msg) is again approximately constant, at about 0.8, that is about 3 times slower than for the cases that do fit the underlying network.

In summary we suggest to use *BaseAlt* mapping for rack sizes 8 and higher, with the exception of 32 where we suggest the usage of *Base* mapping. As for the MPI call setup at these rack sizes we suggest usage of “2ALLRED”, exceptions are 8 and 32 rack sizes, where we suggest the usage of “ORIG” and “1ALLRED” setups, respectively. For all of these rack sizes we suggest running one MPI process per node, i.e., M. = *c*1. Based on these results we do the final set of test runs, results are presented in Table A.3.1, where we summarize the results with the default settings for both *old* and *new* source codes, and the best performing setup with *new* source code, without any additional barriers. (From here one also can see our suggestions for smaller rack sizes, i.e., 4 and below.) For some of the cases, we present several different datasets to verify the robustness of the conclusions independent of differences in matrix properties. All source codes used in this table do not use the synchronization points during the LA iterations, this is how the application runs in production runs. Also, one can notice that *100Avg* of *Default* mapping and MPI call setup presented in Figures 5.6 and 5.7 are marginally higher than that of the ones shown in Table A.3.1, which shows that the overhead caused by the introduced synchronization points are negligible.

In general, for the final set of runs presented in Table A.3.1 the *new* source code gives improved results over the *old* source code most of the times. Though for 16-rack test this is not the case, however, even for this run the *new* source code combined with the column MPI setup combination and the best mapping outperforms the *old* source code too. In column "D." the letter has the same meaning as before and the number indicates the value of the n_{extra} for *old* code. We put them in one column to save space. As it can be seen from the Table A.3.1 with proposed mappings and MPI routine setups for the column communication we are able to achieve up to 39% reduction in single LA execution time (as compared both to *old* and *new* codes).

5.4 Network load model and evaluation of mappings

Topology-aware mapping of processes to physical processors in large-scale computations can significantly reduce the communication overheads and improve overall execution time [8, 16, 9, 31, 17]. We analyze and compare the communication and network behavior under different mappings using metrics from existing models [31, 32]: *maximum dilation*, *average dilation per edge*, *average bytes per link*, *worst-case congestion*, and *average network traffic*. In [32] authors have found that combining the *average bytes per link* and *worst-case congestion* metrics improves the model. This is due to the high values for so-called rank correlation coefficient (RCC) to predict the mapping correctly. The authors of [32] have found that for their numerical simulations both the *worst-case congestion* and the *average bytes per link* showed a RCC of about 90%, and the *maximum dilation* has a RCC of about 60% to 80%. We use the following network load model to measure the quality of our mappings.

The process (task) communication of the LA is modeled by a graph $G = (V_G, E_G)$, where the vertex set V_G is the set of processes, an edge $uv \in E_G$ denotes a message sent from u to v and the weight, $w(uv)$, of the edge is the size of the message. We define two communication graphs G_{col} and G_{row} associated with CSg and RSg on 2D half grid, respectively. Likewise, the physical interconnection network is modeled by a graph $H = (V_H, E_H)$, where the vertex set V_H is the set of physical nodes, and E_H represents the communication links between physical nodes. We assume that the messages are routed between nodes constituting the shortest path, which is the case for IBM BG/Q machines, provided all nodes are in a healthy state. A shortest path between nodes u and v is denoted $p(u, v)$, and the set of all shortest paths between nodes u and v is denoted $P(u, v)$. We use the idea behind the default IBM BG/Q parallel active message interface (PAMI) [37] routing algorithm [20, 28]; if the routing algorithm allows more than one shortest paths then each such path in $P(u, v)$ is used with equal probability for sending a message. We further assume a static routing of messages and that all links in E_H have the same capacity.

The topology mapping is represented by a function $\Gamma : V_G \to V_H$. In order to compare the effectiveness of task mappings, we use the following commonly used metrics. One of the earliest metrics used is the *maximum dilation* (max_dil) of a mapping Γ, and is defined as

$$\text{max\_dil}(\Gamma) = \max_{uv \in E_G} d_\Gamma(uv),$$

where $d_\Gamma(uv)$ is the number of hops between $\Gamma(u)$ and $\Gamma(v)$ in graph H. This metric aims at minimizing the length of the longest wire in a circuit [32]. The *average dilation per edge* (avg_dil) measures the average number of links traversed by each message and is defined as

$$\text{avg\_dil}(\Gamma) = \sum_{uv \in E_G} d_\Gamma(uv)/|E_G|.$$

Next two metrics indicate a congestion on network links, and in [32] authors have found that including both of these metrics in the model improves the prediction accuracy of the model. The *average bytes per link* (avg_bytes) metric measures the average number of bytes that pass through a hardware link on the network,

$$\text{avg\_bytes}(\Gamma) = \sum_{uv \in E_G} d_\Gamma(uv) \cdot w(uv)/|E_H|.$$

The *worst-case congestion* (max_cong) on network is measured by the maximum number of bytes that pass through any link,

$$\text{max\_cong}(\Gamma) = \max_{\ell \in E_H} T_\Gamma(\ell),$$

where

$$T_\Gamma(\ell) = \sum_{uv \in E_G} \left(w(uv) \cdot |p : p \in P(\Gamma(u), \Gamma(v)) \wedge \ell \in p| / |P(\Gamma(u), \Gamma(v))| \right),$$

i.e., the total traffic crossing the link $\ell \in E_H$.

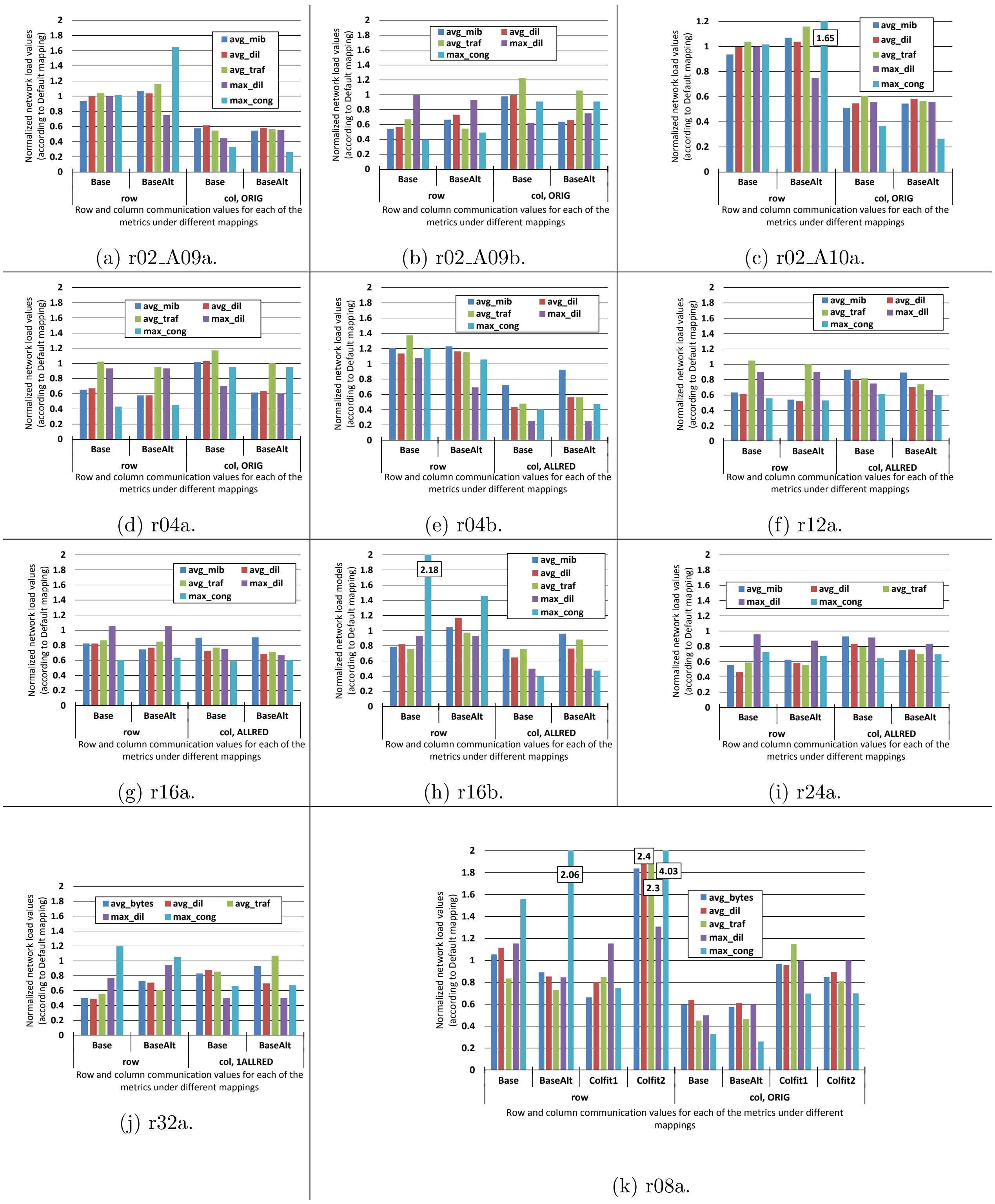
Finally, the *average network traffic* (avg_traf) over all links is defined as,

$$\text{avg\_traf}(\Gamma) = \sum_{\ell \in E_H} T_\Gamma(\ell)/|E_H|.$$

Note that in general the communication graph G does not contain any time-stamp information, and so the worst-case congestion might not be a good approximation to the actual network congestion when G represents a large time window. However, in our case both G_{col} and G_{row} capture the communication happening over a small time window for a number of collective calls. Therefore, two congestion metrics above approximate the network congestion well during the column and row communications in LA.

5.4.1 Evaluation of mappings

The mappings *Base* and *BaseAlt* (in case of 8 rack runs also the mappings *Colfit1* and *Colfit2*) are evaluated for both row and column communication graphs by comparing their metric values to the values of the *Default* mapping. Figure 5.9 gives all metrics computed for both graphs G_{row} and G_{col}. We have normalized the metric values of the *Base*, *BaseAlt*, *Colfit1*, and *Colfit2* mappings with respect to the *Default* mapping in order to show all metrics on one scale. We expect to obtain better performance with the mapping with lower metric values. As mentioned before, all metrics do not necessarily carry the same weight for choosing the correct mapping because some metrics have higher correlation to better performance than others. Therefore, small improvement in one metric can be more effective than a larger improvement in another.



(a) r02_A09a.

(b) r02_A09b.

(c) r02_A10a.

(d) r04a.

(e) r04b.

(f) r12a.

(g) r16a.

(h) r16b.

(i) r24a.

(j) r32a.

(k) r08a.

Figure 5.9: Metrics computed for test runs from dataset #2 starting from 2 racks.

Recall that the message size between tasks u and v is denoted $w(uv)$. During the mapping of tasks to nodes, some pair of tasks are placed in the same nodeboard, midplane, or rack, while others are placed in different racks. In order for our model to capture this difference we adjust the weight of an edge between tasks u and v as follows: if they are mapped to the same nodeboard, then the weight of the message between them is $0.9 \times w(uv)$, if they are mapped to the same midplane or rack, then the weight is $w(uv)$, and if they are mapped to different racks, the weight is given by $1.1 \times w(uv)$.

As we have mentioned in section 5.2.2, the mappings *Base* and *BaseAlt* were introduced with the goal of decreasing the *maximum dilation* for column communication groups. One can see from Figure 5.9 that indeed the *maximum dilation* values for introduced mappings are below 1, i.e., these values are below than the value of *Default* mapping for columns. For the row communication groups, the *BaseAlt* mapping was aimed to further reduce the *maximum dilation* value compared to the *Base* mapping at a (possible) cost of slight increase on column communication. This can also be seen in Figure 5.9, except for the 32-rack size case. However, notice that 32-rack size case is the only one for which the *Base* mapping is more efficient that the *BaseAlt* mapping. Many other mapping values were reduced for the proposed mappings as well, as we discuss in details below.

Overall, the metric values are in qualitative agreement with the experimental results discussed in section 5.3.2.2 and we discuss each rack case separately. For both of the 2-rack cases, the column communication time is reduced with the proposed mappings, *Base* performing slightly better than *BaseAlt*, while the row communication time did not change much. This agrees with the mapping values obtained in Figures 5.9a and 5.9c. Note that for the 2-rack case in $c2$ mode, Figure 5.9b, the initial results where significantly higher than the ones in $c1$ mode, therefore we do not have complete run time results for this case and cannot compare with metrics.

The 4-rack case in $c2$ mode is the only one where the *Default* mapping outperforms other mappings. While in $c1$ mode it is the most inefficient, when we set the mode to $c2$ the row communication seems to significantly overlap with the *Default* mapping. Both mappings *Base* and *Basealt* weaken the overlapping, while slightly improving the column communication in

$c2$ mode. This is justified with the network model results, Figure 5.9d shows that the metric values for the proposed mappings are lower than the default mapping while Figure 5.9e shows that the metric values of the proposed mappings are higher for the row communication and lower for the column communication than the default mapping.

For 8 rack runs, as it can be seen from Figure 5.7d the mapping *Colfit2* has increased tremendously the row communication, and this is also verified by the network model shown in Figure 5.9k, as all metric values have increased significantly compared to the *Default* mapping. However, this mapping improved the column communication, as it can also be seen from the lower metric values. The mapping *Colfit1* only slightly increased the row communication and slightly decreased the column communication. On the other hand, both mappings *Base* and *Basealt* significantly reduced the column communication, while the row communication was increased with the *Base* mapping and did not change much with the *BaseAlt* mapping.

The 12-rack, the $c1$ mode of 16-rack, and 24-rack cases have similar communication pattern, Figures 5.9f, 5.9g,and 5.9i respectively. Both communication times and the metric values of runs using the *Default* mapping were first reduced with the *Base* mapping, and then further improved with the *BaseAlt* mapping. On the other hand, the row communication of the 16-rack case in $c2$ mode, Figure 5.9h,was decreased under the *BaseAlt* mapping, but increased under the *Base* mapping, which can be explained by a high value of the *maximum dilation* metric. The column communication was decreased under both proposed mappings.

Finally, the 32-rack case is the only one when MPI calls are changed to "1ALLRED" and the *Base* mapping is more efficient than *BaseAlt* overall due to decrease in communication for both row and column groups. Notice the lower metric values, Figure 5.9j, for *Base* compared to the *BaseAlt*, while both of these mappings give better performance than the *Default* mapping.

Further investigation of the network model is an interesting problem on its own, and we intend to further explore this. In this study, comparing communication patterns from Figures 5.6, 5.7, and 5.9 we can see that the *maximum dilation* has the highest correlation to the performance of the mapping. This confirms that the choice of minimizing the maximum number of hops is a good strategy for reducing the communication overhead. This metric is then followed by the *average dilation per edge* and *average bytes per link*, and then the *average* *traffic*. The *worst-case congestion* metric becomes only important when its value is large (close to a factor of 2 or more).

While it is possible to predict the most efficient mapping when most of the metrics agree, it is difficult when the metric values differ, as it was the case in some of our test runs. Therefore, it would be helpful to find an objective function that captures the information of all metric values, and gives a single value to each mapping. A simple approach would be to find a good linear combination of metric values. Another possible direction is to use a distance metric other than the hop counts, which would possibly decrease the number of metrics to consider if accurate.

5.5 Conclusion

Work in this chapter can be summarized as follows. We proposed topology-aware mappings based on the communication pattern of the LA used in the large-scale nuclear physics application, MFDn, that exploits the network topology of IBM BG/Q supercomputer. The mappings were evaluated with network load model using metrics: *maximum dilation*, *average dilation per edge*, *average bytes per link*, *worst-case congestion*, and *average network traffic*. The metric values were consistent with performance timings for row and column communication. All test cases benefited from the proposed mappings, and we achieved up to 39% better performance compared to the default mapping for the single LA iteration, and up to 28% for overall MFDn run time. Additional 10% reduction was achieved in several cases with the MPI call change suggested.

If the row communication is mostly overlapped, then one should mainly focus on column communication. This can be achieved by choosing the size of the column communicator groups so that the mapping fits well onto the torus dimensions with the correct $c1$ and $c2$ modes. Among such mappings, select the one that minimizes the *maximum dilation* (and possibly *average dilation per edge*) for the column communicator groups, while keeping the *worst-case congestion* reasonably low. Last, optimize the MPI calls, which can be done dynamically during the runtime. On the other hand, if the row communication is not mostly overlapped, then one should consider both row and column communications. In this case, $c1$ mode seems to be more efficient, so the mappings are selected in $c1$ mode according to their *maximum dilation* values. Tuning the MPI calls dynamically at runtime can also help in this case. The proposed *BaseAlt* mapping achieves significantly better performance than the default mapping, and works most efficient for most of the cases.

The network load model introduced in this chapter has a good potential for predicting mappings before their execution due to its relatively good accuracy compared to execution results. The model detected the efficient mappings and inefficient mappings in our test runs. All five metric values derived from the model play a role in the efficiency of the mappings, with *maximum dilation* being the most relevant, followed by *average dilation per edge*, *average bytes per link* and *worst-case congestion* (when it is of the scale), and then *average network traffic*. Therefore, it can be difficult to select the best mapping among efficient ones, unless several metrics agree. The model also performed well to detect the worst mappings, for example in our 8-rack case. The model can be further improved to predict the best mapping by possibly introducing a single-valued objective function that captures the information given by all five metrics. We intend to further investigate the network load model and consider other objective functions or metrics for improving its accuracy and predicting capability.

Finally, recall that whether or not the row communication is mostly overlapping depends on the ratio of the local workload to the communication load, which in turn depends on a combination of the problem size and the sparsity. At a fixed total workload on a fixed total number of compute nodes the communication load increases as the sparsity decreases, since the message size increases. On the other hand, if the local workload and the sparsity are fixed, then the communication load increases as the problem size increases, since the number of nodes increases. For leadership-class runs on Mira, the communication can be overlapped down to about one in a million nonzeros; at a sparsity of one in ten million nonzeros the row communication becomes a significant part of the LA time.

As a future work, we also intend to further investigate the implementation of the SpMVM kernel so that the row communication is maximally overlapped with computation as full overlap does not seem to be achievable for large cases. Using more than one thread to do the communication is worth exploring, as it can also help to achieve the maximal overlapping.

CHAPTER 6. CONCLUSION

The main results and conclusions of this thesis can be summarized as follows:

1. In Chapter 3 we tackled a challenge faced by the MFDn code: The matrix sizes of the input, 3-body, matrix elements became unmanageable, so new approaches to obtain the, m-scheme 3-body, matrix elements were required. One approach was to read the, 3-body, matrix elements in different format, so-called coupled-JT, which enables MFDn to obtain the same amount of information as in the memory-intensive, m-scheme, format but in a much more memory-efficient manner. This approach is well-parallelizable and has been adapted for GPGPUs.

   Experiences with the initial porting of the coupled-JT to m-scheme transformation to GPGPU using CUDA have been presented and already showed promising results in the range of four-to-ten fold improvements. Further improvements and analysis were needed, however, for both the CPU and GPGPU implementations. For example, the GPGPU code may be able to take advantage of the texture memory and multiple streams. The initial results are presented in [52], finally results after the integration of this part into MFDn are presented in [55].

2. In Chapter 4 we analyzed the performance of a symmetric (“s1”) parallel SpMVM approach, as well as three general (nonsymmetric) parallel SpMVM approaches called “a1”, “a2”, and “a3” presented in the literature on distributed memory multi-core architectures. The performance analysis of these implementations was carried out on large scale sparse matrices using the LA. When the same number of diagonals, n_d, is used the time to completion with approaches “a1”, “a2”, and “a3” is typically lower than “s1”. However, in terms of “CPU core hours” consumed, in all examined cases “s1” is more efficient than

"a1", "a2", and "a3". In particular, the combination of a unique half 2D processor grid, communication hiding techniques, and topology aware mapping heuristics used in "s1" results in highly efficient distributed symmetric SpMVM computations. The approaches "a1", "a2", and "a3" do not make use of the underlying symmetry of the sparse matrix, and thus they consume more memory as well as CPU resources overall.

We observed that the communication and computation overlapping technique that we have implemented in "a2" (based on the idea provided in [38]) outperforms the basic approach "a1" [39] in the sparser cases on Hopper, but generally not on Edison, two Cray supercomputers at NERSC. We also observed that the optimization suggested in approach "a3" [38] is generally more efficient than "a2". Furthermore, on Edison "a3" is competitive with or better than "a1" for the sparser cases, in contrast to "a2". This suggests that the benefits of overlapping communication and computation depends on several factors such as the problem size (amount of computation vs. communication), system architecture, and type of interconnect.

The reduced memory footprint and low data movement overhead properties possessed by "s1" are especially important as we move towards the exascale era where these considerations become of paramount importance. These results hold for both interconnect topologies tested, i.e., 3D Torus and Dragonfly, which are very different in their designs. The approach "s1" is currently being adopted in MFDn and has been shown to achieve good scalability [9].

3. In Chapter 5 we considered topology-aware mappings based on the communication pattern of the LA used in the large-scale nuclear physics application, MFDn, that exploits the network topology of IBM BG/Q supercomputer. Our results show that the default mapping for BG/Q is generally not the best for this problem. The default mapping and the proposed mappings were evaluated with network load model using metrics: *maximum dilation*, *average dilation per edge*, *average bytes per link*, *worst-case congestion*, and *average network traffic*. The metric values have shown consistent results with the performance timings for row and column communications, with the *maximum dilation*

being the most relevant. In most of our cases the mapping that optimized this metric was the best mapping, and using this mapping we obtained up to 40% better performance for LA and up to 28% for the overall run.

Overlapping most of the row communication with computation can reduce the execution time of the application significantly. Whether or not the row communication is mostly overlapped depends on the ratio of the local workload to the communication load, which in turn depends on a combination of the problem size and the sparsity. We suggest the following strategy for selecting the best mapping. If the row communication is mostly overlapped, then one should mainly focus on column communication. This can be achieved by choosing the size of the column communicator groups so that the mapping fits well onto the torus dimensions with the correct $c1$ (one MPI process per node) and $c2$ (two MPI processes per node) modes. Among such mappings, select the one that minimizes the *maximum dilation* (and possibly *average dilation per edge*) for the column communicator groups, while keeping the *worst-case congestion* reasonably low. Last, optimizing the MPI calls can result in up to 10% additional reduction of the LA time, which can be done dynamically during the runtime. On the other hand, if the row communication is not mostly overlapped, then one should consider both row and column communications. In this case, $c1$ mode seems to be more efficient, so the mappings are selected in $c1$ mode according to their *maximum dilation* values. Tuning the MPI calls dynamically at runtime can also help in this case. The proposed *BaseAlt* mapping achieves significantly better performance than the default mapping, and works most efficient for most of the cases.

The network load model introduced in Chapter 5 has a good potential for predicting mappings before their execution due to its relatively good accuracy compared to execution results. The model detected the efficient mappings and inefficient mappings in our test runs. The most important metric for predicting the efficiency of a mapping is the *maximum dilation*, followed by *average dilation per edge*, *average bytes per link* and *worst-case congestion* (when it is of the scale), and then *average network traffic*. We intend to further investigate the network load model and consider other objective functions or metrics for improving its accuracy and predicting capability. As a future work, we also intend to further investigate the implementation of the SpMVM kernel so that the row communication is maximally overlapped with computation, as full overlap does not seem to be achievable for large cases.

APPENDIX ADDITIONAL DATA FOR CHAPTER 5

A.1 Information on column message size during parallel LA execution in MFDn

The size of the data (chunk of sub-vector), in MiB, in send buffer collected during the MPI_ALLGATHERV MPI routine among the CSg members defined as "C.msg". In case of MPI_ALLREDUCE it is same with that of first column communication, where the send buffer size would be equal to the R.msg when |CSg|=|RSg|. When this equality does not hold then this value is slightly smaller than shown send buffer message size. In Tables A.1.1 and A.1.2 we show the "C.msg" values for the datasets #1 and #2 discussed in Chapter 5.

Table A.1.1: Application parameters used for dataset #1. This Table is similar to Table 5.4 with addition of column for "C.msg".

Name	**Procs.**	n_d	n_{extra}	**|RSg|**	**|CSg|**	**R.msg**	**C.msg**
02a	2016	63	0	32	32	37	2
02b	2016	63	0	32	32	103	4
04a	4094	89	1	45	46	113	3
08a	8128	127	0	64	64	74	2
08b	8128	127	0	64	64	128	2
12a	12245	155	1	78	79	127	2
16a	16289	179	1	90	91	145	2
32a	32637	253	2	127	129	182	2
48a	49138	311	2	156	158	203	2

Table A.1.2: Application parameters used for dataset #2. This Table is similar to Table 5.8 with addition of column for "C.msg".

Name	**Procs.**	n_d	**M.**	**|RSg|**	**R.msg**	**C.msg**	**Nucleus**
r005a	496	31	c1	16	66	5	A10Z5N5(09)
r01_A09a	946	43	c1	22	51	3	A09Z5N4(10)
r01_A09b	2016	63	c2	32	35	2	A09Z5N4(10)
r01_A14b	2016	63	c2	32	65	3	A14Z7N7(08)
r02_A09a	2016	63	c1	32	97	4	A09Z4N5(11)
r02_A09b	4005	89	c2	45	69	2	A09Z4N5(11)
r02_A10a	2016	63	c1	32	102	4	A10Z2N8(12)
r04a	4005	89	c1	45	113	3	A09Z6N3(12)
r04b	8128	127	c2	64	79	2	A09Z6N3(12)
r08a	8128	127	c1	64	128	2	A09Z5N4(12)
r12a	12090	155	c1	78	127	2	A08Z3N5(14)
r16a	16110	179	c1	90	196	3	A11Z3N8(12)
r16b	32640	255	c2	128	138	2	A11Z3N8(12)
r24a	24531	221	c1	111	236	3	A10Z5N5(12)
r32a	32640	255	c1	128	355	3	A16Z8N8(10)

A.2 Sample *BaseAlt* mapping file

Here we show the sample *BaseAlt* mapping file for the geometry of $3 \times 3 \times 4 \times 5 \times 2$ in $c1$ mode.

```
0 0 0 0 0 0 | 0 1 1 0 1 0 | 1 2 2 0 1 0 | 1 0 3 1 0 0 | 2 1 2 1 0 0
0 0 0 0 1 0 | 1 1 1 0 1 0 | 1 2 2 0 0 0 | 2 0 3 1 0 0 | 2 1 2 1 1 0
1 0 0 0 1 0 | 1 1 1 0 0 0 | 2 2 2 0 0 0 | 2 0 3 1 1 0 | 2 0 2 1 1 0
1 0 0 0 0 0 | 2 1 1 0 0 0 | 2 2 2 0 1 0 | 2 1 3 1 1 0 | 2 0 2 1 0 0
2 0 0 0 0 0 | 2 1 1 0 1 0 | 2 2 3 0 1 0 | 2 1 3 1 0 0 | 1 0 2 1 0 0
2 0 0 0 1 0 | 2 0 1 0 1 0 | 2 2 3 0 0 0 | 1 1 3 1 0 0 | 1 0 2 1 1 0
2 1 0 0 1 0 | 2 0 1 0 0 0 | 1 2 3 0 0 0 | 1 1 3 1 1 0 | 0 0 2 1 1 0
2 1 0 0 0 0 | 1 0 1 0 0 0 | 1 2 3 0 1 0 | 0 1 3 1 1 0 | 0 0 2 1 0 0
1 1 0 0 0 0 | 1 0 1 0 1 0 | 0 2 3 0 1 0 | 0 1 3 1 0 0 | 0 0 1 1 0 0
1 1 0 0 1 0 | 0 0 1 0 1 0 | 0 2 3 0 0 0 | 0 2 3 1 0 0 | 0 0 1 1 1 0
0 1 0 0 1 0 | 0 0 1 0 0 0 | 0 1 3 0 0 0 | 0 2 3 1 1 0 | 1 0 1 1 1 0
0 1 0 0 0 0 | 0 0 2 0 0 0 | 0 1 3 0 1 0 | 1 2 3 1 1 0 | 1 0 1 1 0 0
0 2 0 0 0 0 | 0 0 2 0 1 0 | 1 1 3 0 1 0 | 1 2 3 1 0 0 | 2 0 1 1 0 0
0 2 0 0 1 0 | 1 0 2 0 1 0 | 1 1 3 0 0 0 | 2 2 3 1 0 0 | 2 0 1 1 1 0
1 2 0 0 1 0 | 1 0 2 0 0 0 | 2 1 3 0 0 0 | 2 2 3 1 1 0 | 2 1 1 1 1 0
1 2 0 0 0 0 | 2 0 2 0 0 0 | 2 1 3 0 1 0 | 2 2 2 1 1 0 | 2 1 1 1 0 0
2 2 0 0 0 0 | 2 0 2 0 1 0 | 2 0 3 0 1 0 | 2 2 2 1 0 0 | 1 1 1 1 0 0
2 2 0 0 1 0 | 2 1 2 0 1 0 | 2 0 3 0 0 0 | 1 2 2 1 0 0 | 1 1 1 1 1 0
2 2 1 0 1 0 | 2 1 2 0 0 0 | 1 0 3 0 0 0 | 1 2 2 1 1 0 | 0 1 1 1 1 0
2 2 1 0 0 0 | 1 1 2 0 0 0 | 1 0 3 0 1 0 | 0 2 2 1 1 0 | 0 1 1 1 0 0
1 2 1 0 0 0 | 1 1 2 0 1 0 | 0 0 3 0 1 0 | 0 2 2 1 0 0 | 0 2 1 1 0 0
1 2 1 0 1 0 | 0 1 2 0 1 0 | 0 0 3 0 0 0 | 0 1 2 1 0 0 | 0 2 1 1 1 0
0 2 1 0 1 0 | 0 1 2 0 0 0 | 0 0 3 1 0 0 | 0 1 2 1 1 0 | 1 2 1 1 1 0
0 2 1 0 0 0 | 0 2 2 0 0 0 | 0 0 3 1 1 0 | 1 1 2 1 1 0 | 1 2 1 1 0 0
0 1 1 0 0 0 | 0 2 2 0 1 0 | 1 0 3 1 1 0 | 1 1 2 1 0 0 | 2 2 1 1 0 0
```

2 2 1 1 1 0	0 1 0 2 1 0	1 0 2 2 0 0	1 0 3 2 0 0	0 1 2 3 1 0
2 2 0 1 1 0	0 1 0 2 0 0	2 0 2 2 0 0	1 0 3 2 1 0	1 1 2 3 1 0
2 2 0 1 0 0	0 2 0 2 0 0	2 0 2 2 1 0	0 0 3 2 1 0	1 1 2 3 0 0
1 2 0 1 0 0	0 2 0 2 1 0	2 1 2 2 1 0	0 0 3 2 0 0	2 1 2 3 0 0
1 2 0 1 1 0	1 2 0 2 1 0	2 1 2 2 0 0	0 0 3 3 0 0	2 1 2 3 1 0
0 2 0 1 1 0	1 2 0 2 0 0	1 1 2 2 0 0	0 0 3 3 1 0	2 0 2 3 1 0
0 2 0 1 0 0	2 2 0 2 0 0	1 1 2 2 1 0	1 0 3 3 1 0	2 0 2 3 0 0
0 1 0 1 0 0	2 2 0 2 1 0	0 1 2 2 1 0	1 0 3 3 0 0	1 0 2 3 0 0
0 1 0 1 1 0	2 2 1 2 1 0	0 1 2 2 0 0	2 0 3 3 0 0	1 0 2 3 1 0
1 1 0 1 1 0	2 2 1 2 0 0	0 2 2 2 0 0	2 0 3 3 1 0	0 0 2 3 1 0
1 1 0 1 0 0	1 2 1 2 0 0	0 2 2 2 1 0	2 1 3 3 1 0	0 0 2 3 0 0
2 1 0 1 0 0	1 2 1 2 1 0	1 2 2 2 1 0	2 1 3 3 0 0	0 0 1 3 0 0
2 1 0 1 1 0	0 2 1 2 1 0	1 2 2 2 0 0	1 1 3 3 0 0	0 0 1 3 1 0
2 0 0 1 1 0	0 2 1 2 0 0	2 2 2 2 0 0	1 1 3 3 1 0	1 0 1 3 1 0
2 0 0 1 0 0	0 1 1 2 0 0	2 2 2 2 1 0	0 1 3 3 1 0	1 0 1 3 0 0
1 0 0 1 0 0	0 1 1 2 1 0	2 2 3 2 1 0	0 1 3 3 0 0	2 0 1 3 0 0
1 0 0 1 1 0	1 1 1 2 1 0	2 2 3 2 0 0	0 2 3 3 0 0	2 0 1 3 1 0
0 0 0 1 1 0	1 1 1 2 0 0	1 2 3 2 0 0	0 2 3 3 1 0	2 1 1 3 1 0
0 0 0 1 0 0	2 1 1 2 0 0	1 2 3 2 1 0	1 2 3 3 1 0	2 1 1 3 0 0
0 0 0 2 0 0	2 1 1 2 1 0	0 2 3 2 1 0	1 2 3 3 0 0	1 1 1 3 0 0
0 0 0 2 1 0	2 0 1 2 1 0	0 2 3 2 0 0	2 2 3 3 0 0	1 1 1 3 1 0
1 0 0 2 1 0	2 0 1 2 0 0	0 1 3 2 0 0	2 2 3 3 1 0	0 1 1 3 1 0
1 0 0 2 0 0	1 0 1 2 0 0	0 1 3 2 1 0	2 2 2 3 1 0	0 1 1 3 0 0
2 0 0 2 0 0	1 0 1 2 1 0	1 1 3 2 1 0	2 2 2 3 0 0	0 2 1 3 0 0
2 0 0 2 1 0	0 0 1 2 1 0	1 1 3 2 0 0	1 2 2 3 0 0	0 2 1 3 1 0
2 1 0 2 1 0	0 0 1 2 0 0	2 1 3 2 0 0	1 2 2 3 1 0	1 2 1 3 1 0
2 1 0 2 0 0	0 0 2 2 0 0	2 1 3 2 1 0	0 2 2 3 1 0	1 2 1 3 0 0
1 1 0 2 0 0	0 0 2 2 1 0	2 0 3 2 1 0	0 2 2 3 0 0	2 2 1 3 0 0
1 1 0 2 1 0	1 0 2 2 1 0	2 0 3 2 0 0	0 1 2 3 0 0	2 2 1 3 1 0

2 2 0 3 1 0	0 0 0 4 1 0	1 2 1 4 0 0	1 0 2 4 0 0	0 2 3 4 1 0
2 2 0 3 0 0	1 0 0 4 1 0	1 2 1 4 1 0	2 0 2 4 0 0	0 2 3 4 0 0
1 2 0 3 0 0	1 0 0 4 0 0	0 2 1 4 1 0	2 0 2 4 1 0	0 1 3 4 0 0
1 2 0 3 1 0	2 0 0 4 0 0	0 2 1 4 0 0	2 1 2 4 1 0	0 1 3 4 1 0
0 2 0 3 1 0	2 0 0 4 1 0	0 1 1 4 0 0	2 1 2 4 0 0	1 1 3 4 1 0
0 2 0 3 0 0	2 1 0 4 1 0	0 1 1 4 1 0	1 1 2 4 0 0	1 1 3 4 0 0
0 1 0 3 0 0	2 1 0 4 0 0	1 1 1 4 1 0	1 1 2 4 1 0	2 1 3 4 0 0
0 1 0 3 1 0	1 1 0 4 0 0	1 1 1 4 0 0	0 1 2 4 1 0	2 1 3 4 1 0
1 1 0 3 1 0	1 1 0 4 1 0	2 1 1 4 0 0	0 1 2 4 0 0	2 0 3 4 1 0
1 1 0 3 0 0	0 1 0 4 1 0	2 1 1 4 1 0	0 2 2 4 0 0	2 0 3 4 0 0
2 1 0 3 0 0	0 1 0 4 0 0	2 0 1 4 1 0	0 2 2 4 1 0	1 0 3 4 0 0
2 1 0 3 1 0	0 2 0 4 0 0	2 0 1 4 0 0	1 2 2 4 1 0	1 0 3 4 1 0
2 0 0 3 1 0	0 2 0 4 1 0	1 0 1 4 0 0	1 2 2 4 0 0	0 0 3 4 1 0
2 0 0 3 0 0	1 2 0 4 1 0	1 0 1 4 1 0	2 2 2 4 0 0	0 0 3 4 0 0
1 0 0 3 0 0	1 2 0 4 0 0	0 0 1 4 1 0	2 2 2 4 1 0	
1 0 0 3 1 0	2 2 0 4 0 0	0 0 1 4 0 0	2 2 3 4 1 0	
0 0 0 3 1 0	2 2 0 4 1 0	0 0 2 4 0 0	2 2 3 4 0 0	
0 0 0 3 0 0	2 2 1 4 1 0	0 0 2 4 1 0	1 2 3 4 0 0	
0 0 0 4 0 0	2 2 1 4 0 0	1 0 2 4 1 0	1 2 3 4 1 0	

A.3 Results of best run setups

Table A.3.1: Results of best runs without any synchronization points in source code during LA.

	rack	**run**	**Nucleus**	**Dim.**	**NNZ**	**LDim.**	**LNNZ**
	r01_A09	1,2,3,4	A09Z5N4(10)	574,827,349	668,289,364,616	9,125,498	337,952,291
run	**Procs,n_d**	**Mode**	**D.**	**Code**	**Mapping**	**MPI**	**100Avg**
1	2016,63	c2	0	OLD	Default	ORIG	1.06
2	2016,63	c2	F	NEW	Default	ORIG	1.04
3	2016,63	c2	F	NEW	Base	ALLRED	0.976
4	2016,63	c2	F	NEW	BaseAlt	ALLRED	0.976
	rack	**run**	**Nucleus**	**Dim.**	**NNZ**	**LDim.**	**LNNZ**
	r01_A14	1,2,3	A14Z7N7(8)	1,056,962,719	867,761,992,371	16,782,710	434,123,420
run	**Procs,n_d**	**Mode**	**D.**	**Code**	**Mapping**	**MPI**	**100Avg**
1	2016,63	c2	0	OLD	Default	ORIG	1.54
2	2016,63	c2	T	NEW	Default	ORIG	1.55
3	2016,63	c2	T	NEW	Default	ALLRED	1.45
	rack	**run**	**Nucleus**	**Dim.**	**NNZ**	**LDim.**	**LNNZ**
	r02_A09	1,2,3	A09Z4N5(11)	1,597,056,996	2,304,776,012,970	25,351,680	1,161,556,620
run	**Procs,n_d**	**Mode**	**D.**	**Code**	**Mapping**	**MPI**	**100Avg**
1	2016,63	c1	0	OLD	Default	ORIG	1.86
2	2016,63	c1	F	NEW	Default	ORIG	1.79
3	2016,63	c1	F	NEW	Base	ORIG	1.64
	rack	**run**	**Nucleus**	**Dim.**	**NNZ**	**LDim.**	**LNNZ**
	r02_A10	1,2,3	A10Z2N8(12)	1,675,942,566	2,285,314,123,353	26,605,737	1,150,923,937
run	**Procs,n_d**	**Mode**	**D.**	**Code**	**Mapping**	**MPI**	**100Avg**
1	2016,63	c1	0	OLD	Default	ORIG	1.81
2	2016,63	c1	T	NEW	Default	ORIG	1.82
3	2016,63	c1	T	NEW	Base	ORIG	1.66
	rack	**run**	**Nucleus**	**Dim.**	**NNZ**	**LDim.**	**LNNZ**

Continued on next page

Table A.3.1 – *Continued from previous page*

	r04	1	A09Z6N3(12)	2,614,910,212	4,300,498,397,663	29,383,221	1,089,914,846
		2,3	A09Z6N3(12)	2,614,910,212	4,300,498,397,663	20,592,675	535,897,640
run	**Procs,n_d**	**Mode**	**D.**	**Code**	**Mapping**	**MPI**	**100Avg**
1	4094,89	c1	1	OLD	Default	ORIG	2.97
2	8128,127	c2	F	NEW	Default	ORIG	2.15
3	8128,127	c2	F	NEW	Default	ALLRED	1.81
	rack	**run**	**Nucleus**	**Dim.**	**NNZ**	**LDim.**	**LNNZ**
	r04	1	A07Z4N3(14)	1,244,131,981	2,993,087,120,995	13,979,801	760,027,043
		2,3	A07Z4N3(14)	1,244,131,981	2,993,087,120,995	9,797,497	374,024,971
run	**Procs,n_d**	**Mode**	**D.**	**Code**	**Mapping**	**MPI**	**100Avg**
1	4094,89	c1	1	OLD	Default	ORIG	1.45
2	8128,127	c2	F	NEW	Default	ORIG	1.25
3	8128,127	c2	F	NEW	Default	ALLRED	1.11
	rack	**run**	**Nucleus**	**Dim.**	**NNZ**	**LDim.**	**LNNZ**
	r08	1	A09Z5N4(12)	4,232,122,420	7,490,682,147,171	33,325,233	957,742,166
		2,3	A09Z5N4(12)	4,232,122,420	7,490,682,147,171	33,325,233	933,263,372
		4	A09Z4N5(12)	4,232,122,420	7,490,682,147,171	33,325,233	933,263,372
run	**Procs,n_d**	**Mode**	**D.**	**Code**	**Mapping**	**MPI**	**100Avg**
1	8128,127	c1	0	OLD	Default	ORIG	2.06
2	8128,127	c1	T	NEW	Default	ORIG	1.73
3	8128,127	c1	T	NEW	BaseAlt	ORIG	1.52
4	8128,127	c1	T	NEW	Default	1ALLRED	1.55
	rack	**run**	**Nucleus**	**Dim.**	**NNZ**	**LDim.**	**LNNZ**
	r08	1,2,3,4	A08Z2N6(14)	2,433,601,898	5,270,919,067,746	19,164,067	657,947,271
run	**Procs,n_d**	**Mode**	**D.**	**Code**	**Mapping**	**MPI**	**100Avg**
1	8128,127	c1	0	OLD	Default	ORIG	1.14
2	8128,127	c1	T	NEW	Default	ORIG	1.14
3	8128,127	c1	T	NEW	BaseAlt	ORIG	0.998
4	8128,127	c1	T	NEW	Default	1ALLRED	1.03
	rack	**run**	**Nucleus**	**Dim.**	**NNZ**	**LDim.**	**LNNZ**
	r08	1,2,3,4	A06Z2N4(18)	2,052,828,564	9,851,311,327,509	16,166,544	1,228,231,085

Continued on next page

Table A.3.1 – *Continued from previous page*

run	**Procs,n_d**	**Mode**	**D.**	**Code**	**Mapping**	**MPI**	**100Avg**
1	8128,127	c1	0	OLD	Default	ORIG	1.71
2	8128,127	c1	F	NEW	Default	ORIG	1.71
3	8128,127	c1	F	NEW	BaseAlt	ORIG	1.59
4	8128,127	c1	F	NEW	Default	1ALLRED	1.62
	rack	**run**	**Nucleus**	**Dim.**	**NNZ**	**LDim.**	**LNNZ**
	r12	1,2,3	A08Z5N3(14)	5,155,975,309	12,475,336,222,575	33,272,515	1,044,666,301
run	**Procs,n_d**	**Mode**	**D.**	**Code**	**Mapping**	**MPI**	**100Avg**
1	12245,155	c1	1	OLD	Default	ORIG	3.28
2	12090,155	c1	F	NEW	Default	ORIG	3.2
3	12090,155	c1	F	NEW	BaseAlt	2ALLRED	2.13
	rack	**run**	**Nucleus**	**Dim.**	**NNZ**	**LDim.**	**LNNZ**
	r16	1,2,3	A11Z3N8(12)	9,183,646,229	14,383,837,094,961	51,308,182	901,414,818
run	**Procs,n_d**	**Mode**	**D.**	**Code**	**Mapping**	**MPI**	**100Avg**
1	16289,179	c1	1	OLD	Default	ORIG	4.74
2	16110,179	c1	T	NEW	Default	ORIG	5.16
3	16110,179	c1	T	NEW	BaseAlt	2ALLRED	3.11
	rack	**run**	**Nucleus**	**Dim.**	**NNZ**	**LDim.**	**LNNZ**
	r16	1,2,3	A08Z4N4(14)	6,770,401,490	17,131,447,517,484	37,829,602	1,075,912,865
		4	A08Z4N4(14)	6,770,401,490	17,131,447,517,484	26,557,713	531,524,652
run	**Procs,n_d**	**Mode**	**D.**	**Code**	**Mapping**	**MPI**	**100Avg**
1	16289,179	c1	1	OLD	Default	ORIG	3.48
2	16110,179	c1	T	NEW	Default	ORIG	3.84
3	16110,179	c1	T	NEW	BaseAlt	2ALLRED	2.46
4	32640,255	c2	T	NEW	BaseAlt	ALLRED	3.33
	rack	**run**	**Nucleus**	**Dim.**	**NNZ**	**LDim.**	**LNNZ**
	r24	1,2	A10Z5N5(12)	13,622,481,722	25,101,244,695,035	61,642,528	1,032,398,385
run	**Procs,n_d**	**Mode**	**D.**	**Code**	**Mapping**	**MPI**	**100Avg**
1	24531,221	c1	T	NEW	Default	ORIG	6.69
2	24531,221	c1	T	NEW	BaseAlt	2ALLRED	4.42

Continued on next page

Table A.3.1 – *Continued from previous page*

	rack	**run**	**Nucleus**	**Dim.**	**NNZ**	**LDim.**	**LNNZ**
	r32	1,2	A16Z8N8(10)	23,709,299,558	28,917,010,554,578	92,977,645	885,937,823
run	**Procs,n_d**	**Mode**	**D.**	**Code**	**Mapping**	**MPI**	**100Avg**
1	32640,255	c1	T	NEW	Default	ORIG	5.74
2	32640,255	c1	T	NEW	Base	1ALLRED	4.56
	rack	**run**	**Nucleus**	**Dim.**	**NNZ**	**LDim.**	**LNNZ**
	r32	1	A08Z2N6(16)	12,020,598,212	37,690,344,305,182	47,517,988	1,184,591,877
		2,3	A08Z2N6(16)	12,020,598,212	37,690,344,305,182	47,145,668	1,166,845,311
run	**Procs,n_d**	**Mode**	**D.**	**Code**	**Mapping**	**MPI**	**100Avg**
1	32637,253	c1	2	OLD	Default	ORIG	4.07
2	32640,255	c1	T	NEW	Default	ORIG	3.1
3	32640,255	c1	T	NEW	Base	1ALLRED	2.47

BIBLIOGRAPHY

[1] Many Fermion Dynamics nuclear. http://nuclear.physics.iastate.edu/mfd.php. Accessed: January, 2015.

[2] Nuclear Computational Low-Energy Initiative. http://nuclei.mps.ohio-state.edu. Accessed: February, 2015.

[3] The 44th Top 500 Supercomputers List. http://top500.org/lists/2014/11/. Accessed: November, 2015.

[4] The graph 500 List. http://www.graph500.org. Accessed: February, 2015.

[5] Universal Nuclear Energy Density Functional. http://www.unedf.org. Accessed: February, 2015.

[6] Yuichiro Ajima, Shinji Sumimoto, and Toshiyuki Shimizu. Tofu: A 6D Mesh/Torus Interconnect for Exascale Computers. *Computer*, 42(11):36–40, 2009.

[7] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. Optimizing Sparse Matrix-Multiple Vectors Multiplication for Nuclear Configuration Interaction Calculations. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 1213–1222, Washington, DC, USA, 2014. IEEE Computer Society.

[8] Hasan Metin Aktulga, Chao Yang, Esmond G. Ng, Pieter Maris, and James P. Vary. *Euro-Par 2012 Parallel Processing: 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings*, chapter Topology-Aware Mappings for Large-Scale Eigenvalue Problems, pages 830–842. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[9] Hasan Metin Aktulga, Chao Yang, Esmond G. Ng, Pieter Maris, and James P. Vary. Improving the scalability of a symmetric iterative eigensolver for multi-core platforms. *Concurrency and Computation: Practice and Experience*, 26(16):2631–2651, 2014.

[10] J. I. Aliaga and V. Hernandez. Symmetric sparse matrix-vector product on distributed memory multiprocessors. In *Parallel Computing and transputer applications*, 1992.

[11] Noga Alon. *LATIN'98: Theoretical Informatics: Third Latin American Symposium Campinas, Brazil, April 20–24, 1998 Proceedings*, chapter Spectral Techniques in Graph Algorithms, pages 206–215. Springer Berlin Heidelberg, Berlin, Heidelberg, 1998.

[12] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[13] Steve Ashby, Pete Beckman, Jackie Chen, Phil Colella, Bill Collins, Dona Crawford, Jack Dongarra, Doug Kothe, Rusty Lusk, Paul Messina, Tony Mezzacappa, Parviz Moin, Mike Norman, Robert Rosner, Vivek Sarkar, Andrew Siegel, Fred Streitz, Andy White, and Margaret Wright. The Opportunities and Challenges of Exascale Computing. Summary report of the advanced scientific computing advisory committee (ascac) subcommittee, U.S. Department of Energy, 2010.

[14] Brian Austin. Characterization of the Cray Aries Network, February 2014. Accessed on February, 2015.

[15] Zhaojun Bai, James Demmel, Jack Dongarra, Axel Ruhe, and Henk van der Vorst. *Templates for the solution of Algebraic Eigenvalue Problems: A Practical Guide*. SIAM, Philadelphia, 2000.

[16] Abhinav Bhatele, Todd Gamblin, Steven H. Langer, Peer-Timo Bremer, Erik W. Draeger, Bernd Hamann, Katherine E. Isaacs, Aaditya G. Landge, Joshua A. Levine, Valerio Pascucci, Martin Schulz, and Charles H. Still. Mapping Applications with Collectives over Sub-communicators on Torus Networks. In *Proceedings of the International Conference*

on High Performance Computing, Networking, Storage and Analysis, SC '12, pages 97:1–97:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[17] Abhinav Bhatele, Nikhil Jain, Katherine E. Isaacs, Ronak Buch, Todd Gamblin, Steven H. Langer, and Laxmikant V. Kale. Optimizing the performance of parallel applications on a 5D torus via task mapping. In *High Performance Computing (HiPC), 2014 21st International Conference on*, pages 1–10, Dec 2014.

[18] Shahid H. Bokhari. On the Mapping Problem. *IEEE Trans. Comput.*, 30(3):207–214, March 1981.

[19] Aydin Buluç, Jeremy T. Fineman, Matteo Frigo, John R. Gilbert, and Charles E. Leiserson. Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks. In *Proceedings of the Twenty-first Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 233–244, New York, NY, USA, 2009. ACM.

[20] Dong Chen, Noel Eisley, Philip Heidelberger, Sameer Kumar, Amith Mamidala, Fabrizio Petrini, Robert Senger, Yutaka Sugawara, Robert Walkup, Anamitra Choudhury, Yogish Sabharwal, Swaty Singhal, Burkhard Steinmacher-Burow, and Jeffrey J. Parker. Looking Under the Hood of the IBM Blue Gene/Q Network. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–12, Nov 2012.

[21] Cray. Managing System Software for Cray XE and Cray XT Systems. Online, June 2010. Accessed on February 2014.

[22] Cray. Workload Management and Application Placement for the Cray Linux Environment. Online, September 2011. Accessed on February 2014.

[23] James W. Demmel. *Applied Numerical Linear Algebra*. Society for Industrial and Applied Mathematics, 1997.

[24] Greg Faanes, Abdulla Bataineh, Duncan Roweth, Tom Court, Edwin Froese, Bob Alverson, Tim Johnson, Joe Kopnick, Mike Higgins, and James Reinhard. Cray Cascade: a Scalable HPC System based on a Dragonfly Network. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–9, Nov 2012.

[25] Maurizio Filippone, Francesco Camastra, Francesco Masulli, and Stefano Rovetta. A survey of kernel and spectral methods for clustering. *Pattern Recognition*, 41(1):176 – 190, 2008.

[26] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving problems on concurrent processors*, volume 1: General techniques and regular problems. Prentice-Hall, Inc. Englewood Cliffs, New Jersey, USA., 1988.

[27] Roman Geus and Stefan Röllin. Towards a fast parallel sparse symmetric matrixvector multiplication. *Parallel Computing*, 27(7):883 – 896, 2001. Linear systems and associated problems.

[28] Megan Gilge. IBM System Blue Gene Solution: Blue Gene/Q Application Development. IBM Redbook SG24-7948-01, 6 2013.

[29] Ahmed H. Abdel-Gawad, Mithuna Thottethodi, and Abhinav Bhatele. RAHTM: Routing-Algorithm Aware Hierarchical Task Mapping. In *Proceedings of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14. IEEE Computer Society, November 2014. LLNL-CONF-653568.

[30] Bruce Hendrickson, Robert Leland, and Steve Plimpton. An Efficient Parallel Algorithm for Matrix-Vector Multiplication. *International Journal of High Speed Computing*, 7:73–88, 1995.

[31] Torsten Hoefler, Emmanuel Jeannot, and Guillaume Mercier. *An Overview of Topology Mapping Algorithms and Techniques in High-Performance Computing*, pages 73–94. John Wiley & Sons, Inc., 2014.

[32] Nikhil Jain, Abhinav Bhatele, Michael P. Robson, Todd Gamblin, and Laxmikant V. Kale. Predicting Application Performance Using Supervised Learning on Communication Features. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 95:1–95:12, New York, NY, USA, 2013. ACM.

[33] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. Technology-Driven, Highly-Scalable Dragonfly Topology. *SIGARCH Comput. Archit. News*, 36(3):77–88, June 2008.

[34] Peter Kogge and John Shalf. Exascale Computing Trends: Adjusting to the "New Normal" for Computer Architecture. *Computing in Science and Engineering*, 15(6):16–26, 2013.

[35] Moritz Kreutzer, Georg Hager, Gerhard Wellein, Holger Fehske, Achim Basermann, and Alan R. Bishop. Sparse matrix-vector multiplication on GPGPU clusters: A new storage format and a scalable implementation. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*, IPDPSW '12, pages 1696–1702, Washington, DC, USA, 2012. IEEE Computer Society.

[36] M. Krotkiewski and M. Dabrowski. Parallel symmetric sparse matrixvector product on scalar multi-core CPUs. *Parallel Computing*, 36(4):181 – 198, 2010.

[37] Sameer Kumar, Amith R. Mamidala, Daniel A. Faraj, Brian Smith, Michael Blocksome, Bob Cernohous, Douglas Miller, Jeff Parker, Joseph Ratterman, Philip Heidelberger, Dong Chen, and Burkhard Steinmacher-Burrow. PAMI: A Parallel Active Message Interface for the Blue Gene/Q Supercomputer. *Parallel and Distributed Processing Symposium, International*, 0:763–773, 2012.

[38] John G. Lewis, David G. Payne, and Robert A. van de Geijn. Matrix-Vector Multiplication and Conjugate Gradient Algorithms on Distributed Memory Computers. In *Scalable High-Performance Computing Conference, 1994., Proceedings of the*, pages 542–550, May 1994.

[39] John G. Lewis and Robert A. van de Geijn. Distributed Memory Matrix-Vector Multiplication and Conjugate Gradient Algorithms. In *Proceedings of the 1993 ACM/IEEE*

Conference on Supercomputing, Supercomputing '93, pages 484–492, New York, NY, USA, 1993. ACM.

[40] John Gregg Lewis. *Algorithms for Sparse Matrix Eigenvalue Problems.* PhD thesis, Stanford, CA, USA, 1977. AAI7718224.

[41] Pieter Maris, Hasan Metin Aktulga, Sven Binder, Angelo Calci, Ümit V. Çatalyürek, Joachim Langhammer, Esmond Ng, Erik Saule, Robert Roth, James P. Vary, and Chao Yang. No Core CI calculations for light nuclei with chiral 2- and 3-body forces. *Journal of Physics: Conference Series*, 454(1):012063, 2013.

[42] Pieter Maris, Hasan Metin Aktulga, Mark A. Caprio, Ümit V. Çatalyürek, Esmond G. Ng, Dossay Oryspayev, Hugh Potter, Erik Saule, Masha Sosonkina, James P. Vary, Chao Yang, and Zheng Zhou. Large-scale *ab initio* configuration interaction calculations for light nuclei. *Journal of Physics: Conference Series*, 403(1):012019, 2012.

[43] Pieter Maris, James P. Vary, Angelo Calci, Joachim Langhammer, Sven Binder, and Robert Roth. ^{12}C properties with evolved chiral three-nucleon interactions. *Phys. Rev. C*, 90:014314, Jul 2014.

[44] MPI Forum. MPI: A Message Passing Interface Standard. `http://www.mpi-forum.org`.

[45] Mariá C.V. Nascimento and André C.P.L.F. de Carvalho. Spectral methods for graph clustering A survey. *European Journal of Operational Research*, 211(2):221 – 231, 2011.

[46] NERSC. Hopper's interconnect. Online. Accessed on February, 2014.

[47] NVIDIA. CUDA C programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, December 2012.

[48] Andrew T. Ogielski and William Aiello. Sparse Matrix Computations on Parallel Processor Arrays. *SIAM J. Sci. Comput.*, 14(3):519–530, May 1993.

[49] OpenMP Architecture Review Board. OpenMP Application Program Interface. `http://www.openmp.org`.

[50] Dossay Oryspayev, Hasan Metin Aktulga, Masha Sosonkina, Pieter Maris, and James P. Vary. Performance analysis of distributed symmetric sparse matrix vector multiplication algorithm for multi-core architectures. *Concurrency and Computation: Practice and Experience*, 27(17):5019–5036, 2015. cpe.3499.

[51] Dossay Oryspayev, Pieter Maris, and Joseph Zambreno. Topology-aware mapping for large-scale eigensolver on a torus network. In preparation.

[52] Dossay Oryspayev, Hugh Potter, Pieter Maris, Masha Sosonkina, James P. Vary, Sven Binder, Angelo Calci, Joachim Langhammer, and Robert Roth. Leveraging GPUs in Ab Initio Nuclear Physics Calculations. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1365–1372, May 2013.

[53] Ilya Popovyan. Efficient Parallelization of Lanczos Type Algorithms. *IACR Cryptology ePrint Archive 01/2011;2011:416*, 2011.

[54] H. D. Potter, S. Fischer, P. Maris, J. P. Vary, S. Binder, A. Calci, J. Langhammer, and R. Roth. *Ab initio* study of neutron drops with chiral Hamiltonians. *Physics Letters B*, 739:445 – 450, 2014.

[55] Hugh Potter, Dossay Oryspayev, Pieter Maris, Masha Sosonkina, James Vary, Sven Binder, Angelo Calci, Joachim Langhammer, Robert Roth, Ümit V. Çatalyürek, and Erik Saule. Accelerating *Ab Initio* Nuclear Physics Calculations with GPUs. *Proceedings of Nuclear Theory in the Supercomputing Era-2013 (NTSE-2013)*, pages 263 – 272, May 2013.

[56] Michael J. Quinn. *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Higher Education, The McGraw-Hill Companies, Inc., 2004.

[57] Robert Roth, Angelo Calci, Joachim Langhammer, and Sven Binder. Evolved chiral $NN+3N$ Hamiltonians for *ab initio* nuclear structure calculations. *Phys. Rev. C*, 90:024325, Aug 2014.

[58] Robert Roth, Joachim Langhammer, Angelo Calci, Sven Binder, and Petr Navrátil. Similarity-Transformed Chiral $NN + 3N$ Interactions for the *Ab Initio* Description of ^{12}**C** and ^{16}**O**. *Phys. Rev. Lett.*, 107:072501, Aug 2011.

[59] Yousef Saad. *Iterative methods for Sparse Linear Systems.* PWS Publishing Company, a division of International Thomson Publishing Inc., 1996.

[60] Gerald Schubert, Holger Fehske, Georg Hager, and Gerhard Wellein. Hybrid-parallel sparse matrix-vector multiplication with explicit communication overlap on current multicore-based systems. *Parallel Processing Letters*, 21(03):339–358, 2011.

[61] Avinash Srinivasa and Masha Sosonkina. Nonuniform Memory Affinity Strategy in Multithreaded Sparse Matrix Computations. In *Proceedings of the 2012 Symposium on High Performance Computing*, HPC '12, pages 9:1–9:8, San Diego, CA, USA, 2012. Society for Computer Simulation International.

[62] Philip Sternberg, Esmond G. Ng, Chao Yang, Pieter Maris, James P. Vary, Masha Sosonkina, and Hung Viet Le. Accelerating Configuration Interaction Calculations for Nuclear Structure. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, pages 15:1–15:12, Piscataway, NJ, USA, 2008. IEEE Press.

[63] Rajeev Thakur, Rolf Rabenseifner, and William Gropp. Optimization of Collective Communication Operations in MPICH. *International Journal of High Performance Computing Applications*, 19(1):49–66, February 2005.

[64] U.S.A. Department of Energy. Scientific Discovery through Advanced Computing. http://www.scidac.gov/aboutSD.html.

[65] James P Vary, Pieter Maris, Esmond Ng, Chao Yang, and Masha Sosonkina. *Ab initio* nuclear structure the large sparse matrix eigenvalue problem. *Journal of Physics: Conference Series*, 180(1):012083, 2009.

[66] James P. Vary and D. C. Zhang. The Many-Fermion-Dynamics Shell-Model Code. Unpublished, 1992.

[67] Zheng Zhou, Erik Saule, Hasan Metin Aktulga, Chao Yang, Esmond G. Ng, Pieter Maris, James P. Vary, and Ümit V. Çatalyürek. An Out-of-core Eigensolver on SSD-equipped Clusters. In *Cluster Computing (CLUSTER), 2012 IEEE International Conference on*, pages 248–256, Sept 2012.