OPTIMIZATION OF LARGE SCALE ITERATIVE EIGENSOLVERS

By

Md Afibuzzaman

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

Computer Science - Doctor of Philosophy

2021

ABSTRACT

OPTIMIZATION OF LARGE SCALE ITERATIVE EIGENSOLVERS

By

Md Afibuzzaman

Sparse matrix computations, in the form of solvers for systems of linear equations, eigenvalue problem or matrix factorizations constitute the main kernel in problems from fields as diverse as computational fluid dynamics, quantum many body problems, machine learning and graph analytics. Iterative eigensolvers have been preferred over the regular method because the regular method not being feasible with industrial sized matrices. Although dense linear algebra libraries like BLAS, LAPACK, SCALAPACK are well established and some vendor optimized implementation like mkl from Intel or Cray Libsci exist, it is not the same case for sparse linear algebra which is lagging far behind. The main reason behind slow progress in the standardization of sparse linear algebra or library development is the different forms and properties depending on the application area. It is worsened for deep memory hierarchies of modern architectures due to low arithmetic intensities and memory bound computations. Minimization of data movement and fast access to the matrix are critical in this case. Since the current technology is driven by deep memory architectures where we get the increased capacity at the expense of increased latency and decreased bandwidth when we go further from the processors. The key to achieve high performance in sparse matrix computations in deep memory hierarchy is to minimize data movement across layers of the memory and overlap data movement with computations. My thesis work contributes towards addressing the algorithmic challenges and developing a computational infrastructure to achieve high performance in scientific applications for both shared memory and distributed memory architectures. For this purpose, I started working on optimizing a blocked eigensolver and

optimized specific computational kernels which uses a new storage format. Using this optimization as a building block, we introduce a shared memory task parallel framework focusing on optimizing the entire solvers rather than a specific kernel. Before extending this shared memory implementation to a distributed memory architecture, I simulated the communication pattern and overheads of a large scale distributed memory application and then I introduce the communication tasks in the framework to overlap communication and computation. Additionally, I also tried to find a custom scheduler for the tasks using a graph partitioner. To get acquainted with high performance computing and parallel libraries, I started my PhD journey with optimizing a DFT code named Sky3D where I used dense matrix libraries. Despite there might not be any single solution for this problem, I tried to find an optimized solution. Though the large distributed memory application MFDn is kind of the driver project of the thesis, but the framework we developed is not confined to MFDn only, rather it can be used for other scientific applications too. The output of this thesis is the task parallel HPC infrastructure that we envisioned for both shared and distributed memory architectures.

I would like to dedicate this thesis to my parents Md. Abdul Motin and Afrin Sultana, my wife Mayeesha Farzana and my Son Taaif Abdullah for supporting me endlessly towards achieving this goal.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Hasan Metin Aktulga who has supported me throughout my PhD career with research grants and enriched my knowledge with innovative ideas. From the very beginning he has taught me the fundamentals of research. He made constructive criticisms throughout my PhD journey and also motivated me to keep going when things were not going as I expected. I would also like to thank my friends in Michigan State University who made my time enjoyable and did not let me fall victim to research related fatigue. I sincerely thank my group mates for their ideas and support. My parents and brother Md Ahiduzzaman always encouraged me and supported me living in a different country. My son Taaif who was born in the final year of my PhD motivated me to push my work with his amazing smile. And most importantly my wife Mayeesha deserves the most credit with her constant support and presence in my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ALGORITHMS

Chapter 1

INTRODUCTION AND MOTIVATION

1.1 Background And Related Work

Eigenvalue calculation is one of the most important parts in application of numerical linear algebra. Almost all kinds of scientific research whether it is nuclear astrophysics or molecular dynamics, whether this is applied machine learning or life science, calculating eigenvalues often becomes one of the primary requirements theoretically. Also with the emerging of Artificial Intelligence and Machine learning in the current computing world, eigenvalue calculations plays a big part. The naive way to find the eigenvalues of a matrix is to find all the roots of the characteristic polynomial of the matrix. But in large scale analysis where the matrix dimensions are in thousands or millions, this is fairly impractical to find the eigenvalues in such a way. Hence a number of iterative algorithms have been developed over the years. These methods work by repeatedly refining approximations to the eigenvectors or eigenvalues, and can be terminated whenever the approximations reach a suitable degree of accuracy. Iterative methods form the basis of much of modern day eigenvalue computation.

Since a graph represented in an adjacency matrix format is essentially a sparse matrix, graph algorithms can also be expressed in the language of sparse linear algebra. In fact, recent studies have shown that graph algorithms expressed in this way achieve significantly better

performance than alternative abstractions [6]. To simplify the presentation, we use a unified nomenclature for both, and use the term sparse matrix to also refer to the network structure in a graph. The term nonzeros will refer to the nonzero matrix elements in sparse matrices or edge meta-data (e.g., weights) in graphs. Finally, we overload the term vector in sparse matrix computations to also refer to the array of vertex properties in graph computations. Most fundamental operation in sparse linear algebra is thought to be the multiplication of a sparse matrix with a vector (SpMV), as it forms the main computational kernel for several applications (e.g., the solution of partial differential equations (PDE) [7] and the Schrodinger Equation [8] in scientific computing, spectral clustering [9] and dimensionality reduction [10] in machine learning, and the Page Rank algorithm [11] in graph analytics). The Roofine model by Williams et al. [12] suggests that the performance of SpMV kernel is ultimately bounded by the memory bandwidth. Consequently, performance optimizations to increase cache utilization and reduce data access latencies for SpMV has drawn significant interest [13, 14, 15, 16, 17, 18, 19, 20, 21], which is a rather incomplete list of related work on this topic.

A closely related kernel is the multiplication of a sparse matrix with multiple vectors (SpMM) which constitutes the main operation in block solvers, e.g., the block Krylov subspace methods and block Jacobi-Davidson method. SpMM has much higher arithmetic intensity than SpMV and can effciently leverage wide vector execution units. As a result, SpMM-based solvers has recently drawn significant interest in scientific computing [22, 23, 24, 25, 26]. SpMM also finds applications naturally in machine learning where several features (or eigenvectors) of sparse matrices are needed [10, 9]. Although SpMM has a significantly higher arithmetic intensity than SpMM, the extended Roofline model that we recently proposed suggests that cache bandwidth, rather than the memory bandwidth, can

still be an important performance limiting factor for SpMM [22]. Multiplication of sparse matrices (SpGEMM) and sparse matrix times sparse vector (SpMSV) operation also find applications in important problems. SpGEMM is the main kernel in the algebraic multi-grid method [27], and the Markov Clustering algorithm, while SpMSV is the main building block for breadth-first search, bipartite graph matching, and maximal independent set algorithms.

1.2 Emergence Of Deep Memory Hierarchies

Given the widening gap between memory system and processor performance [28], irregular data access patterns and low arithmetic intensities of sparse matrix computations have effectively made them "memory-bound" computations. Furthermore, the downward trend in memory space and bandwidth per core in high performance computing (HPC) systems [29] has paved the way for a deepening memory hierarchy. For example, many-core processors (i.e., GPUs and Xeon Phis) have their own high-bandwidth (but limted size) device memories (HBM). NVRAM storages have recently emerged to alleviate issues (such as cost, capacity, energy consumption and resiliency) associated with the DRAM technology. Consequently, ash memory and 3D-XPoint memory have already found wide adoption as a storage-class cache between DRAM and disk systems, and they are being adopted as memory-class storages complementing DRAM in modern HPC systems .

In Fig. 1.1, we give an abstract view of the assumed underlying memory hierarchy, along with some hardware specifications based on current technology. Our target architectures are many-core processors such as GPUs and Xeon Phis, which are essentially the cornerstones of big data analytics and scientific computing. While exact specifications and number of layers change as architectures evolves, the underlying principle of memory hierarachy stays

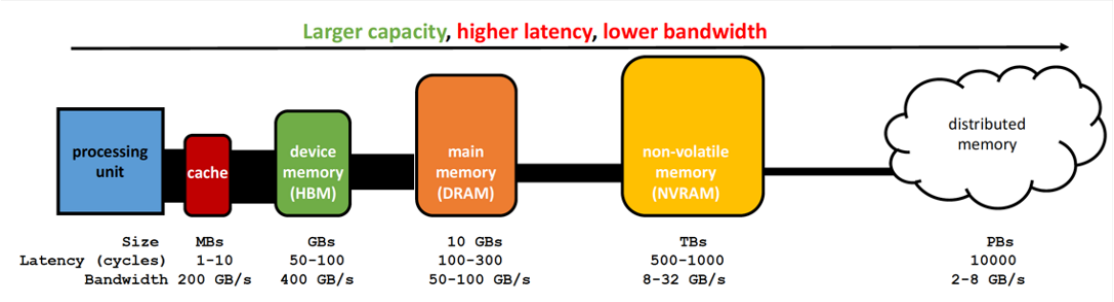


Figure 1.1: Memory hierarchy in deep memory architectures

the same: Going further away from the processor, memory capacity increases at the expense of increased latency and reduced bandwidth. Thus, minimizing data movement across layers of the memory and overlapping data movement with computations are keys to achieving high performance in sparse matrix computations. While we mainly focus on parallelism and performance on a single node, developed techniques and software will be complementary to those aimed at enabling distributed memory parallelism for sparse matrix computations and graphs (such as Par-METIS, Xtra- PuLP, etc.), or manual partitionings specified by a user. Hence, the footprint of applications that can benefit from this project will be significant.

In data analytics and scientific computing, total available memory is often a limiting factor. Hence, data management is an important due to both the size of the data involved and the complexity of the program ow. As an example, in Fig. 3, we give the pseudocode for the locally optimal block preconditioned conjugate gradient algorithm (LOBPCG), a widely used block eigensolver [30]. SpMM of the sparse matrix H and block vector , despite being an expensive step, is only one part of the computation in line 4. In terms of memory, while the H matrix takes up considerable space, when a large number of eigenpairs are needed (e.g. dimensionality reduction, spectral clustering or quantum many-body problems), memory needed for block vector can be comparable to or even greater than that of H . In addition, other block vectors (residual R, preconditioned residual W, previous direction P), block

vectors from the previous iteration and the preconditioning matrix T (if available) must be stored, and accessed at each iteration. Clearly, orchestrating the data movement in a deep memory hierarchy to attain an efficient implementation can become a daunting task for a domain scientist.

In fact, considering the complete range of solvers that sparse matrix computations arise in, LOBPCG algorithm is a relatively simple case. For instance, when performing Singular Value Decompositions (SVD), each node would need to perform operations pertaining also to the transpose MT (or complex conjugate M) matrix of the sparse partition that they own [38], which must be applied on the result of the application of M over the source vector. The interior eigenvalue problem is another example illustrating complexities in a real application. An effective way to accelerate convergence to eigenpairs in a desired range is to build polynomial filters [65] (e.g., [a3M3 + a2M2 + a1M]x is a 3rd order matrix polynomial), which require several applications of a sparse matrix over the source vector in each iteration. Finally, an SpGEMM kernel or sparse LU factorization represent significantly more complex computations, as computations associated with nonzeros form a complex dependency graph in these cases. We propose a comprehensive framework that can effciently handle complex and irregular (due to sparsity) task depedencies arising in a wide variety of applications. In addition to conventional applications involving static sparse matrices, we envision our framework to be generic enough to support incremental algorithms used to to tackle streaming (or online) problems [34, 69, 93]. Such applications are common in data analytics, e.g., to analyze dynamic web graphs or social networks, or incrementally incorporate user feedback.

For our work first we target an application called Sky3d which works on a dense matrix. Then we shift our interest to sparse matrix iterative eigensolvers. We start with the Sky3d

application which is a density functional theory approach to study different kinds of nuclear shapes. After a successful exploration of dense matrix linear algebra libraries we worked on a sparse matrix iterative eigensolver code named MFDn. In that project we optimized the sparse matrix multiple vector kernels using a new storage format for sparse matrices. Taking that experience, we developed a shared memory framework for iterative eigensolvers using task parallelism. At first we used a default scheduler from OpenMP libraries but we also wanted to have our own schedule for tasks. We studied some graph partitioners and created custom partitions and task orders for execution. We wanted to extend the framework for a distributed solver like MFDn. Hence we first studied the communication behavior of MFDn using a simulator called SST. Looking at the observations, we introduced communication tasks in our framework which overlaps the communication and computations. Now I will present some motivations, background studies for all the projects.

1.3 Optimizing Sky3D

Compact objects in space such as neutron stars are great test laboratories for nuclear physics as they contain all kinds of exotic nuclear matter which are not accessible in experiments on earth [31, 32]. Among interesting kinds of astromaterial is the so called nuclear "pasta" phase [33, 34] which consists of neutrons and protons embedded in an electron gas. The naming arises from the shapes, e.g. rods and slabs, which resemble the shapes of the Italian pasta (spaghetti and lasagna).

Nuclear pasta is expected in the inner crust of neutron stars in a layer of about 100 m at a radius of about 10 km, at sub-nuclear densities. Since the typical length scale of a neutron or a proton is on the order of 1 fm, it is impossible to simulate the entire system. The usual

strategy is to simulate a small part of the system in a finite box with periodic boundary conditions. While it is feasible to perform large simulations with semi-classical methods such as Molecular Dynamics (MD) [35, 36, 37, 38, 39, 40, 41] involving 50,000 or even more particles or the Thomas-Fermi approximation [42, 43, 44, 45], quantum mechanical (QM) methods which can yield high fidelity results have been limited to about 1000 nucleons due to their immense computational costs [46, 47, 48, 49, 50, 51, 52, 53].

The side effects of using small boxes in QM methods are twofold: First, the finite size of the box causes finite-volume effects, which have an observable influence on the results of a calculation. Those effects have been studied and can be suppressed by introducing the twist-averaged boundary conditions [51]. More importantly though, finite boxes limit the possible resulting shapes of the nuclear pasta because the unit cell of certain shapes might be larger than the maximum box size. For instance, in MD simulations [54], slabs with certain defects have been discovered. Those have not been observed in QM simulations because they only manifest themselves in large boxes. To observe such defects, we estimate that it is necessary to increase the number of simulated particles (and the corresponding simulation box volume) by about an order of magnitude.

In this work, we focus on the microscopic nuclear density functional theory (DFT) approach to study nuclear pasta formations. The nuclear DFT approach is a particularly good choice for nuclear pasta. The most attractive property is the reliability of its answers over the whole nuclear chart [55, 56], and yet it is computationally feasible for applications involving the heaviest nuclei and even nuclear pasta matter, because the interaction is expressed through one-body densities and the explicit n-body interactions do not have to be evaluated.

In contrast to finite nuclei that are usually calculated employing a harmonic oscillator finite-range basis [57, 58] using mostly complete diagonalization of the basis functions to solve

the self-consistent equations, nuclear pasta matter calculations have to be performed in a suitable basis with an infinite range. We use the DFT code Sky3D [59], which represents all functions on an equidistant grid and employs the damped gradient iteration steps, where fast Fourier transforms (FFTs) are used for derivatives, to reach a self-consistent solution. This code is relatively fast compared to its alternatives and incorporates all features necessary to perform DFT calculations with modern functionals, such as the Skyrme functionals (as used here). Since Sky3D can be used to study static and time-dependent systems in 3d without any symmetry restrictions, it can be applied to a wide range of problems. In the static domain it has been used to describe a highly excited torus configuration of ^{40}Ca [60] and also finite nuclei in a strong magnetic field as present in neutron stars [61]. In the time-dependent context, it was used for calculations on nuclear giant resonances [62, 63], and on the spin excitation in nuclear reactions [64]. The Wigner function, a 6 dimensional distribution function, and numerical conservation properties in the time-dependent domain have also been studied using Sky3D [65, 66].

For the case of time-dependent problems, the Sky3D code has already been parallelized using MPI. The time-dependent iterations are simpler to parallelize, because the treatment of the single particles are independent of each other and can be distributed among the nodes. Only the mean field has to be communicated among the computational nodes. On the other hand, accurate computation of nuclear ground states, which we are interested in, requires a careful problem decomposition strategy and organization of the communication and computation operations as discussed below. However, only a shared memory parallel version of Sky3D (using OpenMP) exists to this date. In this work, we present algorithms and techniques to achieve scalable distributed memory parallelism in Sky3D using MPI.

1.4 Optimization Of Blocked Eigensolver For Sparse Matrix

We found out that for dense matrices there are some very optimized numerical libraries for distributed memory. Being a relatively well studied area, these libraries have been serving the scientific community for quite a while. In our work we observed strong scaling using the highly optimized ScLAPACK library. But we also noticed that for sparse matrices there are a lot of room for improvement. The choice of numerical algorithms and how efficiently they can be implemented on high performance computer (HPC) systems critically affect the time-to-solution for large-scale scientific applications. Several new numerical techniques or adaptations of existing ones that can better leverage the massive parallelism available on modern systems have been developed over the years. Although these algorithms may have slower convergence rates, their high degree of parallelism may lead to better time-to-solution on modern hardware [24]. In the next work, we consider the solution of the quantum many-body problem using the configuration interaction (CI) formulation. We present algorithms and techniques to significantly speed up eigenvalue computations in CI by using a block eigensolver and optimizing the key computational kernels involved.

The quantum many-body problem transcends several areas of physics and chemistry. The CI method enables computing the wave functions associated with discrete energy levels of these many-body systems with high accuracy. Since only a small number of low energy states are typically needed to compute the physical observables of interest, a partial diagonalization of the large CI many-body Hamiltonian is sufficient.

More formally, we are interested in finding a small number of extreme eigenpairs of a

large, sparse, symmetric matrix:

$$x_i = \lambda x_i, \quad i = 1, \ldots, m, \quad m \ll N. \tag{1.1}$$

Iterative methods such as the Lanczos and Jacobi–Davidson [67] algorithms, as well as their variants [68, 69, 70], can be used for this purpose. The key kernels for these methods can be crudely summarized as (repeated) sparse matrix–vector multiplications (SpMV) and orthonormalization of vectors (level-1 BLAS). As alternatives, block versions of these algorithms have been developed [71, 72, 73] which improves the arithmetic intensity of computations at the cost of a reduced convergence rate and increased total number of matrix–vector operations [74]. In block methods, SpMV becomes a sparse matrix multiple vector multiplication (SpMM) and vector operations become level-3 BLAS operations.

Performance of SpMV is ultimately bounded by memory bandwidth [75]. The widening gap between processor performance and memory bandwidth significantly limits the achievable performance in several important applications. On the other hand, in SpMM, one can make use of the increased data locality in the vector block and attain much higher FLOP rates on modern architectures. Gropp et al. was the first to exploit this idea by using multiple right hand sides for SpMV in a computational fluid dynamics application [23]. SpMM is one of the core operations supported by the auto-tuned sequential sparse matrix library OSKI [20]. OSKI's shared memory parallel successor, pOSKI, currently does not support SpMM [76]. More recently, Liu et al. [24] investigated strategies to improve the performance of SpMM [1] using SIMD (AVX/SSE) instructions on modern multicore CPUs. Their driv-

[1] Liu et al. actually uses the name GSpMV for "generalized" SpMV. We refrain from doing so because the same name has been used in conflicting contexts such as SpMV for graph algorithms where the scalar operations can be arbitrarily overloaded.

ing application is the motion simulation of biological macromolecules in solvent using the Stokesian dynamics method. Röhrig-Zöllner et al. [25] discuss performance optimization techniques for the block Jacobi–Davidson method to compute a few eigenpairs of large-scale sparse matrices, and report reduced time-to-solution using block methods instead of single vector counterparts, particularly for problems in quantum mechanics and PDEs.

Our work differs from previous efforts substantially, in part due to the immense size of the sparse matrices involved. We exploit symmetry to reduce the overall memory footprint, and offer an efficient solution to perform SpMM on a sparse matrix and its transpose (SpMM^T) with roughly the same performance [22]. This is achieved through a novel thread parallel SpMM implementation, CSB/OpenMP, which is based on the compressed sparse block (CSB) framework [77] (Sect. 3.3). We demonstrate the efficiency of CSB/OpenMP on a series of CI matrices where we obtain 3–4× speedup over the commonly used compressed sparse row (CSR) format. To estimate the performance characteristics and better understand the bottlenecks of the SpMM kernel, we propose an extended Roofline model to account for cache bandwidth limitations (Sect. 3.3).

In this work, we extend a previous work (presented in [22]) by considering an end-to-end optimization of a block eigensolver implementation. As will be discussed in Sect. 3.7, the performance of the tall-skinny matrix operations in block eigensolvers is critical for an excellent overall performance. We observe that the implementations of these level-3 BLAS operations in optimized math libraries perform significantly below expectations for typical matrix sizes encountered in block eigensolvers. We propose a highly efficient thread parallel implementation for inner product and linear combination operations that involve tall-skinny matrices and analyze the resulting performance.

To demonstrate the merits of the proposed techniques, we incorporate the CSB/OpenMP

implementation of SpMM and optimized tall-skinny matrix kernels into a LOBPCG [73] based solver in MFDn, an advanced nuclear CI code [2, 78, 3]. We demonstrate through experiments with real-world problems that the resulting block eigensolver can outperform the widely used Lanczos algorithm (based on single vector iterations) with modern multicore architectures (Sect. 3.8.8). We also analyze the performance of our techniques on an Intel Xeon Phi Knights Corner (KNC) processor to assess the feasibility of our implementations for future architectures.

While we focus on nuclear CI computations, the impact of optimizing the performance of key kernels in block iterative solvers is broader. For example, spectral clustering, one of the most promising clustering techniques, uses eigenvectors associated with the smallest eigenvalues of the Laplacian of the data similarity matrix to cluster vertices in large symmetric graphs [79, 80]. Due to the size of the graphs, it is desirable to exploit the symmetry, and for a k-way clustering problem, k eigenvectors are needed, where typically $10 \leq k \leq 100$, an ideal range for block eigensolvers. Block methods are also used in solving large-scale sparse singular value decomposition (SVD) problems [81], with most popular methods being the subspace iteration and block Lanzcos. SVDs are critical for dimensionality reduction in applications like latent semantic indexing [82]. In SVD, singular values are obtained by solving the associated symmetric eigenproblem that requires subsequent SpMM and SpMM^T computations in each iteration [83]. Thus, our techniques are expected to have a positive impact on the adoption of block solvers in closely related applications.

1.5 Inrtroducing the DeepSparse Framework

Sparse matrix computations, in the form of solvers for systems of equations, eigenvalue problems or matrix factorizations, constitute the main kernel in fields as diverse as computational fluid dynamics (CFD), quantum many-body problems, machine learning and graph analytics. The scale of problems in these scientific applications typically necessitates execution on massively parallel architectures. Moreover, sparse matrices come in very different forms and properties depending on application area. However, due to the irregular data access patterns and low arithmetic intensities of sparse matrix computations, achieving high performance and scalability is very difficult. These challenges are further exacerbated by the increasingly complex deep memory hierarchies of the modern architectures as they typically integrate several layers of memory storage. While exact specifications and number of layers change as architectures evolve, the underlying principle of memory hierarchy stays the same: Going farther away from the processor, memory capacity increases at the expense of increased latency and reduced bandwidth. As such, minimizing data movement across layers of the memory and overlapping data movement with computations are keys to achieving high performance in sparse matrix computations.

Unlike its dense matrix analogue, the state of the art for sparse matrix computations is lagging far behind. The widening gap between the memory system and processor performance, irregular data access patterns and low arithmetic intensities of sparse matrix computations have effectively made them "memory-bound" computations. Furthermore, the downward trend in memory space and bandwidth per core in high performance computing (HPC) systems [29] has paved the way for a deepening memory hierarchy. Thus, there is a dire need for new approaches both at the algorithmic and runtime system levels for sparse

matrix computations.

In this work, we propose a novel sparse linear algebra framework, named *DeepSparse*, which aims to accelerate sparse solver codes on modern architectures with deep memory hierarchies. Our proposed framework differs from existing work in two ways. First, we propose a holistic approach that targets all computational steps in a sparse solver rather than narrowing the problem into a single kernel, e.g., sparse matrix vector multiplication (SpMV) or sparse matrix multiple vector multiplication (SpMM). Second, we adopt a fully integrated task-parallel approach while utilizing commonly used sparse matrix storage schemes.

In a nutshell, DeepSparse provides a GraphBLAS plus BLAS/LAPACK-like frontend for domain scientists to express their algorithms without having to worry about the architectural details (e.g., memory hierarchy) and parallelization considerations (i.e., determining the individual tasks and their scheduling) [84, 85, 86]. DeepSparse automatically generates and expresses the entire computation as a task dependency graph (TDG) where each node corresponds to a specific part of a computational kernel and edges denote control and data dependencies between computational tasks. We chose to build DeepSparse on top of OpenMP [87] because OpenMP is the most commonly used shared memory programming model, but more importantly it supports task-based data-flow programming abstraction. As such, DeepSparse relies on OpenMP for parallel execution of the TDG.

We anticipate two main advantages of DeepSparse over a conventional bulk synchronous parallel (BSP) approach where each kernel relies on loop parallelization and is optimized independently. First, DeepSparse would be able to expose better parallelism as it creates a global task graph for the entire sparse solver code. Second, since the OpenMP runtime system has explicit knowledge about the TDG, it may be possible to leverage a pipelined execution of tasks that have data dependencies, thereby leading to better utilization of the

hardware cache.

1.6 Exploring Custom Schedule For Tasks Using Graph Partition

In the DeepSparse framework we generate a global task graph. In our executor we use appropriate OpenMP task dependencies with proper memory offsets and sizes to make the task parallel implementation coherent across the entire iteration. OpenMP looks at the in-out dependencies and generates a direct acyclic graph underneath after solving those dependencies. In case of a dependency of a task gets resolved, that task is(or can be) pulled from the task pool by OpenMP engine.

Although we saw that OpenMP does a great job with memory utilization over all level of memories, it is still beyond our control. OpenMP is generating the DAG itself and resolving themselves. Whenever the data dependencies of a task is resolved and it is not dependent on any other task for its execution , it can be immediately pulled and executed. But this might not be optimal scenario if we think from memory usage perspective. A task which does not have any relation with the tasks that are active at the moment can be immediately executed once a thread gets free regardless of its memory input and outputs. Hence there is a possibility of a task which would improve the memory usage with the input already being in the lower level of the memory and having cache hits reduces. The probability of cache misses increases with this kind of scheduling.

This motivated us to use a novel graph partition based schedulers that use the global data flow graphs generated by the PCU to minimize data movements in a deep memory hierarchy. Graph partitioners have been extensively studied but existing approaches do not meet our

needs as they typically handle DAGs by converting them to undirected graphs. However, the directed nature of the task graph must be respected in our case. In this regard, acyclic partitioning heuristics for DAGs, recently introduced by Dr. Catalyurek's group, provided a great starting point.

Our sparse solver DAGs contain a fair number of vertices with high fan-in/fan-out" degrees due to operations such as SpMVs, SpMMs, inner products and vector reductions. There are two immediate issues that call for an alternative scheme. First, the presence of such vertices requires an excessive number of coarsening iterations (which rely on the edge matching" technique), and slows down partitioning to the extent of making it unusable for large graphs. Second, during refinement, the high degree fan-in/fan-out vertices effectively yield 1D partitionings, because they drag their incoming/outgoing vertices into the same partition as them. We wanted to develop a novel scheduler through the following tasks: We will adopt problem-specific coarsening/refinement techniques where vertex matchings are identified by recursively doubling the CSB block (in 2D) or blockrow (in 1D) dimensions. This would allow to preserve the original DAG structure at the coarsest level, while enabling the partitioning of extremely large DAGs.

Like other graph partitioners, the objective function for the acyclic heuristics of [44] is minimization of the edge cut (defined as the sum of all edges crossing partitions). However, for our purposes, each partition is an execution phase" denoting the set of tasks that must be brought together to the higher level memory. In fact, maximizing the edge cut between successive execution phases would be desirable in this context, because edge cuts would correspond to sharing of input data between successive stages or reuse of output from one stage as input in the subsequent stage. In our heuristics, the objective function will take into account the ordering among partitions (which is not a consideration at all for regular

graph partitioners) and try to maximize the (cumulative) size of input/output data overlaps between successive phases (so that the overall data movement is reduced).

Constraints in graph partitioners are generally geared towards ensuring load balance among partitions. However, in our case, the constraint would be the storage limit of the fast memory which the Scheduler will impose by estimating the maximum active memory size" during the execution of a phase. This is different from, but in the worst case equal to, the sum of edge weights in a partition. We anticipate that such memory constraints in conjunction with the above described objective functions will enable our Scheduler to discover better partitionings than those we can find with the techniques of (for instance, 2D-shaped partitionings are known to yield better data locality compared to the 1D-like partitionings.

To facilitate execution in a deep memory architecture, we designed the Scheduler to be hierarchical. This can, for example, be achieved by recursively applying our partitioning algorithms.

To support incremental algorithms for streaming/online problems, the Scheduler will be dynamic, i.e., it will be able to make real-time decisions regarding the placement of new tasks for incoming data and removal of tasks for deleted data. This can simply be achieved by greedily placing new tasks or deleting old ones to ensure real-time response, and periodically recreating the entire schedule to avoid suboptimal performance that may be caused by several greedy decisions made consecutively.

Note that during execution of a given phase, depending on the available fast memory space, it is possible to start loading the input data for the subsequent phase. This way, the Scheduler can overlap the (already minimized) data movement with computations for improved performance.

The DAG structure of our data- flow execution model greatly facilitates the scheduling of independent tasks to available cores on a node. In fact, scheduling heuristics can readily be used for determining the assignment of tasks to individual cores to maximize cache locality.

1.7 Simulating Communication Behavior Of A Real World Distributed Application

Large-scale real-world scientific applications typically need high-performance computing (HPC) platforms 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. Often, one needs to store large amounts of data in main memory, which requires the use of a large number of nodes; with communication between the nodes over high-speed interconnection networks. Typically, the amount of data as well as the number of nodes increases as the size of the simulations increases. Communication between nodes can, and often will, become a bottleneck, in particular for iterative sparse eigensolvers due to their low arithmetic intensities.

When preparing and optimizing scientific applications for specific HPC platforms one therefore has to take into consideration the potential communication overhead. However, it is far from trivial to estimate the actual communication overhead based on HPC design specifications such as peak bisection bandwidth, network topology, individual node and link bandwidths, latencies, among others. On existing HPC platforms, one can in principle run a skeleton of the scientific application code, simulating only the communication, and thus empirically measure the communication overhead. Unfortunately, this tends to be computationally expensive, and only possible if one has access to the specific HPC platforms.

For a future machine, one would have to wait until it is deployed and one has gained access to the system, before being able to realistically measure the actual communication overhead. Ideally, one would like to gain insight in the communication overhead without actually performing such a skeleton run of just the communication.

In this work we use the Ember library, which is part of the Structural Simulation Toolkit (SST) [88], to model the communication costs of a large-scale nuclear physics application [89, 90, 4, 91] on a current (for validation) and a future HPC platform (for prediction). This application uses iterative distributed eigensolvers (specifically, the Lanczos [92] and LOBPCG [4, 93] algorithms) to obtain the lowest eigenvalues and eigenvectors of a large sparse symmetric matrix, and runs on tens of thousands of nodes. It is known that for large-scale runs, the communication can indeed become a bottleneck; most of the communication cost however can be hidden behind local computation. We first compare our SST motif with timings from communication skeleton runs of our application on up to a thousand nodes on Cori-KNL, which is a Knights Landing based cluster at the National Energy Research Scientific Computing Center (NERSC), followed by simulations aimed at Perlmutter, which is a new machine to be installed at the same facility later this fall.

1.8 Introducing Communication for DeepSparse

In this work we implemented two different algorithms Lanczos and LOBPCG algorithms used executed them using our DeepSparse framework. The implementation is based on task parallelism and was an on node optimization. We observed that DeepSparse achieves 2× - 16× fewer cache misses across different cache layers (L1, L2 and L3) over implementations of the same solvers based on optimized library function calls. We also achieve 2× - 3.9×

improvement in execution time when using DeepSparse over the same library versions.

As we discussed in the previous Chapter 6 about the communication pattern in MFDn while doing the distributed matrix multiplication. We noticed that we have an allgather, a broadcast, a reduction and a reduce scatter operation in the MFDn code. The detailed explanation is given in Section 6.2 in Chapter 6. In practice, it is seen that for MFDn, when run in an architecture like knl, the communication usually takes over the computation for a very large simulation consisting of a very high number of mpi ranks involved from a lot of compute nodes. We simulated the performance of the communication patterns and tried to find out a possible cause using a simulator named SST.

We observed that, the broadcast operation takes a much longer time in real life because of possible network congestion and the messages being very large in size also fuels into this behavior. Since our deepsparse framework is a task based parallel framework where each task performs a particular matrix or vector operation on a matrix or a vector block, we were motivated to use blocked communication tasks.

Our motivation was to introduce custom communication tasks where each communication tasks will communicate with other nodes and only transmit a block of the matrix or a vector between themselves. This will help us in multiple ways. Being blocked communication will reduce the size of the messages during the communication much less than the actual code which we expected will help in case of network congestion. The other motivation was to overlap the communication with the computations in the matrix multiplication. Whenever a particular block is is received or ready to compute, the other kernels waiting for this particular block of matrix or a vector can start immediately rather than waiting for the entire matrix or vector to be transmitted and then

In the shared memory implementation of DeepSparse we observed a nice pipelined execu-

tion of different kinds of kernels. The matrix multiplication SpMM and the vector operations like vector vector multiplication or vector vector transpose multiplication. in Figure 4.8 we can see a pipelined execution of an actual iteration of LOBPCG where the tasks are different kernels but they use the same datastructure and ultimately improves the performance. This test was done in a haswell architecture.

We also did similar tests on Broadwell machines with a different matrix to validate the framework and the pipelined execution. In Figure 7.1 we show this pipelined execution for the nlpkkt240 matrix in a broadwell architecture. Here the SpMM is represented using orange color, XY operation is Maroon and XTY is using green color palette. We can clearly observe that the matrix and vector operations are well pipelined. Another interesting observation was that the ration of time spent on the SpMM and vector operations are somewhat in the similar range. We observed that since the matrix and vector operations are taking similar amount of time during an iteration, a well pipelined execution of these kernels will improve the cache performance which it did and we saw the execution time is actually improved in a shared memory architecture. We were motivated to extend this idea for a distributed application like MFDn which also has similar matrix and vector operations. With the introduction of communication tasks, we were motivated to use the idea from shared memory to distributed memory.

Chapter 2

OPTIMIZATION IN LARGE SCALE DISTRIBUTED DENSE MATRICES

2.1 Nuclear Density Functional Theory With Skyrme Interaction

Unlike in classical calculations where a point particle is defined by its position and its momentum, quantum particles are represented as complex wave functions. The square modulus of the wave function in real space is interpreted as a probability amplitude to find a particle at a certain point. Wave functions in the real space and the momentum space are related via the Fourier transform. In the Hartree-Fock approximation used in nuclear DFT calculations, the nuclear N-body wave function is restricted to a single Slater determinant consisting of N orthonormalized one-body wave functions ψ_α, $\alpha = 1..N$. Each of these one-body wave functions have to fulfill the one-body Schrödinger's Equation

$$\hat{h}_q \psi_\alpha = \epsilon_\alpha \psi_\alpha, \tag{2.1}$$

when convergence is reached, *i.e.*, when

$$\overline{\Delta\varepsilon} = \sqrt{\frac{\sum_\alpha \langle\psi_\alpha \mid \hat{h}^2 \mid \psi_\alpha\rangle - \langle\psi_\alpha \mid \hat{h} \mid \psi_\alpha\rangle^2}{\sum_\alpha 1}} \tag{2.2}$$

is small.

In nuclear DFT, the interaction between nucleons (*i.e.*, neutrons and protons) is expressed through a mean field. In this work, we utilize the Skyrme mean field [56]:

$$\begin{aligned}\mathcal{E}_{\mathrm{Sk}} = \sum_{q=n,p} \Big(& C_q^\rho(\rho_0)\rho_q^2 + C_q^{\Delta\rho}\rho_q\Delta\rho_q \\ & + C_q^\tau\rho_q\tau_q + C_q^{\nabla\vec{J}}\rho_q\nabla\vec{J}_q \Big),\end{aligned} \tag{2.3}$$

where the parameters C_q^i have to be fitted to experimental observables. The mean field is determined by nucleon densities and their derivatives:

$$\rho_q(\vec{r}) = \sum_{\alpha\in q}\sum_s v_\alpha^2 \mid \psi_\alpha(\vec{r},s) \mid^2 \tag{2.4a}$$

$$\vec{J}_q(\vec{r}) = -\mathrm{i}\sum_{\alpha\in q}\sum_{ss'} v_\alpha^2\psi_\alpha^*(\vec{r},s)\nabla\times\vec{\sigma}_{ss'}\psi_\alpha(\vec{r},s') \tag{2.4b}$$

$$\tau_q(\vec{r}) = \sum_{\alpha\in q}\sum_s v_\alpha^2 \mid \nabla\psi_\alpha(\vec{r},s) \mid^2 \quad , \tag{2.4c}$$

where ρ_q is the number density, $\vec{J}_q$ is the spin-orbit density and $\tau_q(\vec{r})$ is the kinetic density for $q \in$ (protons,neutrons), which are calculated from the wave functions. We assume a time-reversal symmetric state in the equations. The interaction is explicitly isospin dependent. The parameters v_α^2 are either 0 for non-occupied states or 1 for occupied states for calculations without the pairing force. With those occupation probabilities, the calculation can also be performed using more wave functions than the number of particles. The sum $\sum_\alpha v_\alpha^2$ determines the particle number. A detailed description of the Skyrme energy density

functional can be found in references [94, 95].

In DFT, the ground states associated with a many-body system is found in a self-consistent way, *i.e.*, iteratively. The self-consistent solution can be approached through the direct diagonalization method or, in this case we use the damped gradient iteration method which is described below. While stable finite nuclei as present on earth typically do not contain more than a total of 300 nucleons, nuclear pasta matter in neutron stars is quasi-infinite on the scales of quantum simulations. Therefore it is desirable to simulate as large volumes as possible to explore varieties of nuclear pasta matter. Furthermore, in contrast to finite nuclei which are approximately spherical, pasta matter covers a large range of shapes and deformations and thus many more iterations are needed to reach convergence. Since larger volumes and consequently more nucleons require very intensive calculations, a high performance implementation of nuclear DFT codes is desirable.

DFT is also widely used for electronic structure calculations in computational chemistry. While DFT approaches used in computational chemistry can efficiently diagonalize matrices associated with a large number of basis sets, we need to rely on different iteration techniques in nuclear DFT. The most important reason for this is that in computational chemistry, electrons are present in a strong external potential. Therefore, iterations can converge relatively quickly in this case. However, in nuclear DFT, the problem must be solved in a purely self-consistent manner because nuclei are self-bound. As a result, the mean field can change drastically from one iteration to the next, since no fixed outer potential is present. Especially for nuclear pasta spanning a wide range of shapes, a few thousand iterations are necessary for the solver to converge. Therefore, nuclear DFT iterations have to be performed relatively quickly, making it infeasible to employ the electronic DFT methods which are expensive for a single iteration.

2.2 Sky3D Software

Sky3D is a nuclear DFT solver, which has frequently been used for finite nuclei, as well as for nuclear pasta (for both static and time-dependent) simulations. The time-dependent version of Sky3D is relatively simpler to parallelize compared to the static version, because properties like orthornomality of the wave functions are implicitly conserved due to the fact that the time-evolution operator is unitary. Therefore the calculation of a single nucleon is independent of the others. The only interaction between nucleons takes place through the mean field. Thus only the nuclear densities using which the mean field is constructed has to be communicated. In the static case, however, orthonormality has to be ensured and the Hamiltonian matrix must be diagonalized to obtain the eigenvalues and eigenvectors of the system at each iteration. In this paper, we describe parallelization of the more challenging static version (which previously was only shared memory parallel).

Sky3D operates on a three dimensional equidistant grid in coordinate space. Since nuclear DFT is a self-consistent method requiring an iterative solver, the calculation has to be initialized with an initial configuration. In Sky3D, the wave functions are initialized with a trial state, using either the harmonic oscillator wave functions for finite nuclei or plane waves for periodic systems. The initial densities and the mean field are calculated from those trial wave functions.

After initialization, iterations are performed using the damped gradient iteration scheme [96]

$$\psi_\alpha^{(n+1)} = \mathcal{O}\left\{\psi_\alpha^{(n)} - \frac{\delta}{\hat{T}+E_0}\left(\hat{h}^{(n)} - \langle\psi_\alpha^{(n)} \mid \hat{h}^{(n)} \mid \psi_\alpha^{(n)}\rangle\right)\psi_\alpha^{(n)}\right\} \quad , \tag{2.5}$$

where $\mathcal{O}$ denotes the orthonormalization of the wave functions, $\hat{T}$ denotes the kinetic energy operator, $\psi_\alpha^{(n)}$ and $\hat{h}^{(n)}$ denote the single-particle wave function and the Hamiltonian

at step n, respectively, and δ and E_0 are constants that need to be tuned for fast convergence. The Hamiltonian consists mainly of the kinetic energy, the mean field contribution (Eq.2.3) and the Coulomb contribution. We use FFTs to compute the derivatives of the wave functions. The Coulomb problem is solved in the momentum space, also employing FFTs.

The basic flow chart of the static Sky3D code is shown in Fig 2.1. Since the damped gradient iterations of Eq. 2.5 does not conserve the diagonality of the wave functions with respect to the Hamiltonian, i.e. $\langle\psi_\alpha \mid \hat{h} \mid \psi_\beta\rangle = \delta_{\alpha\,\beta}$, they have to be diagonalized after each step to obtain the eigenfunctions. Subsequently, single-particle properties, *e.g.* single-particle energies, are determined. If the convergence criterion (Eq.(2.2)) is fulfilled at the end of the current iteration, properties of the states and the wave functions are written into a file and the calculation is terminated.

2.3 Distributed Memory Parallelization With MPI

There are basically two approaches for distributed memory parallelization of the Sky3D code. The first approach would employ a spatial decomposition where the three dimensional space is partitioned and corresponding grid points are distributed over different MPI ranks. However, computations like the calculation of the density gradients $\nabla\rho_q(\vec{r})$ are global operations that require 3D FFTs, which are known to have poor scaling due to their dependence on all-to-all interprocess communications [97]. Hence, this approach would not scale well. The second approach would be to distribute the single particle wave functions among processes. While communications are unavoidable, by carefully partitioning the data at each step, it is possible to significantly reduce the communication overheads in this scheme. In what

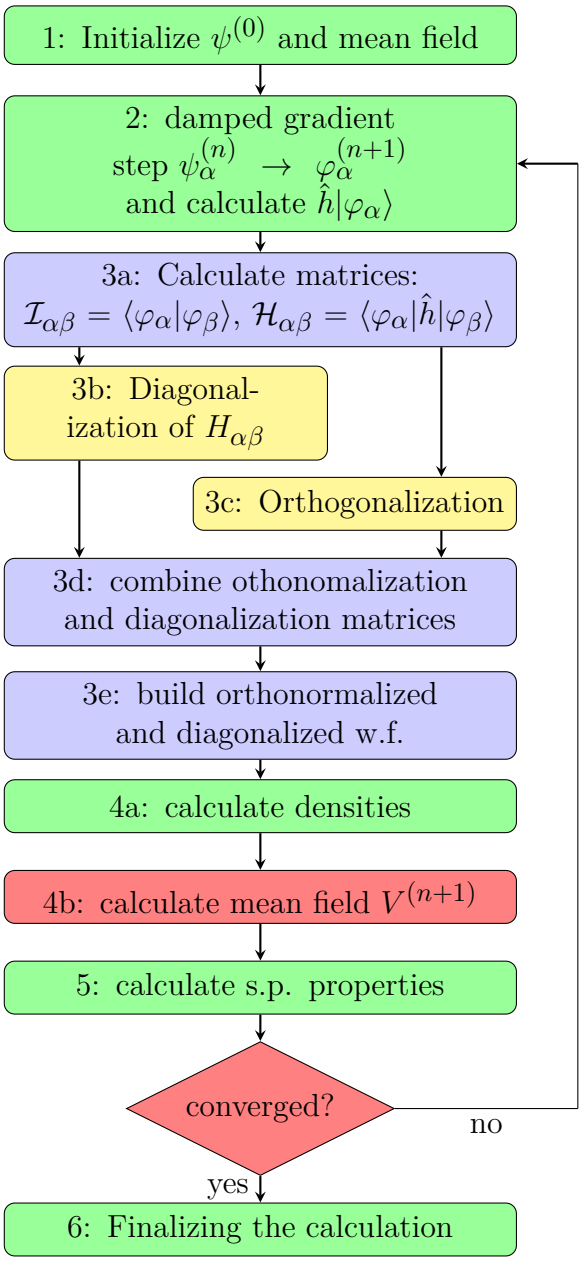


Figure 2.1: Flowchart of the parallelized Sky3D code. Parts in the 1d distribution are marked in green, parts in full 2d distribution are marked in blue, parts in divided 2d distribution are marked in yellow and collective non-parallelized parts are marked in red. $\psi_\alpha^{(n)}$ denotes the orthonormal and diagonal wave function at step n with index α, $|\varphi_\alpha\rangle = \varphi_\alpha^{(n+1)}$ denotes the non-diagonal and non-orthonormal wave function at step n+1. $\hat{h}$ is the one-body Hamiltonian.

follows, we present our parallel implementation of Sky3D using this second approach.

The iterative steps following the initialization phase constitute the computationally expensive part of Sky3D. Hence, our discussion will focus on the implementation of steps 2 through 5 of Fig. 2.1. Computations in these steps can be classified into two groups, i) those that work with matrices (and require 2D distributions to obtain good scaling), and ii) those that work on the wave functions themselves (and utilize 1D distributions as it is more convenient in this case to have wave functions to be fully present on a node). Our parallel implementation progresses by switching between these 2D partitioned steps (marked in violet and yellow in Fig. 2.1) and 1D partitioned steps (marked in green) in each iteration. Steps marked in red are not parallelized.

As discussed in more detail below, an important aspect of our implementation is that we make use of optimized scientific computing libraries such as ScaLAPACK [98] and FFTW [99] wherever possible. Since ScaLAPACK and FFTW are widely used and well optimized across HPC systems, this approach allows us to achieve high performance on a wide variety of architectures without the added burden of fine-tuning Sky3D. This may even extend to future architectures with decreased memory space per core and possibly multiple levels in the memory hierarchy. As implementations of ScaLAPACK and FFTW libraries evolve for such systems, we anticipate that it will be relatively easy to adapt our implementation to such changes in HPC systems.

2.3.1 The 1D and 2D partitionings

The decisions regarding 1D and 2D decompositions are made around the wave functions which represent the main data structure in Sky3D and are involved in all the key computational steps. We represent the wave functions using a two dimensional array `psi(V,A)`,

where $V = n_x \times n_y \times n_z \times 2$ includes the spatial degrees of freedom and the spin degree of freedom with n_x, n_y and n_z being the grid sizes in x, y and z directions, respectively, and the factor 2 originating from the two components of the spinor. In the case of 1D distribution, full wave functions are distributed among processes in a block cyclic way. The block size N_ψ determines how many consecutive wave functions are given to each process in each round. In round one, the first N_ψ wave functions are given to the first process P_0, then the second process P_1 gets the second batch and so on. When all processes are assigned a batch of wave functions in a round, the distribution resumes with P_0 in the subsequent round until all wave functions are exhausted.

In the 2D partitioning case, single particle wave functions as well as the matrices constructed using them (see Sect. 2.3.3.1) are divided among processes using a 2D block cyclic distribution. In Fig. 2.2, we provide a visual example of a square matrix distributed in a 2D block cyclic fashion where processes are organized into a 3×2 grid topology, and the row block size *NB* and the column block size *MB* have been set equal to 2. The small rectangular boxes in the matrix show the arrangement of processes in the 3×2 grid topology – the number of rows are in general not equal to the number of columns in the process grid. For symmetric or Hermitian matrices only the (blue marked) lower triangular part is needed as it defines the matrix fully. In this particular case, P_0 is assigned the matrix elements in rows 1, 2, 7, 8, 13 and 14 in the first column, as well as those in rows 2, 7, 8, 13 and 14 in the second column; P_1 is assigned the matrix elements in rows 7, 8, 13 and 14 in columns 3 and 4, and so on. Single particle wave functions, which are stored as rectangular (non-symmetric) matrices with significantly more number of rows than the number of columnd (due to the large grid sizes needed), are also distributed using the same strategy. The 2D block cyclic distribution provides very good load balance in computations associated with

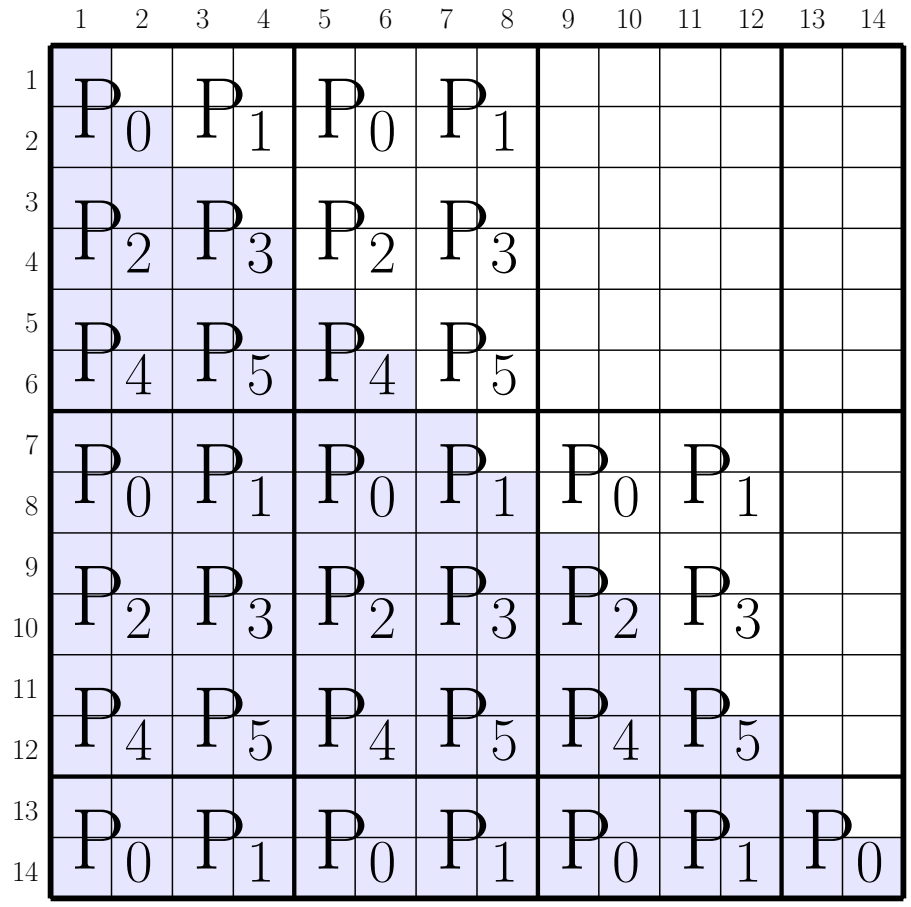


Figure 2.2: 2D block cyclic partitioning example with 14 wave functions using a 3×2 processor topology. Row and column block sizes are set as NB=MB=2. The blue shaded area marks the lower triangular part.

the single particle wave functions and the matrices formed using them, as all processes are assigned approximately the same number of elements.

2.3.2 Parallelization across Neutron and Proton groups

An important observation for parallelization of Sky3D is that neutrons and protons interact only through the mean field. Therefore, the only communication needed between these two species takes place within step 4b. To achieve better scalability, we leverage this fact and

separate computations associated with neutrons and protons to different processor groups, while trying to preserve the load balance between them as explained in detail below.

Suppose $A = N + Z$ is the total number of wave functions, such that N is the number of neutron wave functions and Z is the number of proton wave functions. To distribute the array `psi`, we split the available processors into two groups; one group of size P_N for neutrons, and another group of size P_P for protons. We note that in practical simulations, the number of neutrons are often larger than the number of protons (for example, simulations of neutron star matter are naturally neutron rich). The partitioning of processors into neutron and proton groups must account for this situation to ensure good load balance between the two groups. As will be discussed in Section 2.3.3, Sky3D execution time is mainly determined by the time spent on 2D partitioned steps which is primarily dominated by the construction of the overlap and Hamiltonian matrices step, and to a lesser degree by eigenvalue computations associated with these matrices. Since the computational cost of the matrix construction step is proportional to the square of the number of particles in a group (see Section 2.3.3.1), we choose to split processors into two groups by quadratically weighing the number of particles in each species. More precisely, if the total processor count is given by C, then $P_N = \frac{N^2}{N^2+Z^2}C$ and $P_P = \frac{Z^2}{N^2+Z^2}C$ according to this scheme. It is well-known that 2D partitioning is optimal for scalable matrix-matrix multiplications [100]. Therefore, once the number of processors within neutron and proton groups is set, we determine the number of rows and columns for the 2D process topologies of each of these groups through MPI's `MPI_DIMS_CREATE` function. This function ensures that the number of rows and columns is as close as possible to the square root of P_N and P_P for neutron and proton groups, respectively, thus yielding a good 2D process layout.

As will be demonstrated through numerical experiments, the scheme described above

typically delivers computations with good load balances, but it has a caveat. Under certain circumstances, this division might yield a 2D process grid with a tall & skinny layout, which essentially is more similar to a 1D partitioning and have led to significant performance degradations for our 2D partitioned computations. For instance, in a system with 5000 neutrons and 1000 protons, when we use 256 processors, according to our scheme P_N will be 246, and P_P will be 10. For $P_N = 246$, the corresponding process grid is $(41 * 6)$ which is much closer to a 1D layout than a 2D layout. To prevent such circumstances, we require the number of processors within each group to be a multiple of certain powers of 2 (*i.e.*, 2, 4, 8, 16 or 32 depending on the total core count). For the above example, by requiring the number of cores within each group to be a multiple of 16, we determine P_N to be 240 and P_P to be 16. This results in a process grid of size 16×15 for neutrons which is almost a square shaped grid, and a perfect square grid of size 4×4 for protons.

2.3.3 Calculations with 2D distributions

As a result of the split, neutron and proton processor groups asynchronously advance through steps that require 2D partitionings, *i.e.*, steps 3a to 3e. Main computational tasks here are the construction of the overlap and Hamiltonian matrices (using 1D distributed wave functions) and eigendecompositions of these matrices. These tasks are essentially accomplished by calling suitable ScaLAPACK routines. The choice of a 2D block cyclic distribution maximizes the load balancing with ScaLAPACK for the steps 3a through 3e.

2.3.3.1 Matrix construction (step 3a)

Construction of the overlap matrix $\mathcal{I}$ and the Hamiltonian matrix $\mathcal{H}$

$$\mathcal{I}_{\alpha\beta} = \langle \varphi_\alpha \mid \varphi_\beta \rangle, \text{ and} \tag{2.6}$$

$$\mathcal{H}_{\alpha\beta} = \langle \varphi_\alpha \mid \hat{h} \mid \varphi_\beta \rangle, \tag{2.7}$$

where $\mid \varphi_\alpha \rangle$ marks the non-orthonormalized and non-diagonal wave functions, constitutes the most expensive part of the iterations in Sky3D, because the cost of these operations scales quadratically with the number of particles. More precisely, calculating these matrices costs $\mathcal{O}(I^2V)$, where $I \in \{N, Z\}$ is the number of wave functions. Since these two operations are simply inner products, we use ScaLAPACK's complex matrix matrix multiplication routine `PZGEMM` for constructing these two matrices.

One sublety here is that prior to the start of steps with 2D partitionings, wave functions are distributed in a 1D scheme. To achieve good performance and scaling with `PZGEMM`, we first switch both wave functions $\mid \varphi_\alpha \rangle$ (`psi`) and $\hat{h} \mid \varphi_\alpha \rangle$ (`hampsi`) into a 2D cyclic layout which uses the same process grid created through the `MPI_DIMS_CREATE` function. The `PZGEMM` call then operates on these two matrices and the complex conjugate of (`psi`). The resulting matrices $\mathcal{I}$ and $\mathcal{H}$ are distributed over the same 2D process grid as well.

Since $\mathcal{I}$ and $\mathcal{H}$ are square matrices, we set the row and column block sizes, NB and MB, respectively, to be equal (*i.e.,* $NB = MB$). Normally, in a 2D matrix computation, one would expect a trade-off in choosing the exact value for NB and MB, as small blocks lead to a favorable load balance, but large blocks reduce communication overheads. However, for typical problems that we are interested in, *i.e.*, more than 1000 particles using at least a few hundred cores, our experiments with different NB and MB values such as 2, 4, 8, 32, 64

have shown negligible performance differences. Therefore, we have empirically determined the choice for $NB = MB$ to be 32 for large computations.

2.3.3.2 Diagonalization and Orthonormalization (steps 3b & c)

After the matrix $\mathcal{H}$ is constructed according to Eq. (2.7), its eigenvalue decomposition is computed to find the eigenstates of the Hamiltonian. Since $\mathcal{H}$ is a hermitian matrix, we use the complex Hermitian eigenvalue decomposition routine `PZHEEVR` in ScaLAPACK, which first reduces the input matrix to tridiagonal form, and then computes the eigenspectrum using the Multiple Relatively Robust Representations (MRRR) algorithm [101]. The local matrices produced by the 2D block cyclic distribution of the matrix construction step can readily be used as input to the `PZHEEVR` routine. After the eigenvectors of $\mathcal{H}$ are obtained, the diagonal set of wave functions ψ_α can be obtained through the following matrix-vector multiplication

$$\psi_\alpha = \sum_\beta \mathcal{Z}^H_{\alpha\beta}\varphi_\beta \tag{2.8}$$

for all φ_β where $\mathcal{Z}$ is the matrix containing the eigenvectors of $\mathcal{H}$.

Orthonormalization is commonly accomplished through the modified Gram-Schmidt (mGS) method, a numerically more stable version of the *classical* Gram-Schmidt method. Unlike the original version of Sky3D, we did not opt for mGS for a number of reasons. First, mGS is an inherently sequential process where the orthonormalization of wave function $n+1$ can start only after the first n vectors are orthonormalized. Second, the main computational kernel in this method is a dot product which is a Level-1 BLAS operation and has low arithmetic intensity. Finally, a parallel mGS with a block cyclic distribution of wave functions $|\,\varphi_\alpha\rangle$ as used by matrix construction and diagonalization steps would incur significant syn-

chronization overheads, especially due to the small blocking factors needed to load balance the matrix construction step.

An alternative approach to orthonormalize the wave functions is the Löwdin method [102], which can be stated for our purposes as:

$$\mathcal{C} = \mathcal{I}^{-1/2} \tag{2.9a}$$

$$\psi_\alpha = \sum_\beta \mathcal{C}_{\beta\alpha}\varphi_\beta. \tag{2.9b}$$

The Löwdin orthonormalization is well known in quantum chemistry and has the property that the orthornormalized wave functions are those that are closest to the non-orthonormalized wave functions in a least-squares sense. Note that since $\mathcal{I}$ is a Hermitian matrix, it can be factorized as $\mathcal{I} = X\Lambda X^T$, where columns of X are its eigenvectors and Λ is a diagonal matrix composed of $\mathcal{I}$'s eigenvalues. Consequently, $\mathcal{I}^{-1/2}$ in Eq.2.9a can be computed simply by taking the inverses of the square roots of $\mathcal{I}$'s eigenvalues, *i.e.*, $\mathcal{C} = \mathcal{I}^{-1/2} = X\Lambda^{-1/2}X^T$.

Applying the Löwdin method in our problem is equivalent to computing an eigendecomposition of the overlap matrix $\mathcal{I}$, which can also be implemented by using the `PZHEEVR` routine in ScaLAPACK. Note that exactly the same distribution of wave functions and blocking factors as in the matrix construction step can be used for this step, too.

Detailed performance analyses reveal that the eigendecomposition routine `PZHEEVR` does not scale well for large P with the usual number of wave functions in nuclear pasta calculations. However, the eigendecompositions of the $\mathcal{I}$ and $\mathcal{H}$ matrices (within both the neutron and proton groups) are independent of each other and their construction is also similar with respect to each other. Therefore, to gain additional performance, we perform steps 3b and 3c in parallel using half the number of MPI ranks available in a group, *i.e.*, $P_N/2$ and $P_P/2$,

respectively for neutrons and protons.

2.3.3.3 Post-processing (steps 3d & e)

The post-processing operations in the diagonalization and orthonormalization steps are matrix-vector multiplications acting on the current set of wave functions φ_j. As opposed to applying these operations one after the other, *i.e.*, $\mathcal{C}^T(\mathcal{Z}^H\{\varphi\})$, we combine diagonalization and orthonormalization by performing $(\mathcal{C}^T\mathcal{Z}^H)\{\varphi\}$, where $\{\varphi\} = (\varphi_1, \varphi_2, .., \varphi_n)^T$ denotes a vector containing the single-particle wave functions. While both sequences of operations are arithmetically equivalent, the latter has a benefit in terms of the computational cost, as it reduces the number of multiply-adds from $2I^2V$ to $I^3 + I^2V$. This is almost half the cost of using the first sequence of operations, since we have $I << V$ for both neutrons and protons, because the number of wave functions has to be significantly smaller than the size of the basis to prevent any bias due to the choice of the basis. We describe this optimization in the form of a pseudocode in Fig. **??**. By computing the overlap matrix $\mathcal{I}$ together with the Hamiltonian matrix $\mathcal{H}$, and performing their eigendecompositions, we can combine the update and orthonormalization of wave functions. Lines shown in red in Fig. **??** mark those affected by this optimization. Consequently, the matrix-matrix multiplication $C^T\mathcal{Z}^H$ can be performed prior to the matrix-vector multiplication involving the wave functions $\{\varphi(\vec{r}_\nu)\}$. As a result, the overall computational cost is significantly reduced, and efficient level-3 BLAS routines can be leveraged.

The $C^T\mathcal{Z}^H$ operation is carried out in parallel using ScaLAPACK's `PZGEMM` routine (step 3d). Then the resulting matrix is multiplied with the vector of wave functions for all spatial and spin degrees of freedom in step 3e using another `PZGEMM` call.

It should be noted that with this method, we introduce small errors during iterations,

because the matrix $\mathcal{C}^T$ is calculated from non-orthogonalized wave functions. However, since we take small gradient steps, these errors disappear as we reach convergence and we ultimately arrive at the same results as an implementation which computes the matrix $\mathcal{C}^T$ from orthogonalized wave functions.

2.3.4 Calculations with a 1D distribution

Steps 2, 4 and 5 use the 1D distribution, because for a given wave function ψ_α, computations in these steps are independent of all other wave functions. Within the neutron and proton processor groups, we distribute wave functions using a 1D block cyclic distribution with block size NB_ψ. Such a distribution further ensures good load balance and facilitates the communication between 1D and 2D distributions. The damped gradient step (as shown in the curled brackets in Eq. 2.5) is performed in step 2. Here, the operator $\hat{T}$ shown in Eq. 2.5 is calculated using the FFTW library. Since the Hamiltonian has to be applied to the wave functions in this step, $\hat{h} \mid \psi_\alpha\rangle$ is saved in the array `hampsi`, distributed in the same way as `psi` and will be reused in step 3a. In step 4a, the partial densities as given in Eqs. (2.4a)-(2.4c) are calculated on each node for the local wave functions separately and subsequently summed up with the MPI routine `MPI_ALLREDUCE`. We use FFTs to compute derivatives of wave functions as needed in Eqs. (2.4a)-(2.4c). The determination of the mean field in step 4b does not depend on the number of particles, and is generally not expensive. Consequently, this computation is performed redundantly by each MPI rank to avoid synchronizations. Also the check for convergence is performed on each MPI rank separately. Both are marked in red in Fig. 2.1. Finally, in step 5, single-particle properties are calculated and partial results for single-particle properties are aggregated on all MPI ranks using an `MPI_ALLREDUCE`.

2.3.5 Switching between different data distributions

As described above, our parallel Sky3D implementation uses 3 different distributions: The 1D distribution which is defined separately for neutrons and protons and used in green marked steps of Fig 2.1, the 2d distribution which is again defined separately for neutrons and protons and used in blue marked steps, and the 2d distribution for diagonalization and orthogonalization (used in steps marked in yellow) within subgroups of size $P_N/2$ and $P_P/2$, respectively for neutrons and protons. For calculation of overlap and Hamiltonian matrices, wave functions `psi` and `hampsi` need to be switched from the 1D distribution into the 2D distribution after step 2. After step 3a, matrices $\mathcal{I}$ and $\mathcal{H}$ must be transfered to the subgroups. After eigendecompositions in steps 3b and 3c are completed, the matrices $\mathcal{Z}$ and $\mathcal{C}$, which contain the eigenvectors, need to be redistributed back to the full 2D groups. After step 3e, only the updated array `psi` has to be switched back to the 1D distribution from the 2D distributions.

While these operations require complicated interprocess communications, they are easily carried out with the ScaLAPACK routine `PZGEMR2D` which can transfer distributed matrices from one processor grid to another, even if the involved grids are totally different in their shapes and formations. As we will demonstrate in the performance evaluation section, the time required by `PZGEMR2D` is insignificant, including in large scale calculations.

2.3.6 Memory considerations

Beyond performance optimizations, memory utilization is of great importance for large-scale nuclear pasta calculations. Data structures that require major memory space in Sky3D are the wave functions stored in matrices `psi` and `hampsi`. The latter matrix was not

needed in the original Sky3D code as $\mathcal{H}$ was calculated on the fly, but this is not an option for a distributed memory implementation. As such, the total memory need increases by roughly a factor of 2. Furthermore, we store both arrays in both 1D and 2D distributions, which contributes another factor of 2. Besides the wavefunctions, another major memory requirement is storage of the matrices such as $\mathcal{H}$ and $\mathcal{I}$. These data structures are much smaller because the total matrix size grows only as N^2 for neutrons and Z^2 for protons. In our implementation, we store these matrices twice for the 2D distribution and twice for the 2D distributions within subgroups.

To give an example as to the actual memory utilization, largest calculations we conducted in this work are nuclear pasta calculations with a cubic lattice of 48 points and 6000 wave functions. In this case, the aggregate size of a single matrix to store wave functions in double precision complex format is $48^3 \times 2 \times 6000 \times 16 \approx 21\,\text{GB}$, and all four arrays required in our parallelization would amount to about 84 GBs of memory. The Hamiltonian and overlap matrices occupy a much smaller footprint, roughly 144 MBs per matrix. As this example shows, it is still feasible to carry out large scale nuclear pasta formations using the developed parallel Sky3D code. Even if we choose a bigger grid, *e.g.*, of size 64^3, with typical compute nodes in today's HPC systems having ≥ 64 GB of memory and the memory need per MPI rank decreasing linearly with the total number of MPI ranks in a calculation (there is little to no duplication of data structures across MPI ranks), such calculations would still be feasible using our implementation.

2.4 Shared Memory Parallelization with OpenMP

In addition to MPI parallelization, we also implemented a hybrid MPI/OpenMP parallel version of Sky3D. The rational behind a hybrid parallel implementation is that it would map more naturally to today's multi-core architectures, and it may also reduce the amount of inter-node communication using MPI.

For the 1D distribution calculations, similar to the MPI implementation, we distribute the wave functions over threads by parallelizing loops using OpenMP. Since no communication is needed for step 2, we do not expect a gain in performance as a result of the shared memory implementation in this step. Step 4a, however, involves major communications, as the partial sums of the densities have to be reduced across all threads. The OpenMP implementation reduces the number of MPI ranks when the total core count P is kept constant. This reduces the amount of inter-node communication. Similarly, step 5 involves the communication of single particle properties. Since these quantities are mainly scalars or small sized vectors though, inter-node communications are not as expensive.

The steps with a 2D distribution, *i.e.*, steps 3a-3e, largely rely on ScaLAPACK routines. In this part, shared memory parallelization is introduced implicitly via the usage of multi-threaded ScaLAPACK routines. Consequently, note that we rely mostly on the ScaLAPACK implementation and its thread optimizations for the steps with 2D data distributions.

2.5 Performance Evaluation

2.5.1 Experimental setup

We conducted our computational experiments on Cori - Phase I, a Cray XC40 supercomputing platform at NERSC, which contains two 16-core Xeon E5-2698 v3 Haswell CPUs per node (see Table 7.1). Each of the 16 cores runs at 2.3 GHz and is capable of executing one fused multiply-add (FMA) AVX (4×64-bit SIMD) operation per cycle. Each core has a 64 KB L1 cache (32 KB instruction and 32 KB data cache) and a 256 KB L2 cache, both of which are private to each core. In addition, each CPU has a 40 MB shared L3 cache. The Xeon E5-2698 v3 CPU supports hyperthreading which would essentially allow the use of 64 processes or threads per Cori-Phase I node. Our experiments with hyperthreading have led to performance degradation for both the MPI and MPI/OpenMP hybrid parallel implementations. As such, we have disabled hyperthreading in our performance tests.

For performance analysis, we choose a physically relevant setup. In practice, the grid spacing is about $\Delta x = \Delta y = \Delta z \sim 1\,\mathrm{fm}$. This grid spacing gives results with desired accuracy [53]. For nuclear pasta matter, very high mean number densities ($0.02\,\mathrm{fm}^{-3} - 0.14\,\mathrm{fm}^{-3}$) have to be reached. We choose two different cubic boxes of $L = 32\,\mathrm{fm}$ and $L = 48\,\mathrm{fm}$ and a fixed number of nucleons $N + Z = 6000$. This results in mean densities of $0.122\,\mathrm{fm}^{-3}$ and $0.036\,\mathrm{fm}^{-3}$, respectively.

To eliminate any load balancing effects, we begin with our performance and scalability tests using a symmetric system with 3000 neutrons and 3000 protons. As the systems for neutron star applications are neutron rich, we also test systems with 4000 neutron and 2000 protons and also 5000 neutrons and 1000 protons.

Platform	**Cray XC40**
Processor	**Xeon E5-2698 v3**
Core	**Haswell**
Clock (GHz)	2.3
Data Cache (KB)	64(32+32)+256
Memory-Parallelism	HW-prefetch
Cores/Processor	16
Last-level L3 Cache	40 MB
SP TFlop/s	1.2
DP TFlop/s	0.6
STREAM BW[3]	120 GB/s
Available Memory/node	128 GB
Interconnect	Cray Aries (Dragonfly)
Global BW	5.625 TB/s
MPI Library	MPICHv2
Compiler	Intel/17.0.2.174

Table 2.1: Hardware specifications for a single socket on Cori, a Cray XC40 supercomputer at NERSC. Each node consists of two sockets.

2.5.2 Scalability

First, we consider a system with 3000 neutrons and 3000 protons with two different grid sizes, $L = 32\,\text{fm}$ and $L = 48\,\text{fm}$. On Cori, each "Haswell" node contains 32 cores on two sockets. For both grid sizes, we ran simulations on 1, 2, 4, 8, 16, 32 and 48 nodes using 32, 64, 128, 256, 1024 and 1536 MPI ranks, respectively. In all our tests, all nodes are fully packed, *i.e.*, one MPI rank is assigned to each core, exerting full load on the memory/cache system.

Performance results for $L = 32\,\text{fm}$ and $L = 48\,\text{fm}$ cases are shown in Fig. 2.3 and Fig. 2.4, respectively. In both figures, total execution time per iteration is broken down into the time spent for individual steps of Fig. 2.1. In addition, "communication" represents the time

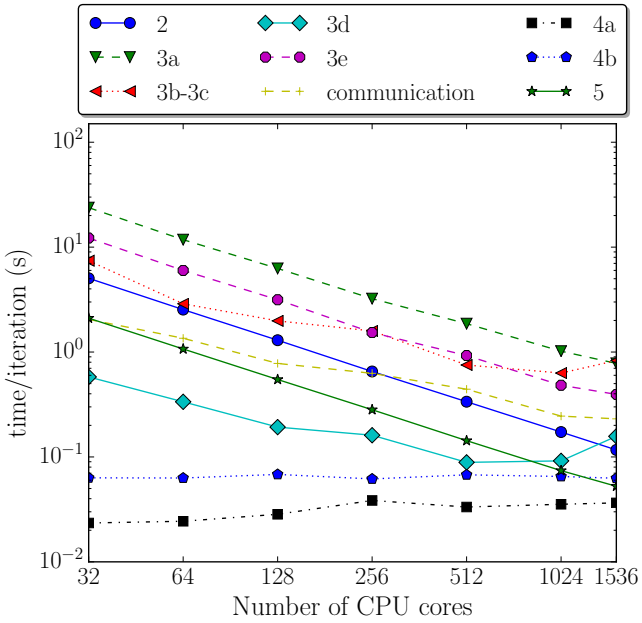


Figure 2.3: Scalability of MPI-only version of Sky3D for the 3000 neutron and 3000 proton system using the $L = 32\,\mathrm{fm}$ grid.

needed by ScaLAPACK's `PZGEMR2D` routine for data movement during switches between different distributions (*i.e.*, 1D, 2D, and 2D subgroups). For each step, we use a single line to report the time for neutron and proton computations by taking their maximum.

As seen in Fig. 2.3, calculation of the matrices (step 3a) is the most expensive step. Another expensive step is step 3e where diagonalized and orthonormalized wave functions are built. Diagonalization of the Hamiltonian $\mathcal{H}$ and the Löwdin orthonormalization procedures (steps 3b-3c, which are combined into a single step as they are performed in parallel) also takes significant amount of time. It can be seen that step 4a does not scale well, because it consists mainly of communication of the densities. We also note that step 4b is not parallelized, it is rather performed redundantly on each process because it takes an insignificant amount of time.

The damped gradient step (step 2) and computation of single particle properties (step 5)

Table 2.2: Scalability of MPI-only version of Sky3D for the $L = 32\,\mathrm{fm}$ grid. Time is given in seconds, and efficiency (eff) is given in percentages.

	calc. matrix		**recombine**		**diag+Löwdin**		**Total**	
cores	**time**	**eff**	**time**	**eff**	**time**	**eff**	**time**	**eff**
32	23.8	100	12.2	100	7.4	100	59.8	100
64	11.8	101.5	6.0	102.0	2.9	128.7	28.4	105.1
128	6.3	94.8	3.2	96.8	2.0	93.6	16.0	93.5
256	3.2	92.2	1.5	99.4	1.6	58.4	9.2	81.4
512	1.9	79.8	0.9	82.4	0.8	61.6	5.3	70.5
1024	1.0	73.0	0.5	79.1	0.6	36.8	3.4	55.0
1536	0.8	63.6	0.4	64.4	0.8	18.6	3.5	36.1

scale almost perfectly. Steps 3a and 3e, which are compute intensive kernels, also exhibit good scaling. While ScaLAPACK's eigensolver routine `PZHEEVR` performs well for smaller number of cores, it does not scale to a high number of cores. In fact, the internal communications in this routine becomes a significant bottleneck to the extent that steps 3b and 3c become the most expensive part of the calculation on 1536 cores. In Table 2.2, we give strong scaling efficiencies for the most important parts of the iterations for the $L = 32\,\mathrm{fm}$ grid. Overall, we observe good scaling up to 512 cores, where we achieve 70.5% efficiency. However, this number drops to 36.1% on 1536 cores and steps 3b-3c are the main reason for this drop.

In Fig. 2.4, strong scaling results for the $L = 48\,\mathrm{fm}$ grid is shown. In this case, the number of neutrons and protons are the same, but the size of wave functions is larger than the previous case. As a result, the computation intensive steps 3a, 3e and 4b which directly work on these wave functions are significantly more expensive than the corresponding runs for the $L = 32\,\mathrm{fm}$ case. As it is clearly visible in this figure, on small number of nodes, the overall iteration time is dominated by these compute-intensive kernels. This changes in larger scale runs, where the times spent in diagonalization and Löwdin orthonormalization

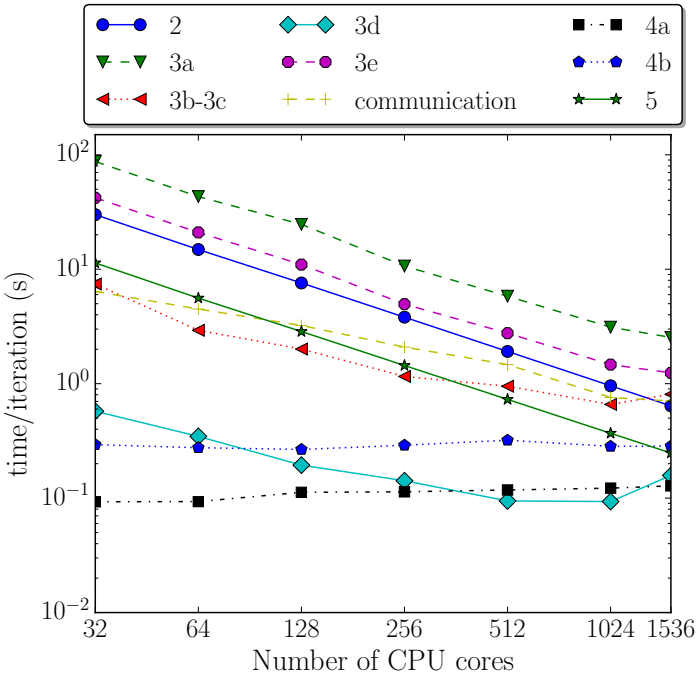


Figure 2.4: Scalability of MPI-only version of Sky3D for the 3000 neutron and 3000 proton system using the $L = 48$ fm grid.

(steps 3b-3c) along with communication operations also become significant.

The increased size of wave functions and increased computational intensity actually results in better scalability. As shown in Table 2.3, we observe 93.1% efficiency during matrix construction and 95.5% efficiency for the recombine step on 1024 cores. We note that the calculate matrix step's efficiency can actually be greater than 100% owing to the perfect square core counts like 64 and 256 cores. However, like in the previous case, the diagonalization and orthonormalization steps do not scale well for larger number of cores.

Overall, Sky3D shows good scalability for small to moderate number of nodes, but this decreases slightly with increased core counts. This decrease in efficiency is mainly due to the poor scaling of ScaLAPACK's eigensolver used in the diagonalization and orthonormalization steps, and partially due to the cost of having to communicate large wave functions at each Sky3D iteration.

Table 2.3: Scalability of MPI-only version of Sky3D for the $L = 48\,\mathrm{fm}$ grid. Time is given in seconds, and efficiency (eff) is given in percentages.

	calc. matrix		**recombine**		**diag+Löwdin**		**Total**	
cores	**time**	**eff**	**time**	**eff**	**time**	**eff**	**time**	**eff**
32	87.3	100	42.1	100	7.4	100	192	100
64	43.1	101.2	21	100.3	2.9	127	95.1	100.9
128	24.7	88.5	11	95.9	2.0	92.6	53.6	89.6
256	10.6	102	4.9	107.2	1.2	80.3	25.51	94.1
512	5.8	94	2.8	95.2	0.9	48.9	15	80.2
1024	2.9	93.1	1.4	95.5	0.6	35.3	8.4	71.3
1536	2.5	71.7	1.2	70.5	0.8	19.1	7.54	53

2.5.3 Comparison between MPI-only and MPI/OpenMP hybrid parallelization

On Cori, "Haswell" compute nodes contain two sockets with 16 cores each. To prevent any performance degradations due to non-uniform memory accesses (NUMA), we performed our tests using 2 MPI ranks per node with each MPI rank having 16 OpenMP threads executed on a single socket. Since we are grouping the available cores into neutron and proton groups which are further divided in half for running diagonalization and orthonormalization tasks in parallel, we need a minimum of 4 MPI ranks in each test. In Figures 2.5 and 2.7, we show strong scalability test results similar to the MPI-only implementation discussed earlier. In this case, the legends along the x-axis denotes the total core counts. For example, 128 means that we are running this test on 4 nodes with 8 MPI ranks and 16 OpenMP threads per rank. For the $L = 32\,\mathrm{fm}$ grid, we have tested the MPI/OpenMP hybrid parallel version with 4, 8, 16, 32, 64, 96 MPI ranks. For the $L = 48\,\mathrm{fm}$ grid, we have a larger number of wave functions for which ScaLAPACK's data redistribution routine `PZGEMR2D` runs out of memory on low node counts. Hence, we tested this case with 16, 32, 64 and 96 MPI ranks only.

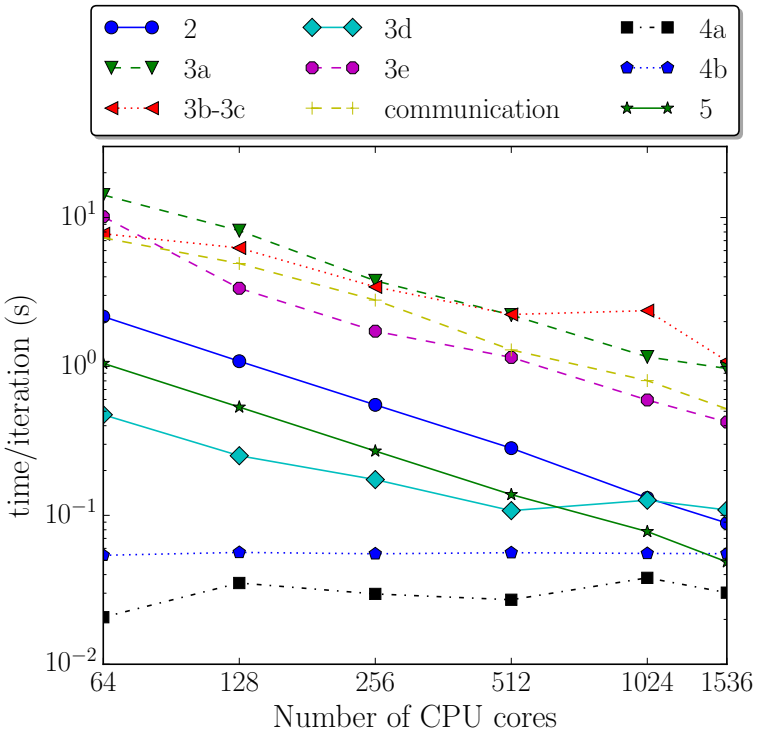


Figure 2.5: Scalability of MPI/OpenMP parallel version of Sky3D for the 3000 neutron and 3000 proton system using the $L = 32\,\mathrm{fm}$ grid.

Table 2.4: Scalability of MPI/OpenMP parallel version of Sky3D for the $L = 32\,\mathrm{fm}$ grid. Time is given in seconds, and efficiency (eff) is given in percentages.

	calc. matrix		**recombine**		**diag+lowedin**		**Total**	
cores	**time**	**eff**	**time**	**eff**	**time**	**eff**	**time**	**eff**
64	14.2	100	10.1	100	7.8	100	50.7	100
128	8.2	86.8	3.34	151.2	6.2	62.3	30.7	82.6
256	3.8	94.2	1.7	146.8	3.4	56.8	16	79.4
512	2.2	80	1.1	110.1	2.2	43.5	9.6	66
1024	1.2	76.4	0.6	106.3	2.4	20.5	6.8	46.7
1536	1	61	0.4	99.7	1.1	29.8	4.4	48.2

In Fig. 2.5, we show the strong scaling results for the $L = 32\,\mathrm{fm}$ case, along with detailed efficiency numbers for the computationally expensive steps in Table 2.4. Similar to the MPI-only case, the compute-intensive matrix construction and recombine phases show good scalability, but the diagonalization and Löwdin orthonormalization part does not perform as well as these two parts. While the strong scaling efficiency numbers in this case look better than the MPI-only case (see Table 2.2), we note that this is due to the inferior performance of the MPI/OpenMP parallel version for its base case of 64 cores. In fact, the recombine part performs so poor on 64 cores that its strong scaling efficiency is constantly over 100% almost all the way up to 1536 cores. But comparing the total execution times, we see that the MPI-only code takes 28.4 seconds on average per iteration, while the MPI/OpenMP parallel version takes 50.7 seconds for this same problem on 64 cores.

This performance issue in the MPI/OpenMP version persists through all test cases as shown in Figure 2.6. In particular, for smaller number of cores, the performance of MPI/OpenMP version is poor compared to the MPI-only version. With increasing core counts, the performance difference lessens slightly, but the MPI-only version still performs better. In general, we observe that the MPI-only version outperforms the MPI/OpenMP

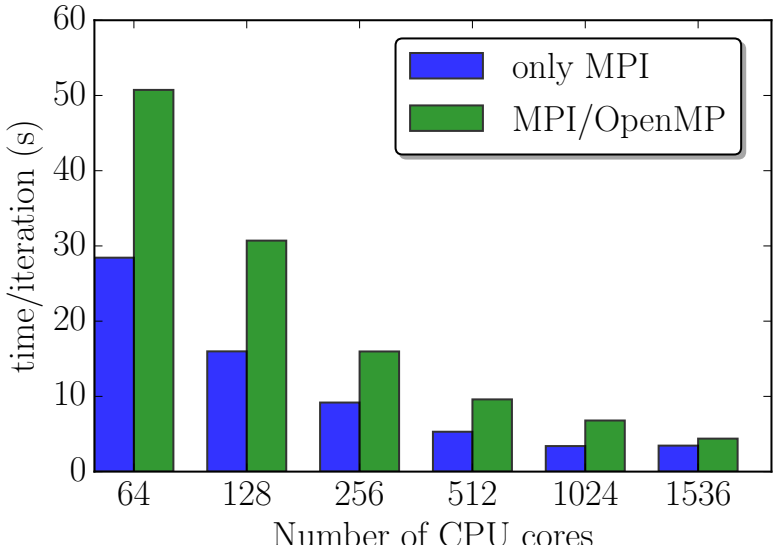


Figure 2.6: Comparison of the execution times for the MPI-only and MPI/OpenMP parallel versions of Sky3D for the 3000 neutron and 3000 proton system using the $L = 32\,\mathrm{fm}$ grid.

version by a factor of about 1.5x to 2x. The main reason behind this is that the thread parallel ScaLAPACK routines used in the MPI/OpenMP implementation perform worse than the MPI-only ScaLAPACK routines, which is contrary to what one might naively expect, given that our tests are performed on a multi-core architecture.

In Fig. 2.7 and Table 2.5, we show the strong scaling results for the $L = 48\,\mathrm{fm}$ grid. Again, in this case parts directly working with the wave functions, *i.e.*, calculation of matrices (step 3a) and building of orthonormalized and diagonalized wave functions (step 3e), become significantly more expensive compared to the diagonalization and Löwdin orthonormalizations (steps 3b & 3c). Of particular note here is the more pronounced communication times during switches between different data distributions which is mainly due to the larger size of the wave functions. Overall, we obtain 64.5% strong scaling efficiency using up to 96 MPI ranks with 16 threads per rank (1536 cores in total). In terms of total execution times though, MPI/OpenMP parallel version still underperforms compared to the MPI-only version (see Fig. 2.8).

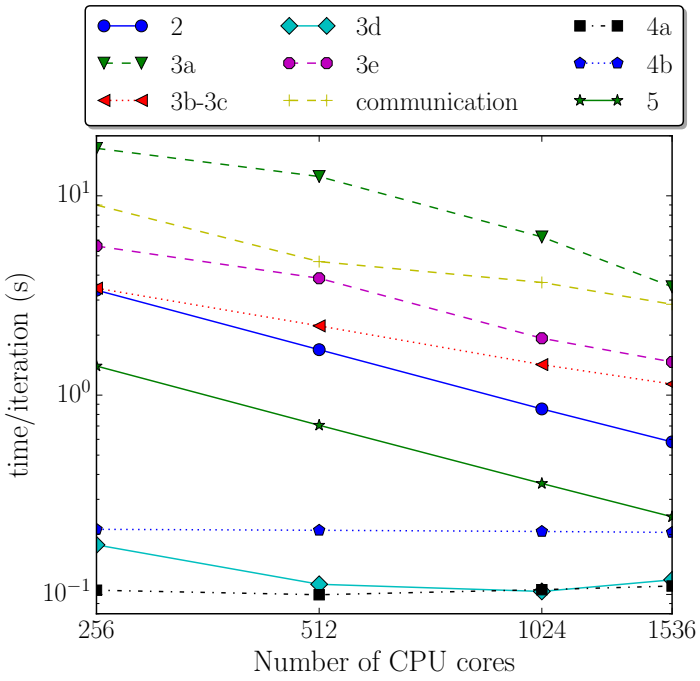


Figure 2.7: Scalability of MPI/OpenMP parallel version of Sky3D for the 3000 neutron and 3000 proton system using the $L = 48\,\mathrm{fm}$ grid.

Table 2.5: Scalability of MPI/OpenMP parallel version of Sky3D for the $L = 48\,\mathrm{fm}$ grid. Time is given in seconds, and efficiency (eff) is given in percentages.

	calc. matrix		**recombine**		**diag+Löwdin**		**Total**	
cores	**time**	**eff**	**time**	**eff**	**time**	**eff**	**time**	**eff**
256	17.3	100	5.6	100	3.4	100	43.7	100
512	12.5	69.3	3.9	72.5	2.2	77.3	28.1	77.9
1024	6.2	69.6	1.9	72.5	1.4	60.5	16.2	67.4
1536	3.5	82.5	1.5	63.6	1.1	50.5	11.3	64.5

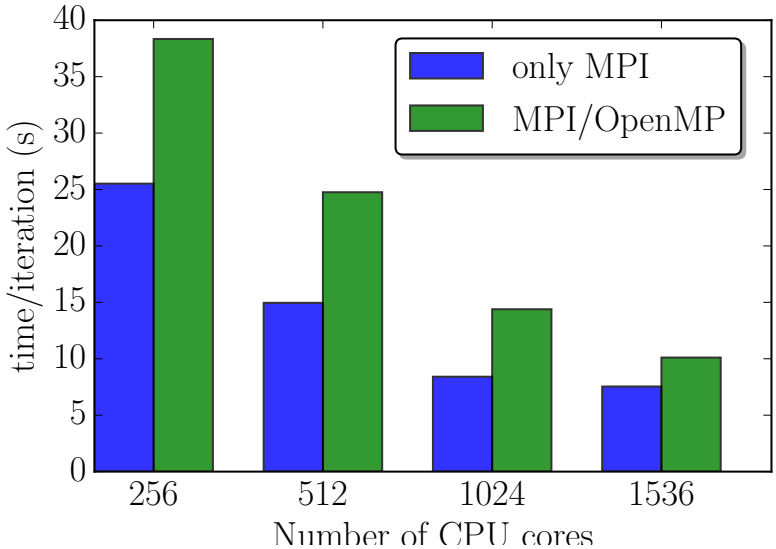


Figure 2.8: Comparison of the execution times for the MPI-only and MPI/OpenMP parallel versions of Sky3D for the 3000 neutron and 3000 proton system using the $L = 32\,\mathrm{fm}$ grid.

2.5.4 Load balancing

In this section, we analyze the performance of our load balancing approach which divides the available cores into neutron and proton groups for parallel execution. For better presentation, we break down the execution time into three major components: Calculations in steps using a 2D data distribution, calculations using a 1D distribution of wave functions and communication times. In Fig. 2.9(a), we show the time taken by the cores in the neutron and proton groups for the 3000 neutron and 3000 proton system using the $L = 48\,\mathrm{fm}$ grid - which is essentially the same plot as in the previous section, but it gives the timings for neutrons and protons separately. As this system has an equal number of neutrons and protons, available cores are divided equally into two groups. As can be seen in Fig. 2.9(a), the time needed for different steps in this case is almost exactly identical for neutrons and protons.

In Figure 2.9(b), we present the results for a system with 4000 neutrons and 2000 protons.

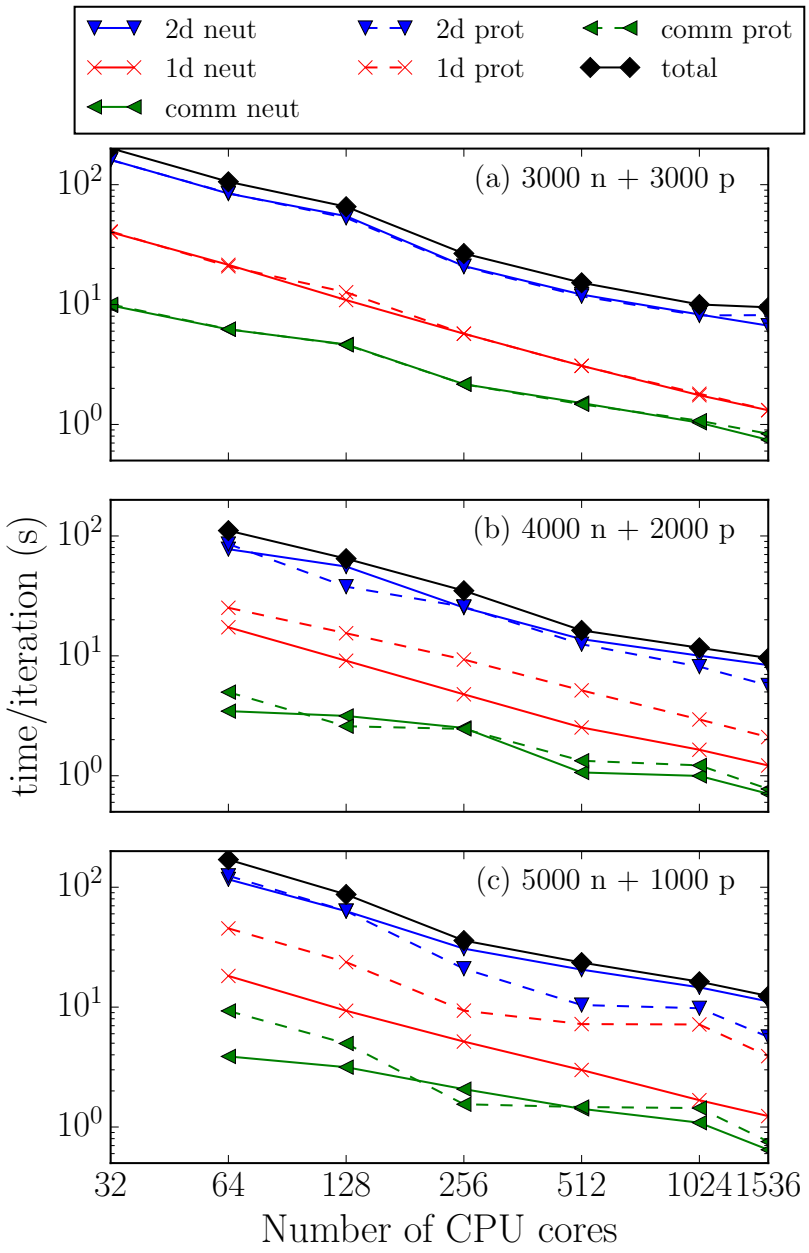


Figure 2.9: Times per iteration for neutron and proton processor groups, illustrating the load balance for the 3000 neutrons and 3000 protons (a), 4000 neutrons and 2000 protons (b), and 5000 neutrons and 1000 protons (c) systems using the $L = 48\,\mathrm{fm}$ grid. Due to memory constraints the latter two cases cannot be calculated using 32 CPU.

In this case, according to our load balancing scheme, the number of cores in the neutron group will be roughly 4x larger than the number of cores in the proton group because we distribute the cores based on the ratio of the square of the number of particles in each group. We observe that all three major parts are almost equally balanced for up to 1024 cores, but 2D calculations for neutrons is slightly more expensive on 1536 cores. A more detailed examination reveals that this difference is due to the eigendecomposition times in steps 3b-3c. However, it is relatively minor compared to the total execution time per iteration.

Note that the times for the steps with 1d distribution show some variation for the system with 4000 neutrons and 2000 protons. This is due to the fact that we split the available cores into neutron and proton groups based on the cost of steps with 2D data distributions. Consequently, 1D distributed steps take more time on the proton processor group, but this difference is negligible in comparison to the cost of 2D distributed steps.

In Figure 2.9(c) results for a more challenging case with 5000 neutrons and 1000 protons are presented. In this case the majority of the available cores are assigned to the neutron group - more precisely, the ratio between the sizes of the two groups is roughly 25. We observe that 1D calculations take significantly more time for protons in this case, but any potential load imbalances are compensated by the reduced 2D calculation times for protons.

A further inspection of the execution time of each step for the 5000 neutron and 1000 proton system is given in Figure 2.10. This inspection reveals that time needed for neutrons and protons mainly differ for step 3b-3c and step 3d due to the large difference between neutron and proton counts. But these difference are not significant compared to the other computationally heavy steps which are well load balanced. As shown in Table 2.6, our implementation still achieves about 50% strong scaling efficiency on 1536 cores for this challanging case with 5000 neutrons and 1000 protons.

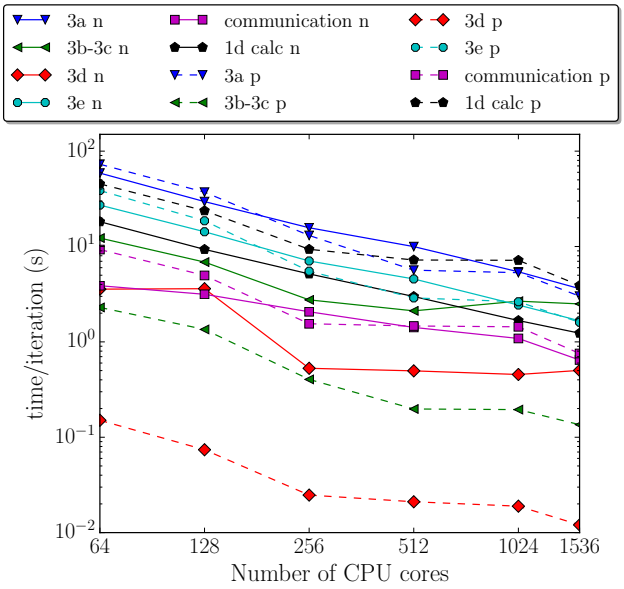


Figure 2.10: A detailed breakdown of per iteration times for neutron and proton processor groups, illustrating the load balance for the 5000 neutrons and 1000 protons system.

Table 2.6: Scalability of MPI-only version of Sky3D for the 5000 neutrons and 1000 protons system using the $L = 48\,\mathrm{fm}$ grid. Time is given in seconds, and efficiency (eff) is given in percentages.

	calc. matrix		**recombine**		**diag+Löwdin**		**Total**	
cores	**time**	**eff**	**time**	**eff**	**time**	**eff**	**time**	**eff**
64	72.9	100	38.6	100	2.3	100	170	100
128	37.1	98.1	18.6	103.4	1.35	85	87	97.6
256	15.7	115.8	7.1	136.9	2.75	20.82	35.9	118.3
512	10	91.3	4.6	105.7	2.1	13.6	23.5	90.3
1024	5.4	83.9	2.4	99.2	2.7	5.3	16.3	65.2
1536	3.6	84	1.7	96.7	2.5	3.8	12.4	57.2

2.5.5 Conclusion of this work

In this work, we described efficient and scalable techniques used to parallelize Sky3D, a nuclear DFT solver that operates on an equidistant grid in a pure MPI framework as well as a hybrid MPI/OpenMP framework. By carefully analyzing the computational motifs in each step and data dependencies between different steps, we used a 2D decomposition scheme for Sky3D kernels working with matrices, while using a 1D scheme for those performing computations on wave functions. We presented load balancing techniques which can efficiently leverage high degrees of parallelism by splitting available processors into neutron and proton groups. We also presented algorithmic techniques that reduce the total execution time by overlapping diagonalization and orthogonalization steps using subgroups within each processor group. Detailed performance analysis on a multi-core architecture (Cori at NERSC) reveal that parallel Sky3D can achieve good scaling to a moderately large number of cores. Contrary to what one might naively expect, the MPI-only implementation outperforms the hybrid MPI/OpenMP implementation, mainly because ScaLAPACK's eigedecomposition routines perform worse in the hybrid parallel case. For larger core counts, the disparity between the two implementations seems to be less pronounced. As a result of detailed performance evaluations, we have observed that 256 to 1024 processors are reasonably efficient for nuclear pasta simulations and we consider these core counts for production runs, depending on the exact calculation.

Using the new MPI parallel Sky3D code, we expect that pasta phases can be calculated for over 10,000 nucleons in a fairly large box using a quantum mechanical treatment. As a result, we expect to reach an important milestone in this field. We plan to calculate properties of more complicated pasta shapes and investigate defects in pasta structures which occur in

large systems.

This work motivated us to work with large scale sparse matrices in the newer architectures. We first started with a blocked eigensolver and tried to optimize its performance. I discuss this in more detail in the next chapter.

Chapter 3

OPTIMIZATION IN LARGE SCALE DISTRIBUTED SPARSE MATRICES

3.1 Eigenvalue Problem in CI Calculations

Nuclear physics faces the multiple hurdles of a very strong interaction, three-nucleon interactions, and complicated collective motion dynamics. The eigenvalue problem arises in nuclear structure calculations because the nuclear wave functions Ψ are solutions of the many-body Schrödinger's equation expressed as a Hamiltonian matrix eigenvalue problem, $H\Psi = E\Psi$.

In the CI approach, both the wave functions Ψ and the Hamiltonian H are expressed in a finite basis of Slater determinants (anti-symmetrized product of single-particle states, typically based on harmonic oscillator wave functions). Each element of this basis is referred to as a many-body basis state. The representation of H within an A-body basis space, using up to k-body interactions with kA, results in a sparse symmetric matrix $\hat{H}$ Thus, in CI calculations, Schrödinger's equation becomes an eigenvalue problem, where one is interested in the lowest eigenvalues (energies) and their associated eigenvectors (wave functions). A specific many-body basis state corresponds to a specific row and column of the Hamiltonian

matrix. A nonzero in the Hamiltonian matrix indicates the presence of an interaction between either the same or different many-body basis states. Both the total number of many-body states N (the dimension of $\hat{H}$) and the total number of nonzero matrix elements in $\hat{H}$ are controlled by the number of nuclear particles, the truncation parameter $N_{\max}$, which is the maximum number of HO quanta above the minimum for a given nucleus (see Fig. 3.1), and by the maximum number of particles allowed to interact simultaneously in the Hamiltonian in Eq, (3). Higher $N_{\max}$ values yield more accurate results, but at the expense of an exponential growth in problem size. Many nuclear applications seek to reach at least an $N_{\max}$ of 10 in order to establish a sequence of values of observables as a function of $N_{\max}$ in order to estimate exact answers through extrapolations to infinite $N_{\max}$.

3.2 Motivation and CI Implementation

As the load balancing issue and communication overheads on distributed memory systems have been addressed in our previous work [78, 103, 104], here we mainly focus on the performance of the thread-parallel computations within a single MPI rank. Conventionally, in MFDn as well as in other CI codes, the Lanczos algorithm is used due to its excellent convergence properties. However, locally optimal block preconditioned conjugate gradient (LOBPCG) [30], a block eigensolver, is an attractive alternative for a number of reasons. First, the LOBPCG algorithm allows effective use of many-body wave functions from closely related model spaces (e.g. smaller basis, or different single-particle wave functions) to be used as good initial guesses. Second, the LOBPCG algorithm can easily incorporate an effective preconditioner which can often be constructed based on physics insights to significantly improve convergence. Third and most relevant to our focus in this paper, the LOBPCG

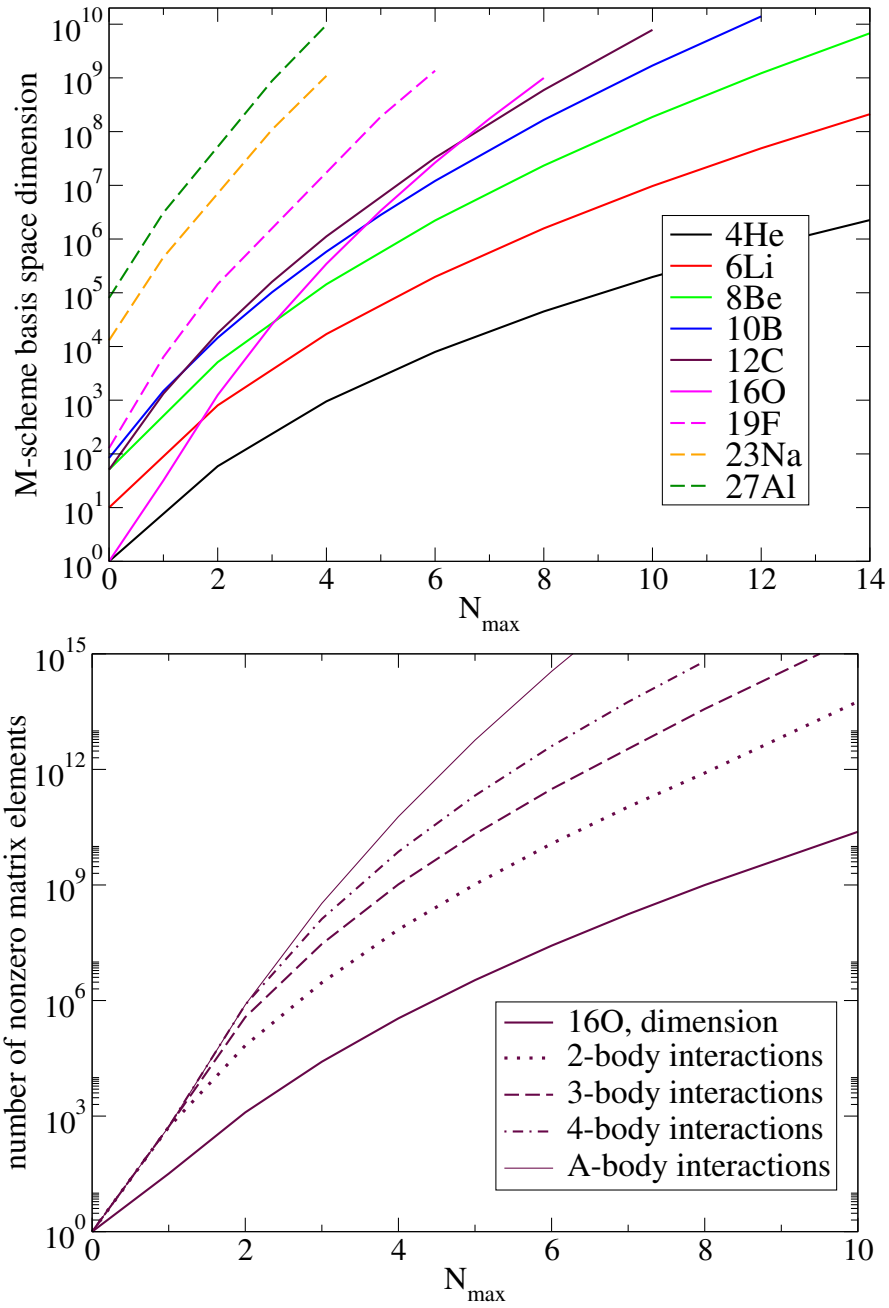


Figure 3.1: The dimension and the number of non-zero matrix elements of the various nuclear Hamiltonian matrices as a function of the truncation parameter $N_{\max}$. While the bottom panel is specific to ^{16}O, it is also representative of a wider set of nuclei [2, 3].

algorithm naturally leads to an implementation with a high arithmetic density, as the main computational kernels involved are the multiplication of a sparse matrix with multiple vectors, and level-3 BLAS on dense vector blocks, as opposed to the SpMVs and level-1 BLAS operations that are the building blocks in Lanczos. Finally, although not studied here, we note that the potential benefits of a block eigensolver can be even more significant for CI implementations based on on-the-fly computation of the Hamiltonian.

Alg. 3 gives the pseudocode for a simplified version of the LOBPCG algorithm without preconditioning. LOBPCG is a subspace iteration method that starts with an initial guess of the eigenvectors (Ψ_0) and refines its approximation at each iteration (Ψ_i). R_i denotes the residual associated with each eigenvector and P_i contains the direction information from the previous step. Hence, in Alg. 3, Ψ_i, R_i and P_i correspond to dense blocks of vectors. To ensure good convergence, the dimension of the initial subspace, m, is typically set to 1.5 to 2 times the number of desired eigenpairs nev. For numerical stability, the converged eigenpairs are locked, i.e., m gets smaller as the algorithm progresses.

In the rest of this paper, we present our techniques to improve the efficiency of sparse matrix computations, and dense vector block operations that constitute the key kernels for LOBPCG. We then evaluate the impact of our techniques in real-world problems and compare the performance of our optimized LOBPCG implementation with a Lanczos-based solver.

The CI method is implemented in MFDn [2, 3]. A major challenge in CI is the massive size of the matrix $\hat{H} \in \mathbb{R}^{N \times N}$, where N can be in the range of several billions and the total number of nonzeros can easily exceed trillions. Since only the low-lying eigenpairs are of interest, iterative eigensolvers are used to tame the computational cost [103, 104]. As the identification of nonzeros in $\hat{H}$ and calculation of their values is very expensive, MFDn

constructs the sparse matrix only once and preserves it throughout the computation. To accelerate matrix construction and reduce the memory footprint, only half of the symmetric $\hat{H}$ matrix is stored in the distributed memory available. A unique 2D triangular processor grid is then used to carry out the computations in parallel [103, 104]. In this scheme, a "diagonal" processor stores only the lower triangular part of a sub-matrix along the diagonal of $\hat{H}$. Each "non-diagonal" processor, a processor that owns a sub-matrix from either the lower or the upper half of $\hat{H}$, is assigned the operations related to the transpose of that sub-matrix. A well-balanced distribution of the nonzeros among processors is ensured through efficient heuristics [78]. Exploiting symmetry in MFDn demands SpMV^T (SpMM^T) in addition to the SpMV (SpMM) operations, and thus data structures that efficiently implement both operations. The accuracy from single-precision arithmetic is in general sufficient to calculate the physical observables. Hence, in MFDn, the Hamiltonian matrix is stored in single-precision to further reduce the memory footprint.

3.3 Multiplication of the Sparse Matrix with Multiple Vectors (SpMM)

To exploit symmetry in a block eigensolver, each process must perform a conventional SpMM ($Y = AX$), as well as a transpose operation SpMM^T ($Y = A^T X$), where A corresponds to the local partition of $\hat{H}$, X to a row partition of Ψ_i. Y is the output vector block in each case. The number of rows and columns of A are typically very close to each other, therefore, for simplicity, we take A to be a square matrix of size $n \times n$. Both X and Y are dense vector blocks of dimensions $n \times m$. As SpMM and SpMM^T are performed in separate phases of the MPI parallel algorithm [104], we use the same input/output vectors to simplify the

presentation.

Naively, one can realize SpMM by storing the vector blocks in column-major order and applying one SpMV to each column of X. However, to exploit spatial locality, a row-major layout should be preferred for vector blocks X and Y. This format also ensures good data locality for the tall-skinny matrix operations of LOBPCG. Thus, the simplest SpMM implementation can be implemented as an extension of SpMV where the operation on scalar elements $y_i = \sum A_{i,j}x_j$ becomes an operation on m-element vectors $Y_i = \sum A_{i,j}X_j$. The input and output vectors can be aligned to 32-byte boundaries for efficient vectorization of the m-element loops. This operation can be implemented by looping over each nonzero $A_{i,j}$.

3.4 Matrix Storage Formats

The most common sparse matrix storage format is compressed sparse rows (CSR) in which the nonzeros of each matrix row are stored consecutively as a list in memory. 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. An additional index array is used to keep the column indices of each nonzero. Nonzero values and column indices are stored in separate arrays of length nnz, and the row pointers array is of length $n+1$. For single-precision sparse matrices whose *local* row and column indices can be addressed with 32-bit integers (i.e., $n \leq 2^{32} - 1$), the storage cost for the CSR format is $8nnz + 4n + 1$. One may reuse matrices stored in the CSR format for the SpMM^T operation by reinterpreting row pointers and column indices as column pointers and row indices, respectively. Such an interpretation would correspond to a compressed sparse column (CSC) representation in which one operates on columns rather than rows to implement the SpMM^T operation.

Large vector blocks (with $4 \leq m \leq 50$, and $n > 10^6$) can potentially prevent a CSR based SpMM implementation from taking full advantage of the locality in vector blocks depending on the matrix sparsity structure. After a few rows, it is likely that vector data will have been evicted from the L2 cache, while after a few hundred rows, it is very likely that data will have been evicted from even the last level L3 cache. Moreover, in a thread-parallel SpMM^T, CSC's scatter operation on thread-private output vectors (necessary to prevent race conditions) coupled with the reduction required for partial thread results can significantly impede performance [2]. Thus, it is imperative that we adopt a data structure that can attain good locality for the vector blocks and does not suffer from the performance penalties associated with the CSR and CSC implementations.

Our data structure for storing sparse matrices is a variant of the compressed sparse blocks (CSB) format [77]. For a given block size parameter β, CSB nominally partitions the $n \times n$ local matrices 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). Therefore, for $\beta = \sqrt{n}$, the storage cost of CSB matches the storage cost of traditional formats such as CSR. In addition, CSB automatically enables cache blocking [105]. In CSB format, each $\beta \times \beta$ block is independently addressable through a 2D array of pointers. SpMM operation can then be performed by processing this 2D array by rows, while SpMM^T can simply be realized by processing it via columns.

The formal CSB definition does not specify how the nonzeros are stored within a block. An existing implementation of CSB for sparse matrix–vector (SpMV) and transpose sparse matrix–vector (SpMV^T) multiplication stores nonzeros within each block using a space filling curve to exploit data locality and enable efficient parallelization of the blocks themselves [77].

3.5 Methodology and Optimization

CSR/OpenMP: Our baseline SpMM implementation uses the CSR format. SpMM operation was threaded using an OpenMP parallel for loop with dynamic scheduling over the matrix rows. SpMM^T operation was threaded over columns (which are simply reinterpretations of CSR rows for the transpose) where each thread uses a private copy of the output vector block to prevent race conditions. Private copies are then reduced (using thread parallelism) to complete the SpMM^T operation.

Rowpart/OpenMP: On multi-core CPUs with several cores, the CSR implementation above is certainly not suitable for performing SpMM^T on large sparse matrices. Thread private copies of the output vector requires an additional $\mathcal{O}(nmP)$ storage, where P denotes the number of threads. In fact, more storage space than the sparse matrix itself could be needed for even small values of m for matrices with only tens of nonzeros per row. In terms of performance, thread private output vectors may adversely effect data reuse in the last level of cache, and requires an expensive post-processing step. Therefore we implemented the *Rowpart* algorithm. It is identical to our baseline CSR implementation for SpMM, but for a memory efficient and load balanced SpMM^T, it preprocesses the columns of the transpose matrix and determines row indices for each thread such that row partitions assigned to threads contain (roughly) equal number of nonzeros. Each thread then maintains a starting and ending index of its row partition boundaries per column. Extra storage space cost of *Rowpart* is only $\mathcal{O}(nP)$ and the preprocessing overheads are insignificant when used in an iterative solver.

CSB/OpenMP: Our new parametrized implementation for SpMM and SpMM^T, CSB/OpenMP, is based on the CSB format. As the other implementations, CSB/OpenMP is written in For-

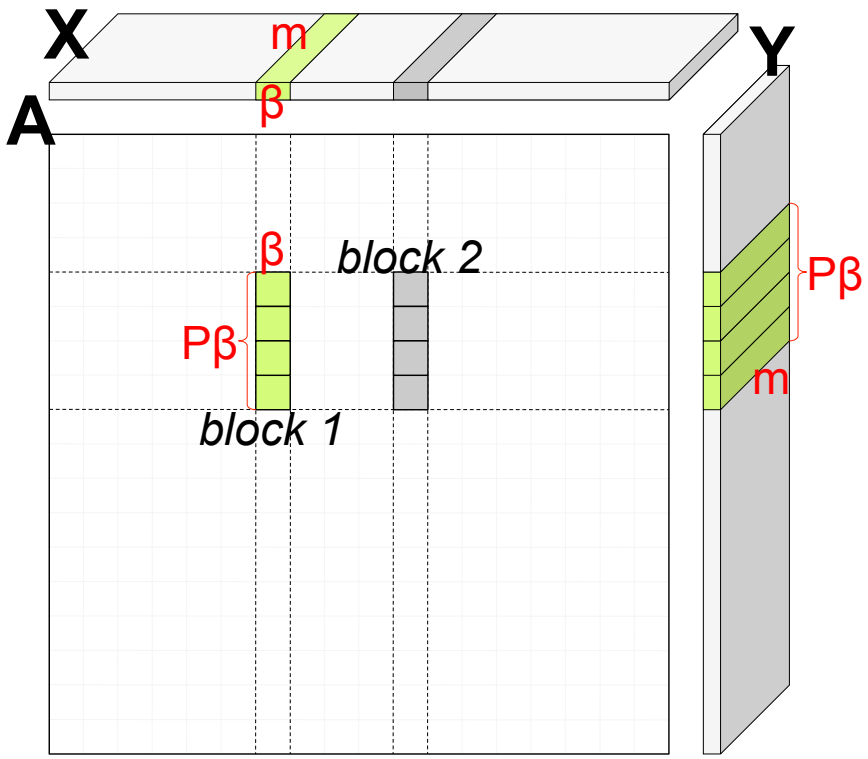


Figure 3.2: Overview of the SpMM operation with $P = 4$ threads. The operation proceeds by performing all $P\beta \times \beta$ local SpMM operations Y=AX+Y one blocked row at a time. The operation $A^T X$ is realized by permuting the blocking ($\beta \times P\beta$ blocks).

tran using OpenMP. As shown in Fig. 3.2, the matrix is partitioned into $\beta \times \beta$ blocks that are stored in coordinate format (COO) with 16-bit indices and 32-bit single-precision values. SpMM is threaded over individual rows of blocks (corresponding to $\beta \times n$ slices of the matrix), which creates block rows of size $P\beta \times n$. In SpMM^T, threads sweep through block columns of size $n \times P\beta$ and use the COO's row indices as column indices and vice versa. We tune for the optimal value of β for each value of m for a given matrix.

CSB/Cilk: For comparisons with the original Cilk-based CSB, we extended the fully parallel SpMV and SpMV^T algorithms [77] in CSB to operate on multiple vectors. We used a vector of `std::array`'s, a compile-time fixed-sized variant of the built-in arrays for storing X and Y. This effectively creates tall-skinny matrices in row major order. CSB heuristically determines the block parameter β, considering the parallel slackness, size of the L2 cache,

and the addressability by 16-bit indices. The parameter β chosen for the single vector cases presented in Sect. 3.8 was 16,384 or 8,192 (depending on the matrix), and it got progressively smaller all to way to $\beta = 1024$ as m increases (due to increased L2 working set limitations).

SpMM and SpMM^T implemented using CSB/Cilk employ three levels of parallelism. For SpMM (the transpose case is symmetric), it first parallelizes across rows of blocks, then within dense rows of block using temporary vectors, and finally within sufficiently dense blocks if needed. Additional parallelization costs of second and third levels are amortized by performing them on sufficiently dense rows of block and individual blocks that threaten load balance. Such blocks and rows of blocks can be shown to have enough work to amortize the parallelization overheads. Our CSB/OpenMP implementation differs from the CSB/Cilk implementation in that CSB/OpenMP does not parallelize within individual rows/columns of blocks or within dense blocks. Rather, CSB/OpenMP partitions the sparse matrix into a sufficiently large number of rows/columns of blocks by choosing an appropriate β. Dynamic scheduling is leveraged to ensure load balance among threads.

In all implementations (CSR, Rowpart, CSB/OpenMP, CSB/Cilk), innermost loops ($Y_i = \sum A_{i,j}X_j$ for SpMM and $Y_j = \sum A_{i,j}X_i$ for SpMM^T) were manually unrolled for each m value. In Fortran `!$dir simd` directives and in C `#pragma simd always` pragmas were used for vectorization. We inspected the assembly code to ensure that packed SIMD/AVX instructions were generated for best performance. To minimize TLB misses, we used large pages during compilation and runtime.

3.6 An Extended Roofline Model for CSB

Conventional wisdom suggests that SpMV performance is a function of STREAM bandwidth and data movement from compulsory misses on matrix elements. Then the simplified Roofline model [75] provides a lower bound to SpMV time by $8 \cdot nnz/BW_{stream}$ for single-precision CSR matrices [106]. This simple analysis may lead one to conclude that performing SpMV's on multiple right-hand sides (SpMM) is essentially no more expensive than performing one SpMV. Unfortunately, this is premised on three assumptions — (i) compulsory misses for vectors are small compared to the matrix, (ii) there are few capacity misses associated with the vectors, and (iii) cache bandwidth does not limit performance. The first premise is certainly invalidated once the number of right-hand sides reaches half the average number of nonzeros per row (assuming an 8-byte total space for single-precision nonzeros, 4-byte single-precision vector elements, and a write-allocate cache). The second would be true for low-bandwidth matrices with working sets smaller than the last level cache. The final assumption is highly dependent on microarchitecture, matrix sparsity structure, and the value of m. We observe that for MFDn matrices and moderate values of m, this conventional wisdom fails to provide a good performance bound.

In this paper, we construct an extended Roofline performance model that captures how cache locality and bandwidth interact to tighten the performance bound for CSB-like sparse kernels. Let us consider three progressively more restrictive cases: vector locality in the L2, vector locality in the L3, and vector locality in DRAM. As it is highly unlikely a $\beta \times \beta$ block acting on multiple vectors attains good vector locality in the tiny L1 caches, we will ignore this case. Although potentially an optimistic assumption, we assume we may always hit peak L2, L3, or DRAM bandwidth with the caveat that, on average, we overfetch 16 bytes.

First, if we see poor L1 locality for the block of vectors but good L2 locality, then for each nonzero, CSB must read 8 bytes of nonzero data, $4m$ bytes of the source vector, and $4m$ bytes of the destination vector. It may then perform $2m$ flops and write back $4m$ bytes of destination data. Thus we perform $2m$ flops and must move $8 + 12m$ bytes ideally at the peak L2 bandwidth. Ultimately, this would limit SpMM performance to 6.6 GFlop/s per core, or about 80 GFlop/s per chip on Edison which has an L2 cache bandwidth of 40 GB/s per core (see Sect. 3.8.1). One should observe that we have assumed high locality in L2. As this is unlikely, this bound is rather loose.

Unfortunately, static analysis of sparse matrix operations has its limits. In order to understand how locality in the L3 and L3 bandwidth constrain performance we implemented a simplified L2 cache simulator to calculate the number of capacity misses associated with accessing X and Y. For each $\beta \times \beta$ block the simulator tries to estimate the size of the L2 working set based on the average number of nonzeros per column. When the average number of nonzeros per column is less than one, then the working set is bounded by $(8m + 32) \cdot nnz$ bytes — each nonzero requires a block of the source vector and a block of the destination vector plus overfetch. When the average number of nonzeros per column reaches one, we saturate the working set at $8m\beta$ bytes — full blocks of source and destination vectors. If the working set is less than the L2 cache capacity we must move $8 \cdot nnz + 4m\beta$ bytes when the number of nonzeros per column is equal to or greater than 1 and $(8 + 4m + 16) \cdot nnz$ bytes (but never more than $8 \cdot nnz + 4m\beta$ bytes) when the number of nonzeros per column is less than 1 (miss on the nonzero and the source vector). If the working set exceeds the cache capacity, then we forgo any assumptions on reuse of X or Y in the L2 and incur $(8 + 4m + 16) \cdot nnz + 8m\beta$ bytes of data movement. So, this bound on data movement depends on both m and the input matrix.

Finally, let us consider the bound due to a lack of locality in L3 and finite DRAM bandwidth. As shown in Fig. 3.2, CSB matrices are partitioned into blocks of size $\beta \times \beta$, and P threads stream through block rows (or block columns for SpMM^T) performing local SpMM operations on blocks of size $P\beta \times \beta$. If one thread (a $\beta \times \beta$ block) gets ahead of the others, then it will likely run slower as it is reading X from DRAM while the others are reading X from the last level cache. Thus, we created a second simplified cache simulator to track DRAM data movement which tracks how a chip processes each $P\beta \times \beta$ block, rather than tracking how individual cores process their $\beta \times \beta$ blocks. Our model streams through the block rows of a matrix (like in Fig. 3.4) and for each nonzero $P\beta \times \beta$ block examines its cache to determine whether the corresponding block of X is present. If it misses, then it fetches the entire block and increments the data movement tally. If the requisite cache working set exceeds the cache capacity, then we evict a block (LRU policy). Finally, we add the nonzero data movement and the read-modify-write data movement associated with the output vector block Y ($8nm$ bytes).

Ultimately, the combined estimates for DRAM, L2, and L3 data movement provide us a narrow range of expected SpMM performance as a function of m. For low arithmetic intensity (small m), the Roofline suggests we would be DRAM-bound, but the Roofline plateaus, it is likely to do so because of either L2 or L3 bandwidth, rather than the peak FLOP rate. In the future, we plan to use this lightweight simulator as a model-based replacement for the expensive empirical tuning of β.

3.7 Kernels with Tall and Skinny Matrices

Besides sparse matrix operations, all block methods require operations on the dense blocks of vectors themselves, which we denote as tall-skinny matrix computations owing to the shape of the multiple vector structures involved. The LOBPCG algorithm mainly involves inner product and linear combination operations. Performance in these kernels are critical for the overall eigensolver performance for three reasons. First, an optimized SpMM algorithm incurs a significantly reduced cost on a per SpMV basis. Second, while the per iteration cost of vector operations is $\mathcal{O}(N)$ for Lanczos-like solvers, in block methods these operations cost $\mathcal{O}(Nm^2)$ which grows quickly with m. Finally, and most importantly, the LOBPCG algorithm involves several of these operations in each iteration. For example, computing E_i and updating the residual R_i before the Rayleigh–Ritz procedure in Alg. 3 requires an inner product and a linear combination, respectively. The Rayleigh–Ritz procedure itself requires computing the overlap matrix between each pair of the current Ψ_i, R_i, P_i vectors themselves, as well as their overlap with the vector blocks from the previous iteration, leading to a total of 18 inner product operations. Following the Rayleigh–Ritz procedure is the updates to the Ψ_i, R_i, P_i blocks of vectors for the next iteration, which require computing linear combinations. There are a total of 10 such linear combination operations per Rayleigh–Ritz procedure.

Note that the Hamiltonian matrix in MFDn is partitioned into a 2D triangular grid, and during parallel SpMM the X and Y vector blocks are shared/aggregated among the processes in the row/column groups of this triangular grid [103, 104]. Efficient parallelization of LOBPCG operations requires further partitioning X and Y among processes in the same row groups, resulting in smaller local blocks of vectors of size $l \times m$ ($l = n/p_{row}$, where

p_{row} is the number of process in a row of the triangular grid). In fact, as shown in Alg. 3, each process has to keep several matrices of size $l \times m$ due to the need for storing the residual R and previous direction P information locally. Since we are mainly interested in the performance of the kernels, we will generically denote the local $l \times m$ blocks of vectors as V and W. We observe that achieving numerical stability with LOBPCG requires using double-precision arithmetic for most MFDn calculations. Therefore, we convert the result of the sparse matrix operations into double-precision before starting LOBPCG computations.

We denote the inner product of two blocks of vectors V and W as V^TW, and the linear combination of the vectors in a block by a small square coefficient matrix C as VC. Both V^TW and VC have high arithmetic intensities. Specifically, for both kernels the number of flops is $\mathcal{O}(lm^2)$ and the total data movement is $\mathcal{O}(lm)$, yielding an arithmetic intensity of $\mathcal{O}(m)$. These kernels can be implemented as level-3 BLAS operations using optimized math libraries such as Intel's MKL or Cray's LibSci. While one would expect to achieve a high percentage of the peak performance (especially for large m), as demonstrated in Sect. 3.8.7, both MKL and LibSci perform poorly for these kernels. This is most likely due to the unusual shape of the matrices involved (typically $l \gg m$ for large-scale computations).

To eliminate the performance bottlenecks with the V^TW and VC computations, we developed custom thread-parallel implementations for them. Fig. 3.3 gives an overview of our V^TW implementation. We store V and W in row-major order, consistent with the storage of the vector blocks in sparse matrix computations. We partition V and W into small row blocks of size $s \times m$, and compute the inner product V^TW by aggregating the results of (vendor tuned) `dgemm` operations between a row block in V and the corresponding one in W. The loop over $s \times m$ blocks is thread parallelized using OpenMP. To achieve load balance with minimal scheduling overheads, we use the `guided` scheduling option. Race

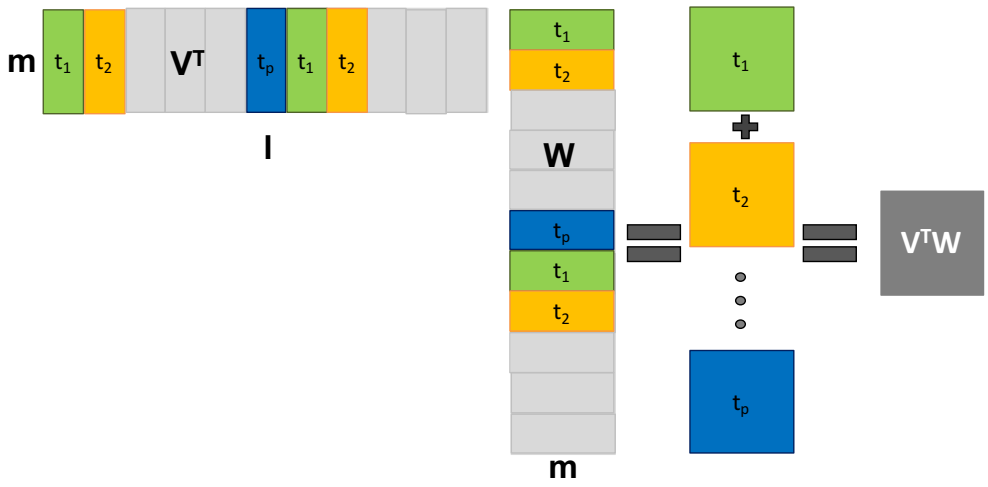


Figure 3.3: Performance in GFlop/s for vector block inner product, V^TW, and vector block scaling, VX kernels using Intel MKL and Cray libsci libraries on a Cray XC30 system (Edison @ NERSC).

conditions in the output matrix are resolved by keeping a thread-private buffer matrix of size $m \times m$. We perform a reduction, which is also thread-parallel, over the buffer matrices to compute the final overlap matrix.

Our custom VC kernel is implemented similarly by partitioning V into row blocks. In this case, C is a square matrix of size $m \times m$ which is read-shared by all threads. Again, the loop over the $s \times m$ blocks of V is thread parallelized with `guided` scheduling. To prevent race conditions, we let each thread perform the computation using the full C matrix, i.e., a `dgemm` between matrices of size $s \times m$ and $m \times m$. Each thread then uniquely produces the corresponding set of s output rows.

3.8 Performance Evaluation

We now present a comprehensive evaluation of our methods for sparse and tall-skinny matrix computations.

3.8.1 Experimental setup

We use a series of computations with MFDn. As the overall execution time is dominated by on-node computations, we begin with single-socket performance evaluations of SpMM (Sect. 3.8.2) and LOBPCG computations (Sect. 3.8.7). In Sect. 3.8.8, we inspect the resulting solver's performance in a distributed memory setting.

MFDn matrices: We identified three test cases, "Nm6", "Nm7" and "Nm8", which are matrices corresponding to the ^{10}B Hamiltonian at $N_{\max} = 6$, 7, and 8 truncation levels, respectively. The actual Hamiltonian matrices are very large and therefore are nominally distributed across several processes in the actual calculations. For a given nucleus, the sparsity of $\hat{H}$ is determined by (i) the underlying interaction potential, and (ii) the $N_{\max}$ parameter. We used a 2-body interaction potential; a 3-body or a higher order interaction potential would result in denser matrices presenting more favorable conditions for achieving computational efficiency. For a given nucleus and interaction potential, increasing the $N_{\max}$ value reduces the density of nonzeros in each row, thereby allowing us to evaluate the effectiveness of our techniques on a range of matrix sparsities.

Each process on a distributed memory execution receives a different sub-matrix of the Hamiltonian, but these sub-matrices have similar sparsity structures. For simplicity and consistency, we use the first off-diagonal processor's sub-matrix as our input for single-socket evaluations. Table 3.1 enumerates the test matrices used in this paper. Note that the test matrices have millions of rows and hundreds of millions of nonzeros. As discussed in Sect. 3.3, we use the compressed sparse block (CSB) format [77] in our optimized SpMM implementation. Therefore a sparse matrix is stored in blocks of size $\beta \times \beta$. When blocked with $\beta = 6000$, we observe that both the number of block rows and the average number of

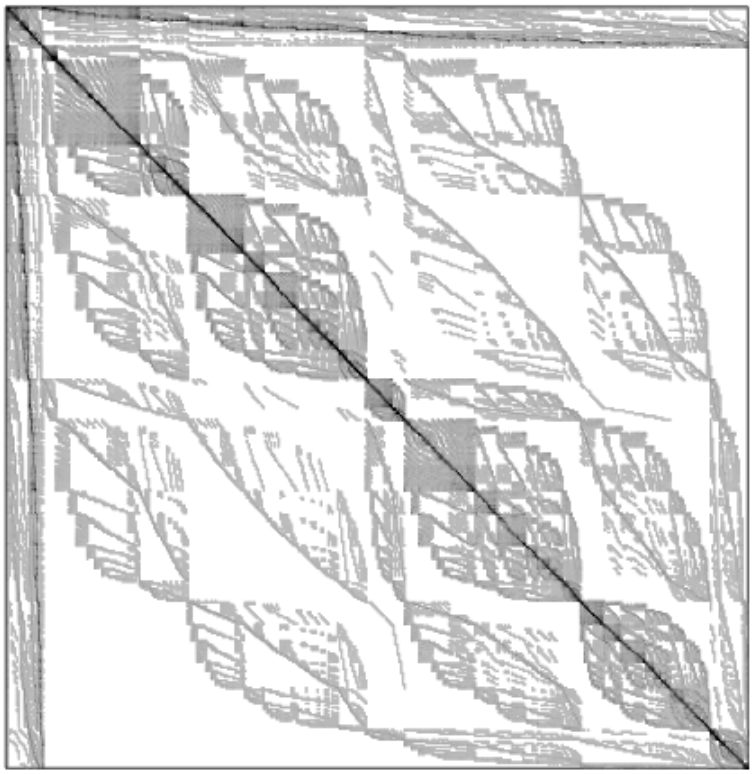


Figure 3.4: Sparsity structure of the local Nm6 matrix at process 1 in an MFDn run with 15 processes. A block size of $\beta = 6000$ is used. Each dot corresponds to a block with nonzero matrix elements in it. Darker colors indicate denser nonzero blocks.

Matrix	**Nm6**	**Nm7**	**Nm8**
Rows	2,412,566	4,985,944	7,583,735
Columns	2,412,569	4,985,944	7,583,938
Nonzeros (nnz)	429,895,762	648,890,590	592,325,005
Blocked Rows	403	831	1264
Blocked Columns	403	831	1264
Average nnz per Block	7991	4191	2311

Table 3.1: MFDn matrices (per-process sub-matrix) used in our evaluations. For the statistics in this table, all matrices were cache blocked using $\beta = 6000$.

nonzeros per nonzero block remain high. Fig. 3.4 gives a sparsity plot of the Nm6 matrix at the block level, where each nonzero block is marked by a dot whose intensity represents the density of nonzeros in the corresponding block. For our test matrices, 41–64% of these blocks are nonzero. We observe a high variance on the number of nonzeros per nonzero block.

Vector block sizes: In nuclear physics applications, up to 20–25 eigenvalues are needed, and 5–15 eigenpairs will be sufficient. In LOBPCG, to ensure a rich representation of the subspace and ensure good convergence, the number of basis vectors m needs to be set to 1.5 to 2 times the number of desired eigenvectors nev, As the algorithm proceeds and eigenvectors converge, converged eigenpairs are deflated (or locked) and the subspace shrinks. Therefore, we examine the performance of our optimized kernels for values of m in the range 1 to 48.

Computing platforms: We primarily use high-end multi-core processors (Intel Xeon) for performance studies. However, the energy efficiency requirements of HPC systems point to an outlook where many-core processors will play a prominent role. To guide our future efforts in this area, we conduct performance evaluations on an Intel Xeon Phi processor as well.

Our multi-core results come from the Cray XC30 MPP at NERSC (Edison) which contains more than 10 thousand, 12-core Xeon E5 CPUs. Each of the 12 cores runs at 2.4 GHz and is capable of executing one AVX (8×32-bit SIMD) multiply and one AVX add per cycle and includes both a private 32 KB L1 data cache and a 256 KB L2 cache. Although the per-core L1 bandwidth exceeds 75 GB/s, the per-core L2 bandwidth is less than 40 GB/s. There are two 128-bit DDR3-1866 memory controllers that provide a sustained STREAM bandwidth of 52 GB/s per processor. The cores, the 30MB last level L3 cache and memory controllers are interconnected with a complex ring network-on-chip which can sustain a bandwidth of about 23 GB/s per core.

Our many-core results have been obtained at the Babbage testbed at NERSC. Intel Xeon Phi (MIC) cards are connected to the host processor through the PCIe bus and contain 60 cores running at 1 GHz, with 4 hardware threads per core. Each MIC card has an on-device 8 GB of high bandwidth memory (320 GB/s). Cores are interconnected by a high-speed

Processor	Xeon E5-2695 v2	Xeon Phi 5110P
Core	**Ivy Bridge**	**Pentium P54c**
Clock (GHz)	2.4	1.05
Data Cache (KB)	32+256	32 + 512
Memory-Parallelism	HW-prefetch	SW-prefetch + MT
Cores/Processor	12	60
Threads/Processor	24[1]	4
Last-level L3 Cache	30 MB	–
SP GFlop/s	460.8	2,022
DP GFlop/s	230.4	1,011
Aggregate L2 BW	480 GB/s	960 GB/s[2]
Aggregate L3 BW	276 GB/s	–
STREAM BW[3]	52 GB/s	320 GB/s
Available Memory	32 GB	8 GB

Table 3.2: Overview of Evaluated Platforms. [1] *With hyper threading, but only 12 threads were used in our computations.* [2] *Based on the* `saxpy1` *benchmark in [1].* [3] *Memory bandwidth is measured using the STREAM copy benchmark.*

bidirectional ring. Each core has a 32 KB L1 data cache and 512 KB L2 cache locally with high speed access to all other L2 caches to implement a fully coherent cache. Note that there is not a shared last level L3 cache on the MIC cards. Each core supports 512-bit wide AVX-512 vector ISA that can execute 8 double-precision (or 16 single-precision or integer) operations per cycle. With Fused Multiply-Add (FMA), this amounts to 16 DP or 32 SP FLOPS/cycle. Peak performance for each MIC card is 1 TFlop/s (in DP). The characteristics of both processors are summarized in Table 7.1.

We use the Intel Fortran compiler with flags `-fast -openmp`. For comparison with the original CSB using Intel Cilk Plus, we use the Intel C++ compiler with flags `-O2 -no-ipo -parallel -restr` As Intel Cilk Plus uses dynamically loaded libraries not natively supported by the Cray operating system, we use Cray's cluster compatibility mode that causes only a small performance degradation. The Xeon Phi's performance was evaluated in the *native* mode, enabled through the `-mmic` flag in Intel compilers.

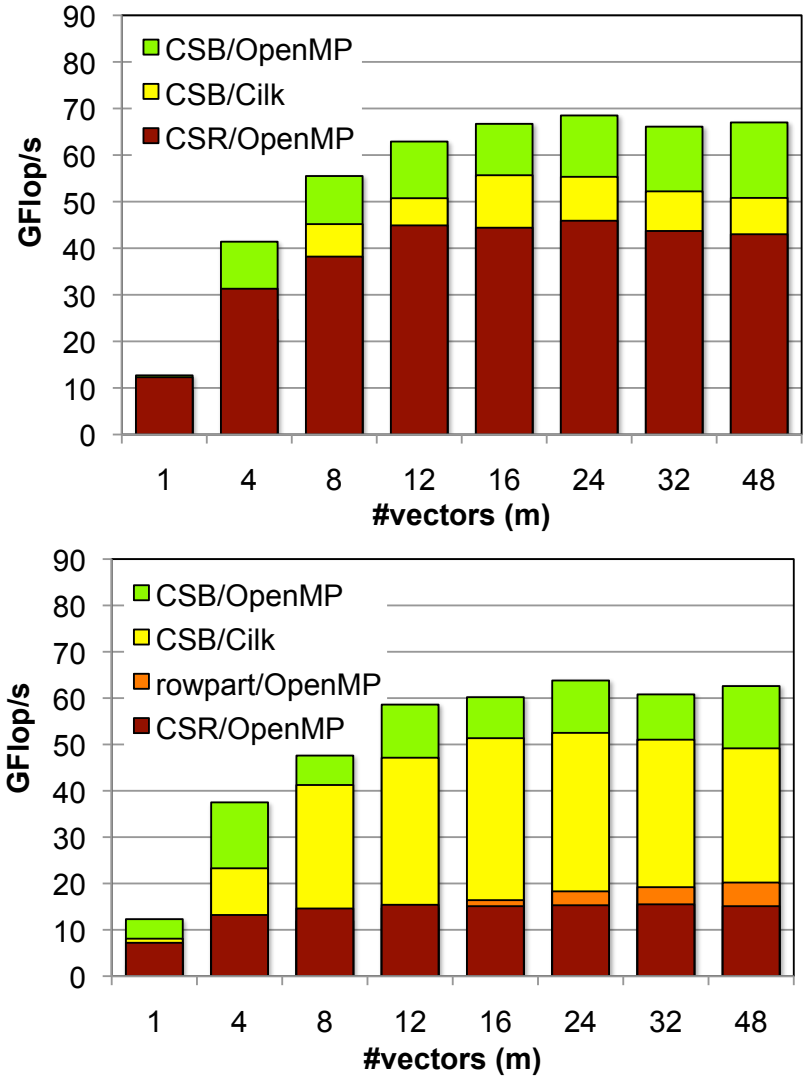


Figure 3.5: Optimization benefits on Edison using the Nm6 matrix for SpMM (top) and SpMM^T (bottom) as a function of m (the number of vectors).

3.8.2 Performance of SpMM and SpMM^T

We first present the SpMM and SpMM^T performance results for our optimized implementations. We report the average performance over five iterations where the number of requisite floating-point operations is $2 \cdot nnz \cdot m$.

3.8.3 Improvement with using CSB

Fig. 3.5 presents SpMM ($Y = AX$) and SpMM^T ($Y = A^T X$) performance for the Nm6 matrix as a function of m (the number of vectors). For $m = 1$, a conventional CSR SpMV implementation does about as well as can be expected. However, as m increases, the benefit of CSB variants' blocking on cache locality is manifested. The CSB/OpenMP version delivers noticeably better performance than the CSB/Cilk implementation. This may be due in part to performance issues associated with Cray's cluster compatibility mode, but most likely due to additional parallelization overheads of the Cilk version that uses temporary vectors to introduce parallelism at the levels of block rows and blocks. This additional level of parallelism is eliminated in CSB/OpenMP by noting that the work associated with each nonzero is significantly increased as m increases, and we leverage the large dimensionality of input vectors for load balancing among threads. Ultimately, we observe that CSB/OpenMP's performance saturates at around 65 GFlop/s for $m > 16$. This represents a roughly 45% increase in performance over CSR, and 20% increase over CSB/Cilk.

CSB truly shines when performing SpMM^T. The ability to efficiently thread the computation coupled with improvements in locality allows CSB/OpenMP to realize a 35% speedup for SpMV over CSR and nearly a 4× improvement in SpMM for $m \geq 16$. The row partitioning scheme has only a minor benefit and only at very large m. Moreover, CSB ensures SpMM and SpMM^T performance are now comparable (67 GFlop/s vs 62 GFlop/s with OpenMP) — a clear requirement as both computations are required for MFDn.

As an important note, we point out that the increase in arithmetic intensity introduced by SpMM allows for more than 5× increase in performance over SpMV. This should be an inspiration to explore algorithms that transform numerical methods from being memory

bandwidth-bound (SpMV) to compute-bound (SpMM).

3.8.4 Tuning for the Optimal Value of β

As discussed previously, we wish to maintain a working set for the X and Y vector blocks as close to the processor as possible in the memory hierarchy. Each $\beta \times \beta$ block demands a working set of size βm in the L2 for X and Y. Thus, as m increases, we are motivated to decrease β. Fig. 3.6 plots performance of the combined SpMM and SpMM^T operation using CSB/OpenMP on the Nm8 matrix as a function of m for varying β. For small m, there is either sufficient cache capacity to maintain locality on the block of vectors, or other performance bottlenecks are pronounced enough to mask any capacity misses. However, for large m (shown up to $m = 96$ for illustrative purposes), we clearly see that progressively smaller β are the superior choice as they ensure a constrained resource (e.g., L3 bandwidth) is not flooded with cache capacity miss traffic. Still, note in Fig. 3.6 that no matter what β value is used, the maximum performance obtained for $m > 48$ is lower than the peak of 45 Gflops/s achieved for lower values of m. This suggests that for large values of m, it may be better to perform SpMM and SpMM^T as batches of tasks with narrow vector blocks. In the following sections, we always use the best value of β for a given m.

3.8.5 Combined SpMM/SpMM^T performance

Our ultimate goal is to include the LOBPCG algorithm as an alternative eigensolver in MFDn. As discussed earlier, the computation of both SpMM and SpMM^T is needed for this purpose. We are therefore interested in the performance benefit for the larger (and presumably more challenging) MFDn matrices. Fig. 3.7 presents the combined performance of SpMM and SpMM^T as a function of m for our three test matrices. Clearly, the CSB variants

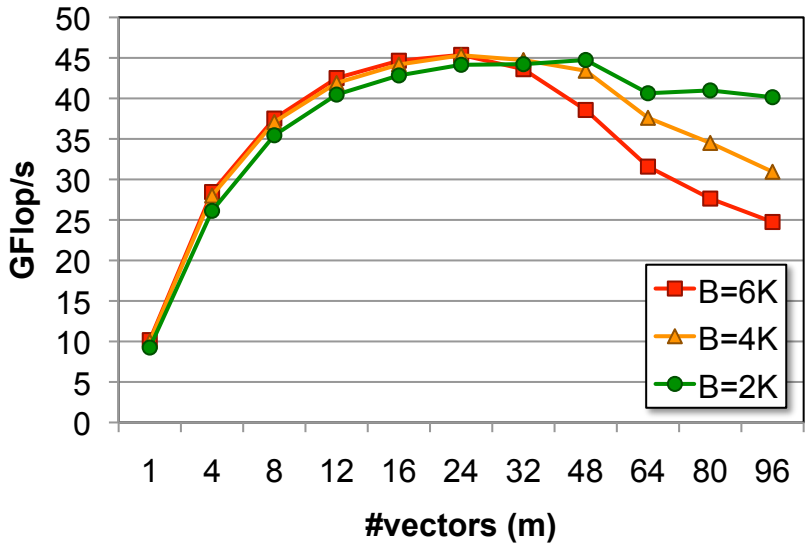


Figure 3.6: Performance benefit on the combined SpMM and SpMM^T operation from tuning the value of β for the Nm8 matrix.

deliver extremely good performance for the combined operation with the CSB/OpenMP delivering the best performance. We observe that as one increases the number of vectors m, performance increases to a point at which it saturates. A naive understanding of locality would suggest that regardless of matrix size, the ultimate SpMM performance should be the same. However, as one moves to the larger and sparser matrices, performance saturates at lower values. Understanding these effects and providing possible remedies requires introspection using our performance model.

3.8.6 Performance analysis

Given the complex memory hierarchies of varying capacities and bandwidths in highly parallel processors, the ultimate bottlenecks to performance can be extremely non-intuitive and require performance modeling. In Fig. 3.7, we provide three Roofline performance bounds based on DRAM, L3, and L2 data movements and bandwidth limits as described in Sect. 3.6. In all cases, we use the empirically determined optimal value of β for each m as a parameter

in our performance model. The L2 and L3 bounds take the place of the traditional in-core (peak flop/s) performance bounds. Bounding data movement for small m (where compulsory data movement dominates) is trivial and thus accurate. However, as m increases, capacity and conflict misses start dominating. In this space, quantifying the volume of data movement in a deep cache hierarchy with an unknown replacement policy and unknown reuse pattern is non-trivial. As Fig. 3.4 clearly demonstrates, the matrices in question are not random (worst case), but exhibit a structure. We note that these Roofline curves for large m are not strict performance bounds but rather guidelines.

Clearly, for small m performance is highly-correlated with DRAM bandwidth. As we proceed to larger m, we see an inversion for the sparser matrices where L3 bandwidth can surpass DRAM bandwidth as the preeminent bottleneck. We observe that for the denser Nm6 matrix, performance is close to our optimistic L2 bound. Nevertheless, the model suggests that the L3 bandwidth is the true bottleneck while DRAM bandwidth does not constrain performance for $m \geq 8$. Conversely, the sparser Nm8's performance is well correlated with the DRAM bandwidth bound for $m \leq 16$ at which point the L3 and DRAM bottlenecks reach parity.

Ultimately, our Roofline model tracks the performance trends well and highlights potential bottlenecks — DRAM, L3, and L2 bandwidths and capacities — as one transitions to larger m or larger and sparser matrices.

3.8.7 Performance of tall-skinny matrix operations

In Fig. 3.8, we analyze the performance of our custom inner product ($V^T W$) and linear combination (VC) operations proposed as alternatives to the BLAS library calls for tall-skinny matrices. As mentioned in Sect. 3.7, our custom implementations still rely on the

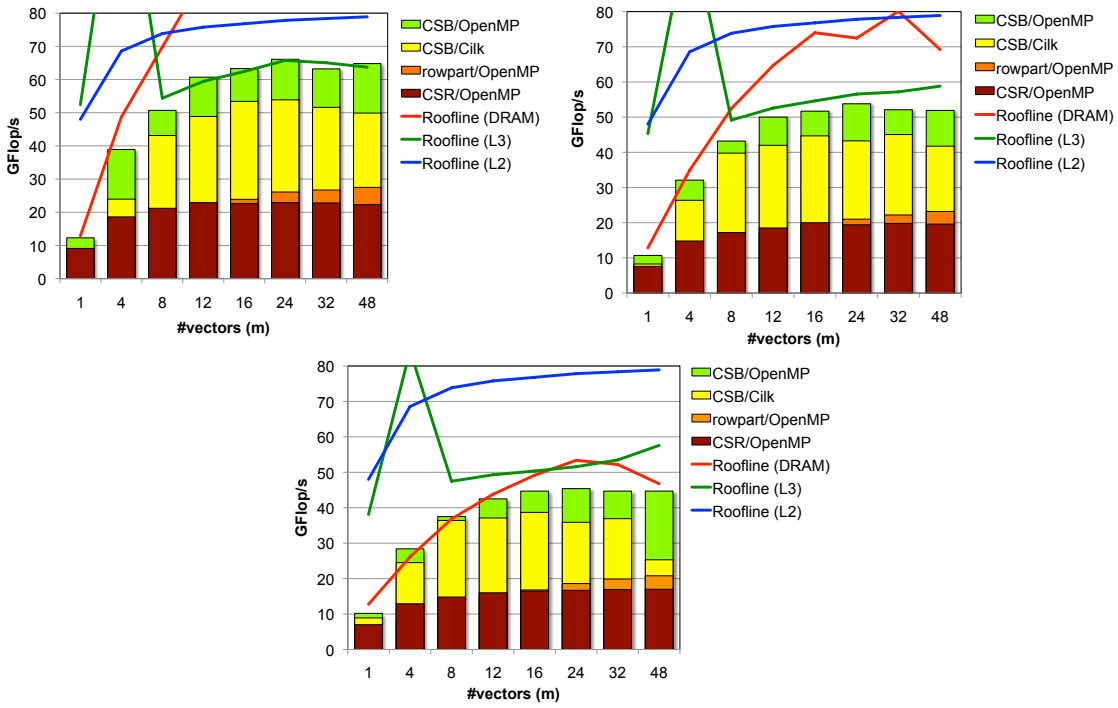


Figure 3.7: SpMM and SpMM^T combined performance results on Edison using the Nm6, Nm7 and Nm8 matrices (from top to bottom) as a function of m (the number of vectors). We identify 3 Rooflines (one per level of memory) as per our extended roofline model for CSB.

library `dgemm` calls to perform multiplications between small matrix blocks. Hence, we report the performance of two different versions, *Custom/MKL* based on MKL `dgemm`, and *Custom/LibSci* based on the LibSci `dgemm`.

As shown in Fig. 3.8, we obtain significant speedups over MKL and LibSci in computing V^TW. Both of our custom V^TW kernels exhibit a similar performance profile and outperform their library counterparts significantly. The speedups we obtain range from about 1.7× (for larger values of m) up to 10× (for $m \approx 16$). As small m values are common in an application like MFDn, this represents a significant performance improvement for LOBPCG over using the library `dgemm`. However, for the VC kernel, we do not observe speedups from our custom implementations (Fig. 3.8) – they closely track the performance of their library counterparts. In fact, for larger values of m, the *Custom/MKL* implementation is outperformed slightly by MKL. While l was fixed at 1 M for these tests, we observed similar

results in other cases (l=10 K, 100 K, and 500 K).

A key observation based on Fig. 3.8 is that the overall performance of both kernels are significantly higher for larger m values. Since LOBPCG algorithm contains operations with multiple blocks of vectors, i.e., Ψ_i, R_i, P_i, one potential optimization is to combine these three blocks of vectors into a single $l \times 3m$ matrix. In this case, the 18 inner product operations with $l \times m$ matrices would be turned into 2 separate inner products with $l \times 3m$ matrices. As the *Custom bundled* curve shows in Fig. 3.8, the additional performance gains are significant for $V^T W$, achieving up to 15× speedup compared to the library counterparts. However, the linear combination operation VC does not benefit from bundling as much as the inner product does. For VC, the improvements we observe are limited to a factor of 1.5 for values of m ranging from 12 to 48. The main reason behind the limited performance gains in this case is that the tall-skinny matrix products can be converted into `dgemm`'s of dimensions $l \times 3m$ and $3m \times m$, as opposed to being extended to $3m$ in all shorter dimensions as in the case of $V^T W$.

3.8.8 Performance summary

We now demonstrate the benefits of an architecture-aware eigensolver implementation in actual CI problems. MFDn's existing solver uses the Lanczos algorithm with full orthogonalization (Lanczos/FO) for numerical stability and good convergence. We implemented the LOBPCG algorithm in MFDn using the optimized SpMM and tall-skinny matrix kernels described above. The distributed memory implementations for both solvers are similar and use the 2D partitioning scheme described in Sect. 3.2 and in more detail in [104].

Beyond optimizing the kernels, there are a number of key issues that need to be considered to leverage the full benefits of LOBPCG. These include the choice of initial eigenvector

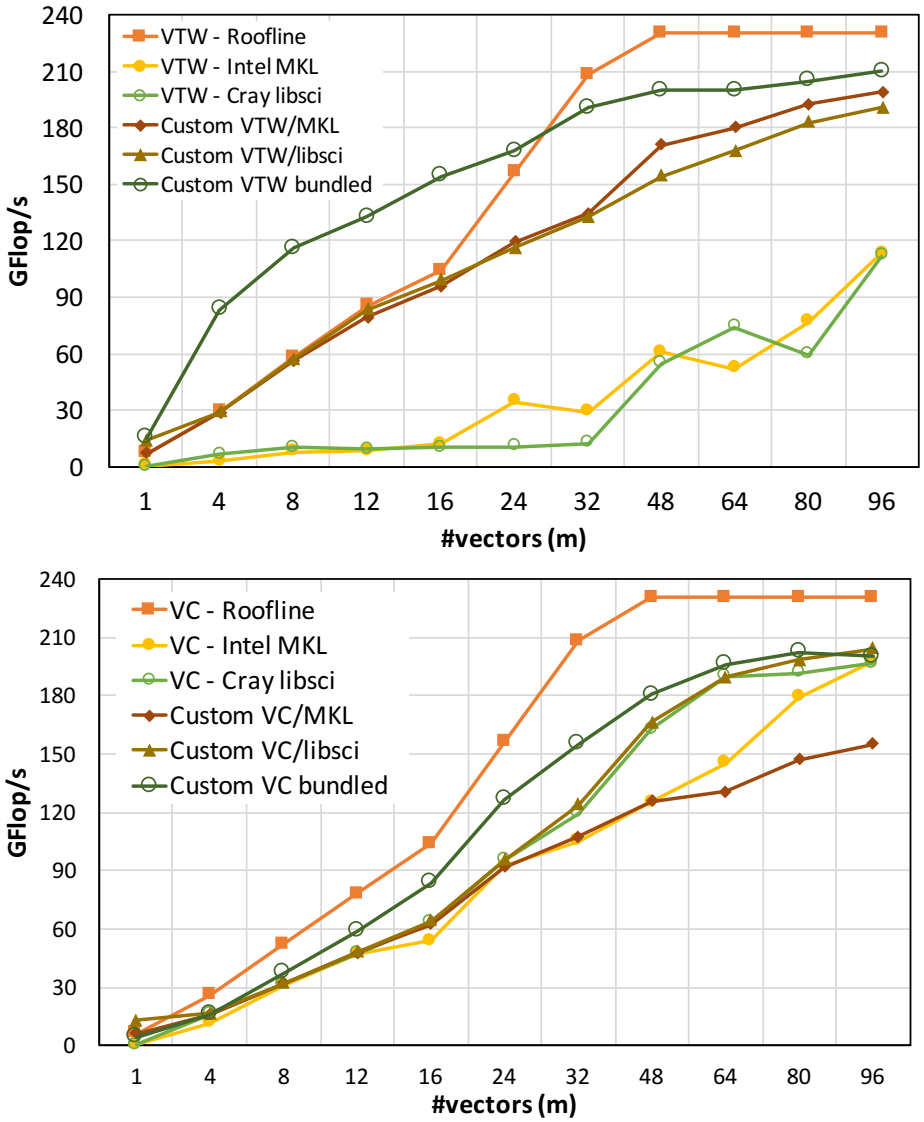


Figure 3.8: Performance in GFlop/s for inner product V^TW (top), and linear combination VC (bottom) operations, using Intel MKL and Cray LibSci *libraries, as well as our custom implementations on Edison. Tall-skinny matrix sizes are $l \times m$, where $l = 1\,M$.*

guesses, the design of a preconditioner to accelerate convergence, and suitable data structures for combining the optimized SpMM and tall-skinny kernels. Full details and analyses of initial guesses and preconditioning techniques used are beyond the scope of this paper and will be discussed in a subsequent publication—we describe these techniques briefly below:

Initial Guesses: CI models with truncations smaller than the target $N_{\max}$ result in significantly smaller problem sizes. Roughly speaking, reducing $N_{\max}$ by 2 (subsequent truncations must be evenly separated) gives an order of magnitude reduction in matrix dimensions. We observe that eigenvectors computed with smaller $N_{\max}$ values provide good approximations to the eigenvectors in the target model space, so we solve the eigenvalue problem of the smaller $N_{\max}$ first and use these results as initial guesses to our target problem. This idea can be applied recursively for additional performance benefits.

Preconditioning: Preconditioners transform a given problem into a form that is more favorable for iterative solvers, often by reducing the condition number of the input matrix. To be effective in large-scale computations, a preconditioner must be easily parallelizable and computationally inexpensive to compute and apply, while still providing a favorable transformation. We build such a preconditioner in MFDn by computing crude approximations to the inverses of the diagonal blocks of the Hamiltonian (easy to parallelize). The diagonal blocks in CI typically contain very large nonzeros (important for a quality transformation).

Bundling Blocks of Vectors: While bundling all three blocks of vectors into a single, but thicker tall-skinny matrix is favorable for LOBPCG (see Sect. 3.7), this would harm the performance of the SpMM and SpMM^T operations as the locality between consecutive rows of input and output vectors would be lost. A work around to this problem is to copy the input vectors Ψ_i from the vector bundle into a separate block of vectors at the end of

LOBPCG (in preparation for SpMM), and then copy $H\Psi_i$ back into the vector bundle after SpMM^T (in preparation for LOBPCG). Our experiments show that overheads associated with such copies are small compared to the gains obtained from bundled V^TW and VC operations, hence we opt to bundle the blocks of vectors in LOBPCG into a single one in our implementation.

Alignment/Padding: If vector rows are not aligned with the 32-bit word boundaries, the overall performance of the solver is significantly reduced due to the presence of unpacked vector instructions. Hence we set the initial basis dimension m for LOBPCG such that m is a multiple of 4 to ensure vectorization at least with SSE instructions. When m is a multiple of 8, AVX instructions are automatically used, but forcing m to always be a multiple of 8 introduces computational overheads not compensated by AVX vectorization. As the converged eigenvectors need to be locked, the basis size would slowly shrink during LOBPCG iterations. To maintain good performance throughout, we shrink the basis only when the number of active vectors decreases to a smaller multiple of 4, and replace the converged vectors with 0 vectors in the meantime.

Our testcases to compare the eigensolvers are the full ^{10}B problems with $N_{\max}$ truncations of 6, 7 and 8, seeking 8 eigenpairs in all cases. Table 3.3 gives more details for the testcases and the distributed memory runs. We executed the MPI/OpenMP parallel solvers using 6 threads/rank (despite having 12 cores/socket), because Intel's MPI library currently supports only serialized MPI communications by multiple threads (i.e., `MPI_THREAD_SERIALIZED` mode). Using 12 threads per rank resulted in not being able to fully saturate the network injection bandwidth, and therefore increased communication overheads in both cases.

Problem	^{10}B, Nm6	^{10}B, Nm7	^{10}B, Nm8
Dimension (in millions)	12.06	44.87	144.06
Nonzeros (in billions)	5.48	26.77	108.53
nev (*m* for LOBPCG)	8 (12)	8 (12)	8 (12)
residual tolerance	1e-6	1e-6	1e-6
MPI ranks (w/6 omp threads)	15	66	231
Lanczos Iterations	240	320	240
SpMVs	240	320	240
Inner Products	28,800	51,200	28,800
LOBPCG Iterations	28	48	38
SpMVs	324	548	428
Inner Products	72,656	121,296	93,936

Table 3.3: Statistics for the full MFDn matrices used in distributed memory parallel Lanczos/FO and LOBPCG executions.

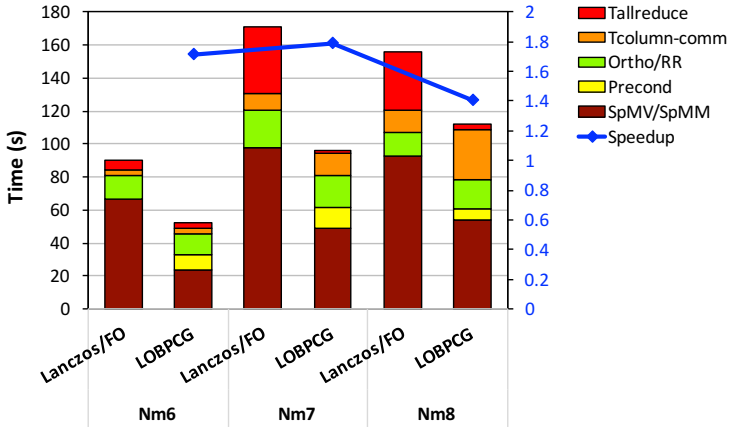


Figure 3.9: Comparison and detailed breakdown of the time-to-solution using the new LOBPCG implementation vs. the existing Lanczos/FO solver. Nm6, Nm7 and Nm8 testcases executed on 15, 66, and 231 MPI ranks (6 OpenMP threads per rank), respectively, on Edison.

In Fig. 3.9, we break down the overall timing into the following parts: sparse matrix computations (*SpMV/SpMM*), application of the preconditioner (*Precond*), orthonormalization for Lanczos and Rayleigh–Ritz procedure for LOBPCG (*Ortho/RR*), communications among column groups of the 2D processor grid ($T_{col-comm}$)—row communications are fully overlapped with (*SpMV/SpMM*) [104], and finally `MPI_Allreduce` calls needed for reductions during *Ortho/RR* step ($T_{allreduce}$). The solve times for smaller $N_{\max}$ truncations to obtain the initial LOBPCG guesses are included in the execution times in Fig. 3.9. We

observe that our LOBPCG implementation consistently outperforms the existing Lanczos solver by a factor of 1.7×, 1.8×, and 1.4× for the Nm6, Nm7 and Nm8 cases, respectively. Although LOBPCG requires more SpMVs overall for convergence (see Table 3.3), the main reason for the improved time-to-solution is the high performance SpMM and SpMM^T kernels we presented. Since two threads in a socket (one per MPI rank) are used to overlap communications with SpMM computations, the peak SpMM flop rate during the eigensolver iterations is slightly lower than those in Fig. 3.7. For LOBPCG computations though, we observe that V^TW and VC kernels execute at rates in line with those in Fig. 3.8. We note that without the preconditioner and initial guess techniques we adopted, LOBPCG's slow convergence rate leads to similar or worse solution times compared to Lanczos/FO, wiping out the gains from replacing SpMVs with SpMMs. In this regard, inexpensive solves with smaller $N_{\max}$ truncations and low cost preconditioners are crucial for the performance benefits obtained with LOBPCG.

In Table 3.3, we also show the total number of inner products required by both solvers. The LOBPCG algorithm relies heavily on vector operations as discussed in Sect. 3.7 and evidenced by the total number of inner products reported here. So despite the use of Lanczos with full orthogonalization, LOBPCG requires more inner products overall. In addition, a smaller but still significant number of linear combination operations, as well as solutions of small eigenvalue problems are required for LOBPCG. Cumulatively, these factors lead to a computationally expensive *Ortho/RR* part for the LOBPCG solver. So despite using highly optimized BLAS 3 based kernels in LOBPCG, we observe that the time spent in *Ortho/RR* part is comparable for Lanczos/FO and LOBPCG. Finally, in the larger runs, i.e., Nm7 and Nm8, we observe that Lanczos/FO solver incurs significant $T_{allreduce}$ times, possibly due to slight load imbalances exacerbated by frequent synchronizations. On the

other hand, $T_{allreduce}$ times are much lower for LOBPCG (an expected consequence of the fewer iteration counts), but LOBPCG's $T_{col-comm}$ are significantly higher due to the larger volume of communication required in this part.

3.9 Evaluation on Xeon Phi Knights Corner (KNC)

Our performance evaluation on Xeon Phi is limited to the isolated SpMM and tall-skinny matrix kernels due to MFDn's large memory requirements and the limited device memory available on the KNC (which was used in the native mode as a proxy for the upcoming Knights Landing architecture). We used the same testcases as before, and experimented with 30, 60, 120, 180, and 240 threads to determine the ideal thread count. We obtained the best performance with 120 threads for both SpMM and tall-skinny matrix kernels and report those results.

Comparing the average SpMM Gflop rates on KNC (given in Fig. 3.10) with the Ivy Bridge (IB) Xeon, we see that the peak performance on KNC is much lower (as much as 3× for Nm6, $m = 12$, for instance) than that on IB for all cases. This is likely due to the significantly smaller cache space available per KNC thread. In any case, we can clearly see that the CSB/OpenMP implementation delivers significantly better performance than traditional CSR and Rowpart implementations, as it did on IB. On KNC, we also observe the pattern of increased performance with increasing values of m values. But unlike IB where the use of SSE vs AVX vectorization did not make a significant difference (see the similar GFlop rates for $m = 12$ and $m = 16$ in Fig. 3.7), on KNC utilizing packed AVX-512 vectorization is crucial as indicated by the sharp performance drop in going from $m = 16$ to $m = 24$. We utilize the same extended Roofline model as before, but KNC does not offer

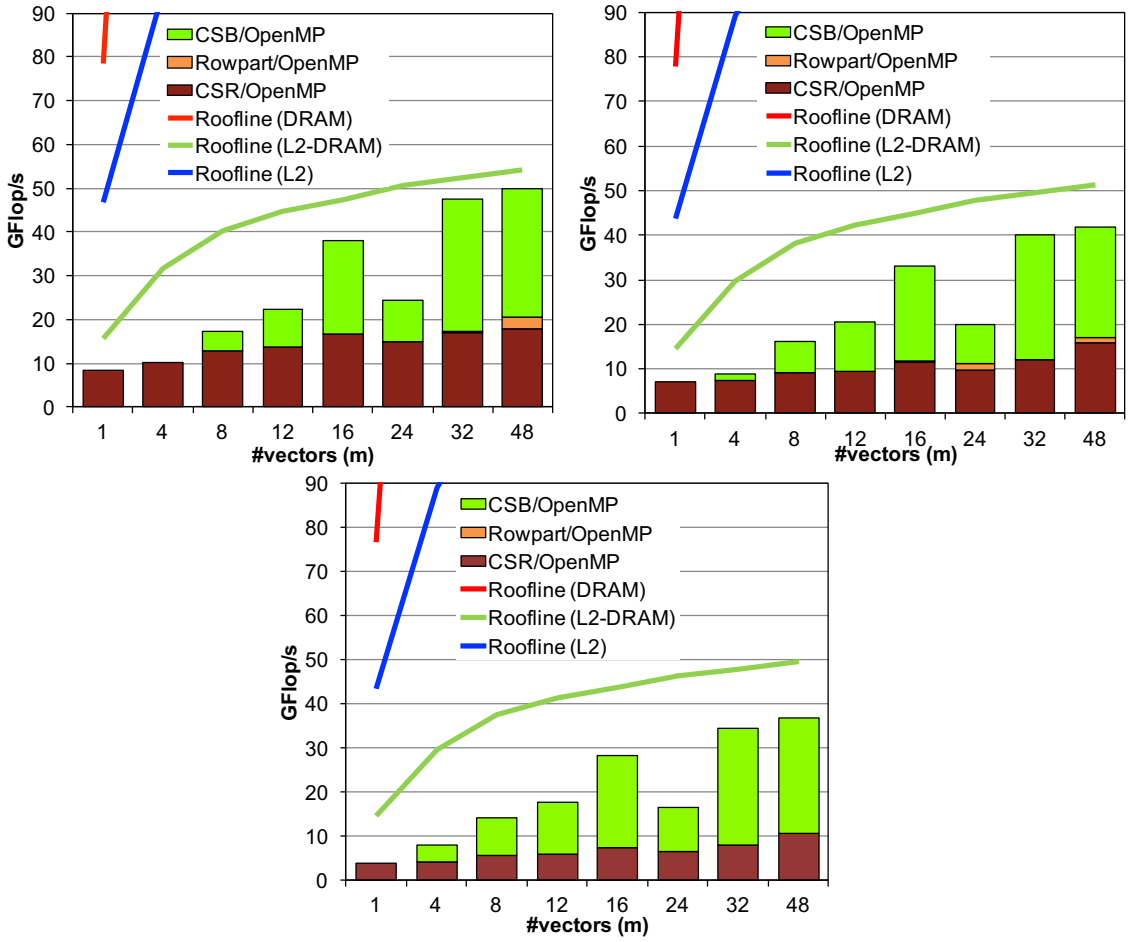


Figure 3.10: SpMM and SpMM^T combined performance results on Babbage using the Nm6, Nm7 and Nm8 matrices (from top to bottom) as a function of m (the number of vectors).

a shared L3 cache, as such the L3 Roofline of Fig. 3.7 is replaced with the device memory to L2 bandwidth limit, i.e., Roofline L2-DRAM. With KNC's high bandwidth memory and limited cache space, original DRAM Roofline gives a very loose bound, and so does the plain L2 Roofline. But L2-DRAM Roofline provides a tight envelope.

For the tall-skinny matrix operations, we only present results using the MKL library as LibSci was not available on KNC. In Fig. 3.11, we observe that our custom V^TW kernel significantly outperforms MKL's `dgemm` (up to 25×). *Custom/Bundled* implementation gives important performance gains for $8 < m < 48$. For the VC kernel however, our custom implementations provide only slight improvements over MKL. Overall, the performance achieved

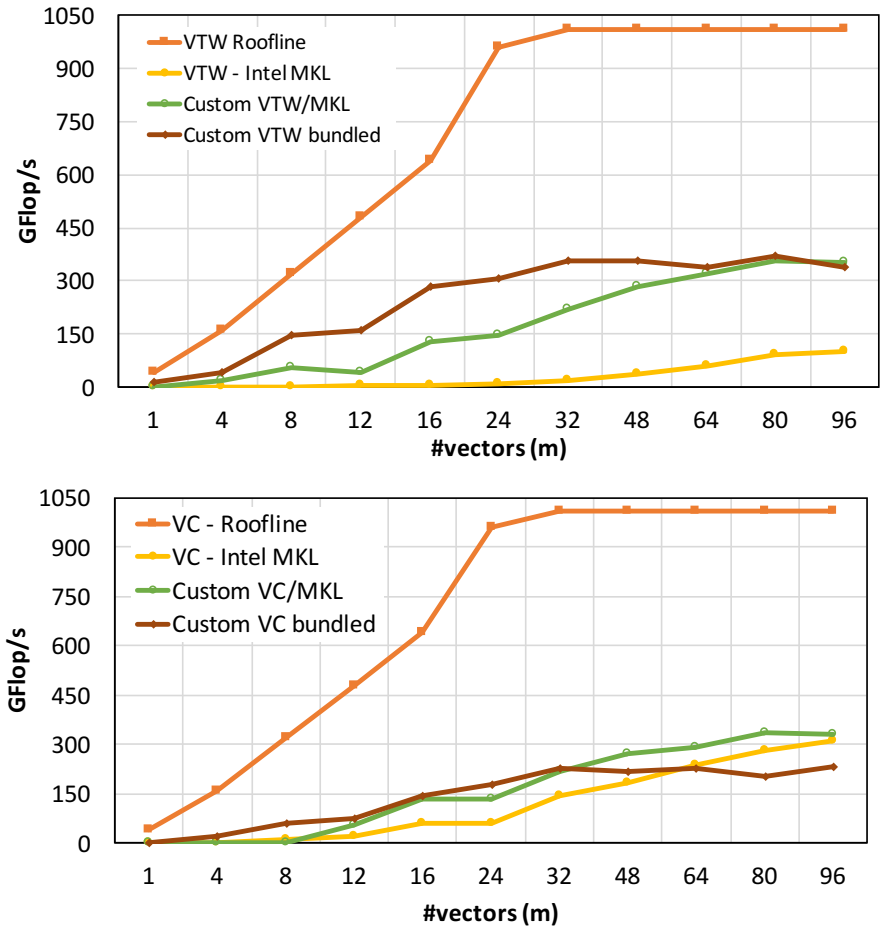


Figure 3.11: Performance of $V^T W$ (top) and VC (bottom) kernels, using the MKL library, as well as our custom implementations on an Intel Xeon Phi processor. Local vector blocks are $l \times m$, where $l = 1\,M$.

for both kernels is very low compared to the peak performance predicted by the Roofline model.

The poor performance observed for SpMM and tall-skinny matrix operations on KNC suggests that further optimizations are necessary to achieve good eigensolver performance on future systems.

3.9.1 Conclusions of this work

In this work we observed block eigensolvers are favorable alternatives to SpMV-based iterative eigensolvers for modern architectures with a widening gap between processor and memory system performance. In this study, we focused on the sparse matrix multiple vectors (SpMM, SpMM^T) and tall-skinny matrix operations (V^TW, VC) that constitute the key kernels of a block eigensolver. Using many-body nuclear Hamiltonian test matrices extracted from MFDn, we demonstrated that the use of compressed sparse blocks (CSB) format in conjunction with manual unrolling for vectorization and tuning can improve SpMM and SpMM^T performance by up to 1.5× and 4×, respectively, on modern multi-core processors. As block eigensolvers are sufficiently compute-intensive, the DRAM bandwidth may be relegated to a secondary bottleneck. We presented an extended Roofline model that captures the effects of L2 and L3 bandwidth limits in addition to the original DRAM bandwidth limit. This extended model highlighted how the performance bottleneck transitions from DRAM to the L3 bandwidth for large m values or sparser matrices.

Contrary to the common wisdom, we observe that simply calling `dgemm` in optimized math libraries does not suffice to attain high flop rates for tall-skinny matrix operations in block eigensolvers. Through custom thread-parallel implementations for inner product and linear combination operations and bundling separate vector blocks into a single large block, we have obtained 1.2× to 15× speedup (depending on m and the type of operation) over MKL and LibSci libraries.

Taking the ideas from this work we wanted to extend this approach to an entire solver rather than focusing only on a single kernel. Hence we started working on a task parallel framework which we discuss next in detail.

Chapter 4

ON NODE TASK PARALLEL OPTIMIZATION

This work has been published in IEEE HiPC 2019[107]. This work is a collaborative effort from Md Afibuzzaman and Fazlay Rabbi. Both contributed equally in this work.

The most fundamental operation in sparse linear algebra is arguably the multiplication of a sparse matrix with a vector (SpMV), as it forms the main computational kernel for several applications, such as, the solution of partial differential equations (PDE) [7] and the Schrödinger Equation [8] in scientific computing, spectral clustering [9] and dimensionality reduction [108] in machine learning, the Page Rank algorithm [11] in graph analytics, and many others. The Roofline model by Williams et al. [12] suggests that the performance of SpMV kernel is ultimately bounded by the memory bandwidth. Consequently, performance optimizations to increase cache utilization and reduce data access latencies for SpMV has drawn significant interest, e.g., [13, 14, 15, 16].

A closely related kernel is the multiplication of a sparse matrix with multiple vectors (SpMM) which constitutes the main operation in block solvers, e.g., the block Krylov subspace methods [109, 7] and block Jacobi-Davidson method. SpMM has much higher arithmetic intensity than SpMV and can efficiently leverage wide vector execution units. As a result, SpMM-based solvers has recently drawn significant interest in scientific comput-

ing [22, 23, 24, 26, 25, 110, 111, 112]. SpMM also finds applications naturally in machine learning where several features (or eigenvectors) of sparse matrices are needed [108, 9]. Although SpMM has a significantly higher arithmetic intensity than SpMV, the extended Roofline model that we recently proposed suggests that cache bandwidth, rather than the memory bandwidth, can still be an important performance limiting factor for SpMM [22].

LAPACK [85] is a linear algebra library for solving systems of simultaneous linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. LAPACK routines mostly exploit Basic Linear Algebra Subprograms (BLAS) to solve these problems. PLASMA aims to overcome the shortcomings of the LAPACK library in efficiently solving the problems in dense linear algebra on multicore processors [113, 114]. PLASMA can solve dense general systems of linear equations, symmetric positive definite systems of linear equations and linear least squares problems, using LU, Cholesky, QR and LQ factorizations and supports both single precision and double precision arithmetic. However, PLASMA does not support general sparse matrices and does not solve sparse eigenvalue or singular value problems. PLASMA supports only shared-memory machines.

MAGMA is a dense linear algebra library (like LAPACK) for heterogeneous systems, i.e., systems with GPUs [115, 116, 117], to fully exploit the computational power that each of the heterogeneous components would offer. MAGMA provides very similar functionality like LAPACK and makes it easier for the user to port their code from LAPACK to MAGMA. MAGMA supports both CPU and GPU interfaces. The users do not have to know details of GPU programming to use MAGMA.

Barrera *et. al.* [118] use computational dependencies and dynamic graph partitioning method to minimize NUMA effect on shared memory architectures. StarPU [119] is a runtime

system that facilitates the execution of parallel tasks on heterogeneous computing platforms, and incorporates multiple scheduling policies. However, the application developer has to create the computational tasks by themselves in order to use StarPU.

While the concept of task parallelism based on data flow dependencies is not new, exploration of the benefits of this idea in the context of sparse solvers constitutes a novel aspect of this work. Additionally, to the best of our knowledge, related work on task parallelism has not explored its impact on cache utilization compared to the BSP model as we do in this work.

4.1 DeepSparse Overview

Figure 4.1 illustrates the architectural overview of DeepSparse. As shown, DeepSparse consists of two major components: i) *Primitive Conversion Unit* (PCU) which provides a front-end to domain scientists to express their application at a high-level; and ii) *Task Executor* which creates the actual tasks based on the abstract task graph generated by PCU and hands them over to the OpenMP runtime for execution.

As sparse matrix related computations represent the most expensive calculations in many large-scale scientific computing, we define tasks in our framework based on the decomposition of the input sparse matrices. For most sparse matrix operations, both 1D (block row) and 2D (sparse block) partitioning are suitable options. A 2D partitioning is ideal for exposing high degrees of parallelism and reducing data movement across memory layers [120], as such 2D partitioning is the default scheme in DeepSparse. For a 2D decomposition, DeepSparse defines tasks based on the Compressed Sparse Block (CSB) [14] representation of the sparse matrix, which is analogous to tiles that are commonly used in task parallel implementation

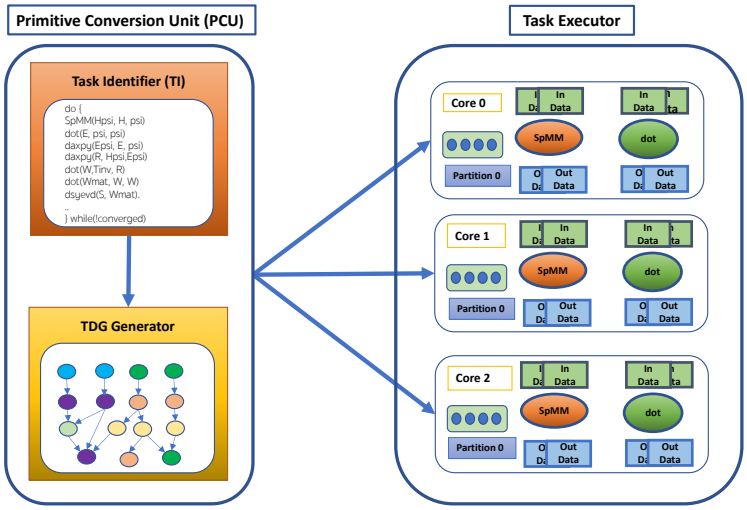


Figure 4.1: Schematic overview of DeepSparse.

of dense linear algebra libraries. However, CSB utilizes much larger block dimensions (on the order of thousands) due to sparsity [22, 27, 14]. Consequently, DeepSparse starts out by decomposing the sparse matrix (or matrices) for a given code into CSB blocks (which eventually corresponds to the tasks during execution with each kernel producing a large number of tasks). Note that the decomposition of a sparse matrix dictates partitioning of the input and output vectors (or vector blocks) in the computation as well, effectively inducing decomposition of all data structures used in the solver code.

DeepSparse creates and maintains fine-grained dependency information across different kernels of a given solver code based on the result of the above decomposition scheme. As such, instead of simply converting each kernel into its own task graph representation and concatenating them, DeepSparse generates a global task graph, allowing for more optimal data access and task scheduling decisions based on global information. Since the global task graph depends on the specific algorithm and input sparse matrix, DeepSparse will explicitly generate the corresponding task dependency graph. While this incurs some computational and memory overheads, such overheads are negligible. The main reason for computational

overheads to be negligible is that sparse solvers are typically iterative, and the same task dependency graph is used for several iterations. The reason why memory overheads is negligible is that each vertex in the task graph corresponds to a large set of data in the original problem. After this brief overview, we explain the technical details in DeepSparse.

4.1.1 Primitive Conversion Unit (PCU)

PCU is composed of two parts: i) Task Identifier, and ii) Local Task Dependency Graph (TDG) Generator.

4.1.1.1 Task Identifier (TI)

The application programming interface (API) for DeepSparse is a combination of the recently proposed GraphBLAS interface [84] (for sparse matrix related operations) and BLAS/LAPACK [85, 86] (for vector and occasional dense matrix related computations). This allows application developer to express their algorithms at a high-level without having to worry about architectural details (e.g., memory hierarchy) or parallelization considerations (e.g., determining the individual tasks and their scheduling). Task identifier parses a code expressed using the DeepSparse API to identify the specific BLAS/LAPACK and GraphBLAS calls, as well as the input/output of each call. It then passes this information to the local task dependency graph generator.

TI builds two major data structures:

ParserMap: `ParserMap` is an unordered map that holds the parsed data information in the form of (`Key`, `Value`) pairs. As TI starts reading and processing the DeepSparse code, it builds a `ParserMap` from the function calls. To uniquely identify each call

in the code, *Key* class is made up of three components: `opCode` which is specific to each type of operation used in the code, `id` which keeps track of the order of the same function call in the code (e.g., if there are two matrix addition operations, then the first call will have id = 1 and the second one will have id = 2), and `timestamp` which stores the line number of the call in the code and is used to detect the input dependencies of this call to the ones upstream. For each key, the corresponding `Value` object stores the input and output variable information. It also stores the dimensions of the matrices involved in the function call.

Keyword & idTracker: `Keyword` is a vector of strings that holds the unique function names (*i.e.*, `cblas_dgemm, dsygv, mkl_dcrmm`, etc.) that have been found in the given code, and the `idTracker` keeps track of the number of times that function (Keyword) has been called so far. Keyword and idTracker vectors are synchronized with each other. When TI finds a function call, it searches for the function name in the *Keyword* vector. If found, the corresponding idTracker index is incremented. Otherwise, the *Keyword* vector is expanded with a corresponding initial *idTracker* value of 1.

4.1.1.2 Task Dependency Graph Generator (TDGG)

The output of Task Identifier (TI) is a dependency graph at a very coarse-level, *i.e.*, at the function call level. For an efficient parallel execution and tight control over data movement, tasks must be generated at a much finer granularity. This is accomplished by the Task Dependency Graph (TDGG), which goes over the input/output data information generated by TI for each function call and starts decomposing these data structures. As noted above, the decomposition into finer granularity tasks starts with the first function call involving

the sparse matrix (or matrices) in the solver code which is typically an SpMV, SpMM or SpGEMM operation. After tasks for this function call are identified by examining the non-zero pattern of the sparse matrix, tasks for prior and subsequent function calls are generated accordingly. As part of task dependency graph generation procedure, TDGG also generates the dependencies between individual fine-granularity tasks by examining the function call dependencies determined by TI. Note that the dependencies generated by TDGG may (and often do) span function boundaries and this is an important property of DeepSparse that separates it from a bulk synchronous parallel (BSP) program which effectively imposes barriers at the end of each function call.

The resulting task dependency graph generated by TDGG is essentially a directed acyclic graph (DAG) representing the data flow in the solver code where vertices denote computational tasks, incoming edges represent the input data and outgoing edges represent the output data for each task. TDGG also labels the vertices in the task dependency graph with the estimated computational cost of each task, and the directed edges with the name and size of the corresponding data, respectively. During execution, such information can be used for load balancing among threads and/or ensuring that active tasks fit in the available cache space. In this initial version of DeepSparse though, such information is not yet used because we rely on OpenMP's default task execution algorithms, as explained next.

4.1.2 Task Executor

To represent a vertex in the task graph, TDGG uses an instance of the TaskInfo structure [listing 4.1] which provides all the necessary information for the Task executor to properly spawn the corresponding OpenMP task. The task executor receives an array of TaskInfo structures [listing 4.1] from the PCU that represents the full computational dependency

Listing 4.1: TaskInfo Structure

```
struct TaskInfo
{
  int opCode; //type of operation
  int numParamsCount;
  int *numParamsList; //tile id, dimensions etc.
  int strParamsCount;
  char **strParamsList; //i.e. buffer name
  int taskID; //analogous to id of Key Class
}
```

graph, picks each node from this array one by one and extracts the corresponding task information. DeepSparse implements OpenMP task based functions for all computational kernels (represented by `opCode`) it supports. Based on the `opCode`, partition id of the input/output data structures and other required parameters (given by `numParamsList` and `strParamsList`) found in the TaskInfo structure at hand, Task Executor calls the necessary computational function found in the DeepSparse library, effectively spawning an OpenMP task.

In DeepSparse, the master thread spawns all OpenMP tasks one after the other, and relies on OpenMP's default task scheduling algorithms for execution of these tasks. OpenMP's Runtine Environment then determines which tasks are ready to be executed based on the provided task dependency information. When ready, those tasks are executed by any available member of the current thread pool (including the master thread). Note that OpenMP supports task parallel exeuction with named dependencies, and better yet these dependencies can be specified as variables. This feature is fundamental for DeepSparse to be able to generate TDGs based on different problem sizes and matrix sparsity patterns. This is exemplified in Algorithm 1, where SpMM tasks for the compressed sparse block at row i and j is simply invoked by providing the $X[i, j]$ sparse matrix block along with $Y[j]$ input vector

Algorithm 1: SpMM Kernel

Input: X[i,j] ($\beta \times \beta$, Sparse CSB block), Y[j] ($\beta \times b$)
Output: Z[i] (Dense vector block, $\beta \times b$)

```
1 #pragma omp task depend(in: X[i,j], Y[j], Z[i]) depend(out: Z[i])
2 {
3 foreach val ∈ X[i,j].nnz do
4     r = X[i,j].row_loc[val]
5     c = X[i,j].col_loc[val]
6     for k = 0 to b do
7         Z[r × b + k] = Z[r × b + k] + val × Y[c × b + k]
8 }
```

block and $Z[i]$ output vector block in the *depend* clause.

An important issue in a task parallel program is the data race conditions involving the output data that is being generated. Fortunately, the task-parallel execution specifications of OpenMP requires only one thread to be active among threads writing into the same output data location. While this ensures a race-condition free exeuction, it might hinder performance due to a lack of parallelism. Therefore, for data flowing into tasks with a high incoming degree, DeepSparse allocates temporary output data buffers based on the number of threads and the available memory space. Note that this also requires the creation of an additional task to reduce the data in temporary buffer space before it is fed into the originally intended target task.

4.1.3 Illustrative Example

We provide an example to demonstrate the operation of DeepSparse using the simple code snippet provided in Listing 4.2. As TI parses the sample solver code, it discovers that the first `cblas_dgemm` in the solver corresponds to a linear combination operation (see Fig. 4.2), the second line is a sparse matrix vector block multiplication (SpMM, see Fig. 4.3) and

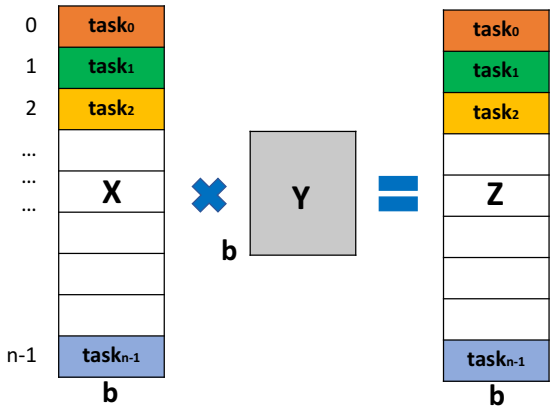


Figure 4.2: Overview of input output matrices partitioning of task-based matrix multiplication kernel.

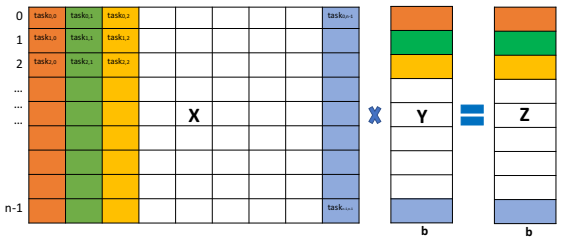


Figure 4.3: Overview of matrices partitioning of task-based SpMM kernel.

the second `cblas_dgemm` at the end is an inner product of two vector blocks (see Fig. 4.4). These function calls, their parameters as well as dependencies are captured in the ParserMap, Keyword, and idTracker data structures as shown in Table 4.1.

Listing 4.2: An example pseudocode

```
cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, k, 1.0, A, k, B, n
    , 0, C, n);
SpMM(X, C, D, m, n);
cblas_dgemm(CblasRowMajor, CblasTrans, CblasNoTrans, n, n, m, 1.0, D, n, C, n,
    0, E, n);
```

Task Dependency Graph (TDG) generator receives the necessary information from TI and determines the tasks corresponding to partitionings of operand data structures of each

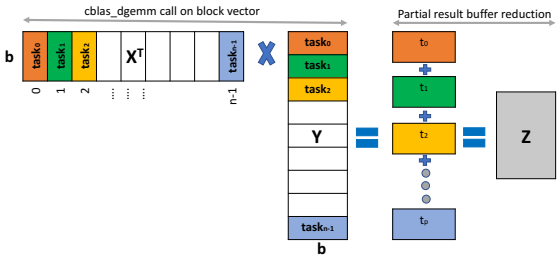


Figure 4.4: Overview of matrices partitioning of task-based inner product kernel.

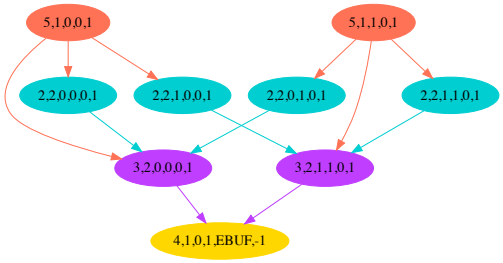


Figure 4.5: Task graph for the psudocode in listing 4.2.

operation, as well as their origins (whether the necessary data are coming from another task or from a variable). TDGG then builds the DAG of each computational kernel and appends it to the global DAG with proper edge connectivity (*i.e.*, dependencies). While generating the DAG, the TDGG also encodes the value of the TaskInfo structure instance that represent each of the vertices into the vertex name. The vertex naming convention is *<opCode, numParamsCount, numParamsList, strParamsCount, strParamsList, taskID>*. Figure 4.5 shows the task dependency graph of the solver code in Listing 4.2 (assuming m = 100, k = 8, n = 8, $CSBtile/blocksize$ = 50, so each input matrix is partitioned into 2 chunks).

The task executor receives an array of TaskInfo structures that contains the node information as shown in Figure 4.5. The task executor goes over each of the tasks in the array of TaskInfo structure. At first, it reads the nodes (*<5,1,0,0,1>*, *<5,1,1,0,1>*) of the first operation and spawns two matrix multiplication (xY) tasks with proper input output matrices. The task executor then reads all the task information for all SpMM tasks {*<2,2,0,0,0,1>*,

Data Structure	Content
ParserMap	<{XY, 1, 1},{<A, B>, <C>, <m, n, k>}>
	<{SpMM, 1, 2},{<X, C>, <D>, <m, m, n>}>
	<{XTY, 1, 3},{<D, C>, <E>, <m, n, n>}>
keyword	<XY, SpMM, XTY>
idTracker	<1, 1, 1>

Table 4.1: Major data structures after parsing third line.

<2,2,0,1,0,1>, *<2,2,1,0,0,1>*, *<2,2,1,1,0,1>*} and spawns four SpMM tasks with proper input/output matrix blocks. Finally, the task executor reads *<3,2,0,0,0,1>*, *<3,2,1,1,0,1>* and *<4,1,0,1,EBUF,-1>* and spawns two inner product ($X^T Y$) tasks and one partial output buffer reduction task for the inner product operation.

4.1.4 Limitations of the Task Executor

Despite the advantages of an asynchronous task-parallel execution, the Task Executor has the following limitations:

Synchronization at the end of an iteration: Most computations involving sparse matrices are based on iterative techniques. As such, the TDG generated for a single iteration can be reused over several steps (until the algorithm reaches convergence). However, it is necessary to introduce a `#pragma omp taskwait` at the end of each solver iteration and force all tasks of the current iteration to be completed to ensure computational consistency among different iterations of the solver. For relatively simple solvers, the `taskwait` clause adds some overhead to the total execution time due to threads idling at `taskwait`s.

Limited number of temporary buffers: While OpenMP allows the use of program

variables in the dependency clauses, it does not allow dynamically changing the variable lists of the *depend* clauses. As such, the number of buffer lists in the partial output reduction tasks need to be fixed to overcome this issue. Depending on the available memory, there are at most $nbuf$ number of partial output buffers for a reduction operation. If $nbuf$ is less than the total number of threads, then there might be frequent read after write (RAW) contentions on partial output buffers. This could be have been potentially avoided, if the list of variables in the *depend* clause could have been dynamically changed.

4.2 Benchmark Applications

We demonstrate the performance of the DeepSparse framework on two important eigensolvers widely used in large-scale scientific computing applications: Lanczos eigensolver [121] and Locally Optimal Block Preconditioned Conjugate Gradient algorithm (LOBPCG) [122].

4.2.1 Lanczos

Lanczos algorithm finds eigenvalues of a symmetric matrix by building a matrix $Q_k = [q_1, \ldots, q_k]$ of orthogonal Lanczos vectors [123]. The eigenvalues of the sparse matrix A is then approximated by the Ritz values. As shown in Algorithm 2, it is a relatively simple algorithm consisting of an Sparse Matrix Vector Multiplication (SpMV) along with some vector inner products for orthonormalization.

Algorithm 2: Lanczos Algorithm in Exact Arithmetic

$q\_1 = b/ \parallel b \parallel_2, \beta_0 = 0, q_0 = 0$ **for** *j = 1 to k* **do**
  $z = Aq_j$
  $\alpha_j = {q_j}^T z$
  $z = z - \alpha_j q_j - \beta_{j-1} q_{j-1}$
  $\beta_j = \parallel z \parallel_z$
  $if \beta_j = 0, quit$
  $q_{j+1} = z/\beta_j$
  Compute eigenvalues, eigenvectors, and error bounds of T_k

4.2.2 LOBPCG

LOBPCG is a commonly used block eigensolver based on the SpMM kernel [122], see Figure 3 for a pseudocode. Compared to Lanczos, LOBPCG comprises high arithmetic intensity operations (SpMM and Level-3 BLAS). In terms of memory, while the $\widehat{H}$ matrix takes up considerable space, when a large number of eigenpairs are needed (e.g. dimensionality reduction, spectral clustering or quantum many-body problems), memory needed for block vector Ψ can be comparable to or even greater than that of $\widehat{H}$. In addition, other block vectors (residual R, preconditioned residual W, previous direction P), block vectors from the previous iteration and the preconditioning matrix T must be stored, and accessed at each iteration. Figure 4.6 shows a sample task graph for LOBPCG generated by TDGG using a very small matrix. Clearly, orchestrating the data movement in a deep memory hierarchy to obtain an efficient LOBPCG implementation is non-trivial.

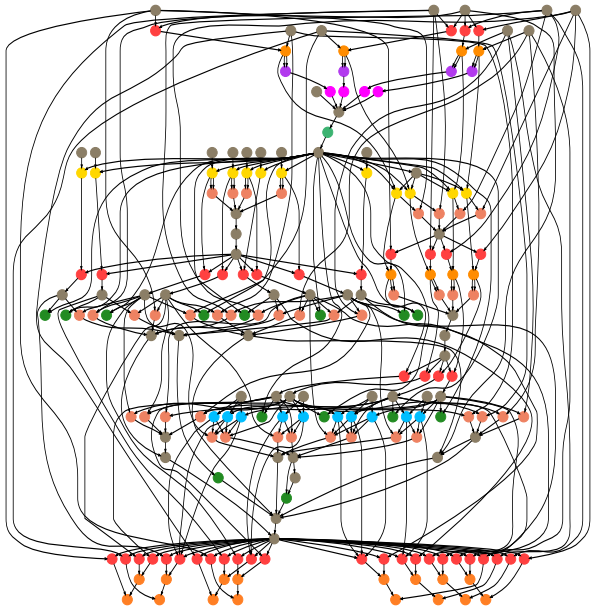


Figure 4.6: A sample task graph for the LOBPCG algorithm using a small sparse matrix.

4.3 Performance Evaluation

4.3.1 Experimental setup

We conducted all our experiments on Cori Phase I, a Cray XC40 supercomputer at NERSC, mainly using the GNU compiler. Each Cori Phase I node has two sockets with a 16-core Intel Xeon Processor E5-2698 v3 Haswell CPUs. Each core has a 64 KB private L1 cache (32 KB instruction and 32 KB data cache) and a 256 KB private L2 cache. Each CPU has a 40 MB shared L3 cache (LLC). We use thread affinity to bind threads to cores and use a maximum of 16 threads to avoid NUMA issues. We test DeepSparse using five matrices with different size, sparsity patterns and domains (see Table 4.2). The first 4 matrices are from The SuitSparse Matrix Collection and the Nm7 matrix is from nuclear no-core shell model code MFDn.

We compare the performance of DeepSparse with two other library implementations: i) **libcsr** is implementation of the benchmark solvers using thread-parallel Intel MKL Library

Algorithm 3: LOBPCG Algorithm (for simplicity, without a preconditioner) used to solve $\hat{H}\Psi = E\Psi$

Input: $\hat{H}$, matrix of dimensions $N \times N$
Input: Ψ_0, a block of vectors of dimensions of $N \times m$
Output: Ψ and E such that $\|\hat{H}\Psi - \Psi E\|_F$ is small, and $\Psi^T\Psi = I_m$

Orthonormalize the columns of Ψ_0
$P_0 \leftarrow 0$
for *i = 0, 1, . . . , until convergence* **do**
  $E_i = \Psi_i^T \hat{H} \Psi_i$
  $R_i \leftarrow \hat{H}\Psi_i - \Psi_i E_i$
  Apply the Rayleigh–Ritz procedure on $\text{span}\{\Psi_i, R_i, P_i\}$
  $\Psi_{i+1} \leftarrow \underset{S \in \text{span}\{\Psi_i.R_i,P_i\},\ S^TS=I_m}{\text{argmin}} \text{trace}(S^T \hat{H} S)$
  $P_{i+1} \leftarrow \Psi_{i+1} - \Psi_i$
  Check convergence

calls (including SpMV/SpMM) with CSR storage of the sparse matrix, ii) **libcsb** is an implementation again using Intel MKL calls, but with the matrix being stored in the CSB format. Performance data for LOBPCG is averaged over 10 iterations, while the number of iterations is set to 50 for Lanczos runs. Our performance comparison criteria are L1, L2, LLC misses and execution times for both solvers. All cache miss data was obtained using the Intel VTune software.

Performance of the DeepSparse and libcsb implementations depends on the CSB block sizes used. Choosing a small block size creates a large number of small tasks. While this is preferable on a highly parallel architecture, the large number of tasks may lead to significant task execution overheads, in terms of both cache misses and execution times. Increasing the block size reduces such overheads, but this may then lead to increased thread idle times and load imbalances. Therefore, the CSB block size is a parameter to be optimized based on the specific problem. Different block sizes we experimented with have been 1K, 2K, 4K, 8K and 16K.

Table 4.2: Matrices used in our evaluation.

Matrix	Rows	Columns	Nonzeros
inline1	503,712	503,712	36,816,170
dielFilterV3real	1,102,824	1,102,824	89,306,020
HV15R	2,017,169	2,017,169	283,073,458
Queen4147	4,147,110	4,147,110	316,548,962
Nm7	4,985,422	4,985,422	647,663,919

4.3.2 LOBPCG evaluation

In Fig. 4.7, we show the number of cache misses at all three levels (L1, L2 and L3) and execution time comparison between all three versions of the LOBPCG algorithm compiled using the GNU compiler. LOBPCG is a complex algorithm with a number of different kernel types; its task graph results in millions of tasks for a single iteration. As shown in Fig. 4.7, except for the Nm7 matrix, libcsb and libcsr versions achieve similar number of cache misses; for Nm7, libcsb has important cache miss reductions over the libcsr version. On the other hand, DeepSparse achieves 2.5× - 10.7× fewer L1 misses, 6.5× - 16.2× fewer L2 misses and 2× - 7× fewer L3 cache misses compared to the libcsr version. As the last row of Fig. 4.7 shows, even with the implicit task graph creation and execution overheads of DeepSparse, the significant reduction in cache misses leads to 1.2× - 3.9× speedup over the execution times of libcsr. Given the highly complex DAG of LOBPCG and abundant data re-use opportunities available, we attribute these improvements to the pipelined execution of tasks which belong to different computational kernels (see Fig. 4.8) but use the same data structures. We note that the Task Executor in DeepSparse solely relies on the default scheduling algorithm used in the OpenMP runtime environment. By making use of the availability of the entire global task graph and labeling information on vertices/edges, it might be possible to improve the performance of DeepSparse even further.

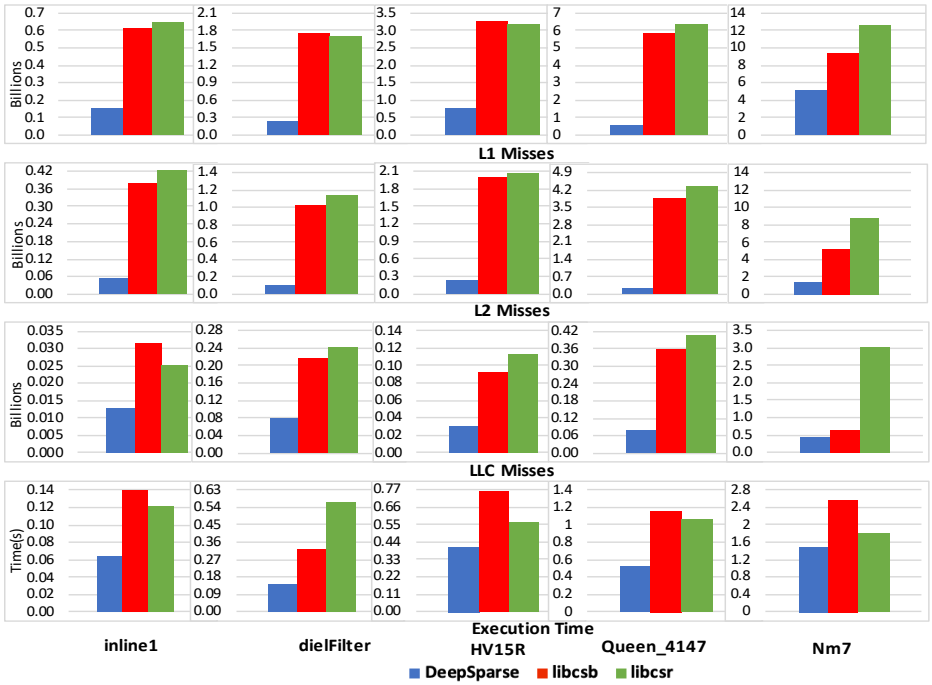


Figure 4.7: Comparison of L1, L2, LLC misses and execution times between Deepsparse, libcsb and libcsr for the LOBPCG solver.

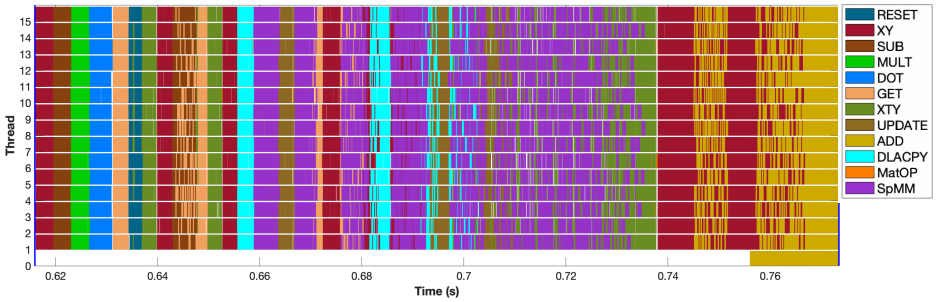


Figure 4.8: LOBPCG single iteration execution flow graph of dielFilterV3real.

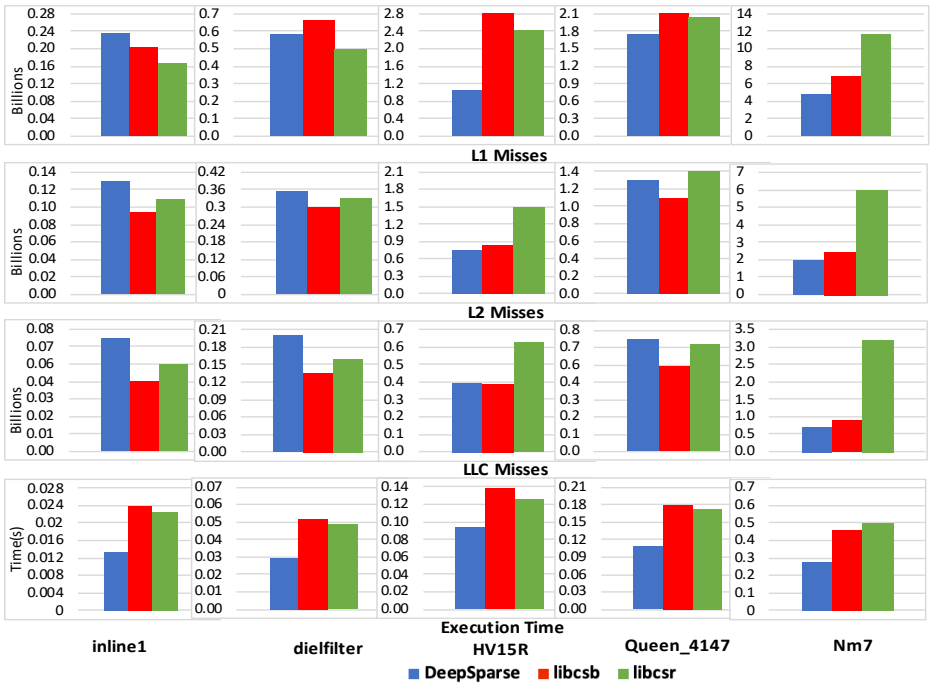


Figure 4.9: Comparison of L1, L2, LLC misses and execution times between Deepsparse, libcsb and libcsr for the Lanczos solver.

4.3.3 Lanczos evaluation

In Fig. 4.9, cache misses and execution time comparisons for different Lanczos versions are shown. Lanczos algorithm is much simpler than LOBPCG, it has much fewer types and numbers of tasks than LOBPCG (basically, one SpMV and one inner product kernel at each iteration). As such, there are not many opportunities for data re-use. In fact, we observe that DeepSparse sometimes leads to increases in cache misses for smaller matrices. However, for the Nm7 and HV15R matrices, which are the largest matrices among our benchmark set, we observe an improvement in cache misses, achieving up to 2.4× fewer L1 cache misses, 3.1× fewer L2 misses and 4.5× fewer L3 misses than libcsr. But most importantly, DeepSparse achieves up to 1.8× improvement in terms of execution time. We attribute the execution time improvement observed across the board to the increased degrees of parallelism exposed by the global task graph of DeepSparse, which is in fact highly critical for smaller matrices.

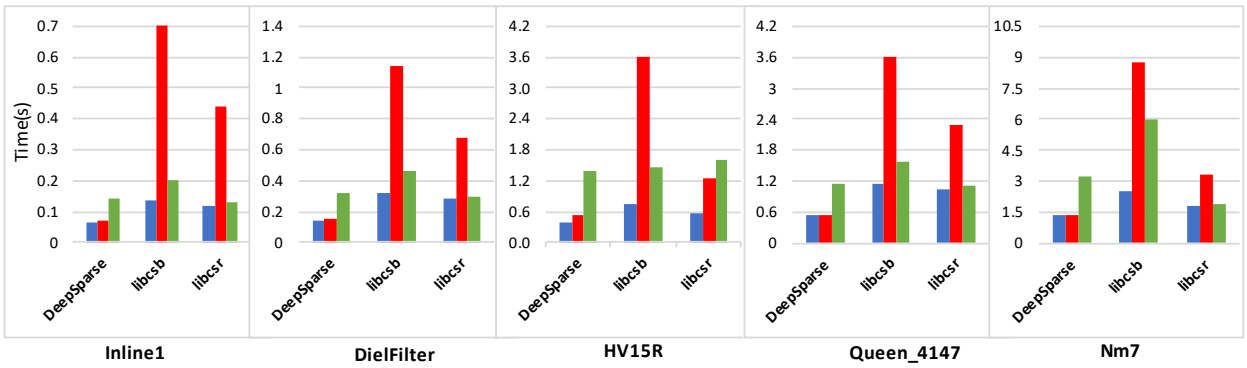


Figure 4.10: Comparison of execution time for different compilers between Deepsparse, libcsb and libcsr for Lanczos Algorithm. (Blue/Left: GNU, Red/Middle: Intel, Green/Right: Clang compiler.)

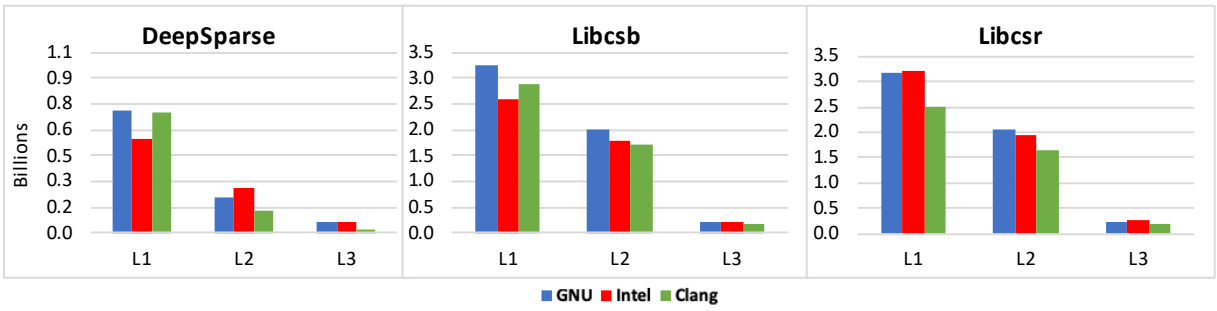


Figure 4.11: Cache Miss comparison between compilers for HV15R

4.3.4 Compiler comparison

For all of our experiments, we use OpenMP as our backend. To explore the impact of different task scheduling approaches in different OpenMP implementations, we have experimented with three compilers: Intel, GNU and Clang/LLVM compilers. In Figure 4.10, we show the comparison in execution time among different compilers for the three implementations. We see that the execution time for the Clang/LLVM compiler is significantly higher compared to GNU and Intel compilers for all matrices. However, cache misses stay pretty much the same when one moves to a different compiler. We show the cache miss comparison between the three compilers in Figure 4.11 for one matrix, HV15R. All other matrices follow a similar cache miss pattern like HV15R. Here, we can clearly see that regardless of the compiler, DeepSparse achieves fewer cache misses over libcsb and libcsr implementations. We can see that Clang/LLVM shows fewer cache misses for DeepSparse as well, but it eventually has a poor running time. We believe that this is because Clang/LLVM is not able to schedule tasks as efficiently as GNU and Intel. Compared to Intel compiler, GNU compiler sometimes shows more L1 and L2 misses. But the execution time is higher in Intel. This may be due to the scheduling strategy and and the implementation of task scheduling points in the compilers. Overall, GNU does best with running times among the three compilers, and Intel compilers do not do well with the library based solver implementations.

4.3.5 Conclusions of this work

This work introduces a novel task-parallel sparse solver framework which targets all computational steps in sparse solvers. We show that our approach achieves significantly fewer cache misses across different cache layers and also improves the execution time over the li-

brary versions. Future works will be in the direction of further reducing the cache misses and execution time over the current versions by experimenting with more advanced partitioning and scheduling algorithms compared to the default schemes in OpenMP. In this work we used the scheduler provided by OpenMP but we also wanted to partition the tasks using a memory heuristics to generate a custom scheduler. We discuss this in detail in the next chapter.

Chapter 5

SCHEDULING TASKS WITH A GRAPH PARTITIONER

In the DeepSparse framework, we saw that OpenMP does a great job with memory utilization over all memory levels. But it is still beyond our control. OpenMP is generating the DAG itself and resolving themselves. Whenever the data dependencies of a task is resolved and it is not dependent on any other task for its execution , it can be immediately pulled and executed. But this might not be optimal scenario if we think from memory usage perspective. A task which does not have any relation with the tasks that are active at the moment can be immediately executed once a thread gets free regardless of its memory input and outputs. Hence there is a possibility of a task which would improve the memory usage with the input already being in the lower level of the memory and having cache hits reduces. The probability of cache misses increases with this kind of scheduling.

This motivated us to use a novel graph partition based scheduler that use the global data flow graphs generated by the PCU to minimize data movements in a deep memory hierarchy. Graph partitioners have been extensively studied but existing approaches do not meet our needs as they typically handle DAGs by converting them to undirected graphs. However, the directed nature of the task graph must be respected in our case.

Our sparse solver DAGs contain a fair number of vertices with high fan-in/fan-out" de-

grees due to operations such as SpMVs, SpMMs, inner products and vector reductions.

In this work we take the graph generated by the task generator, create appropriate data structures for the graph with accordance to the partitioner code and then partition the graph using the following algorithm steps.

5.1 Coarsening

Experimental DAGs are usually very large in size. Often they have millions or tens of millions of nodes. The partitioning algorithms they use have different complexities, the worst being $\mathcal{O}(n^2)$ in the kernighan algorithm where n is the number of vertices. This makes the partitioning phase really slow and impractical if we want to use this partitioner as a supporting tool. This is why the original graph is coarsened to a smaller coarsened graph until a minimum size is reached.

In the original code we received, they coarsen the graph using a matching algorithm. At each coarsening step, they compute how many vertices can be matched. They consider all the edges one by one and put them in the matching if they respect the acyclicity property. Let us see how the matching is actually done in their implementation with a simple graph which has almost all kinds of cases that might arise in our lobpcg DAG in graph 5.1.

Here we can see a graph with 15 vertices. There is vertex 5 which has multiple incoming edges and multiple outgoing edges. The algorithm first creates a topological order of the nodes visited in dfs traversal. Then it goes through each of the vertices and checks the constraints for acyclicity. Node 15 will be visited first according to the topological order. A sorted neighbor algorithm is run to find the outgoing edge with the least cost. In this case, node 4 is chosen rather than 3. So, node 15 and 4 are matched. Node 3 is then visited

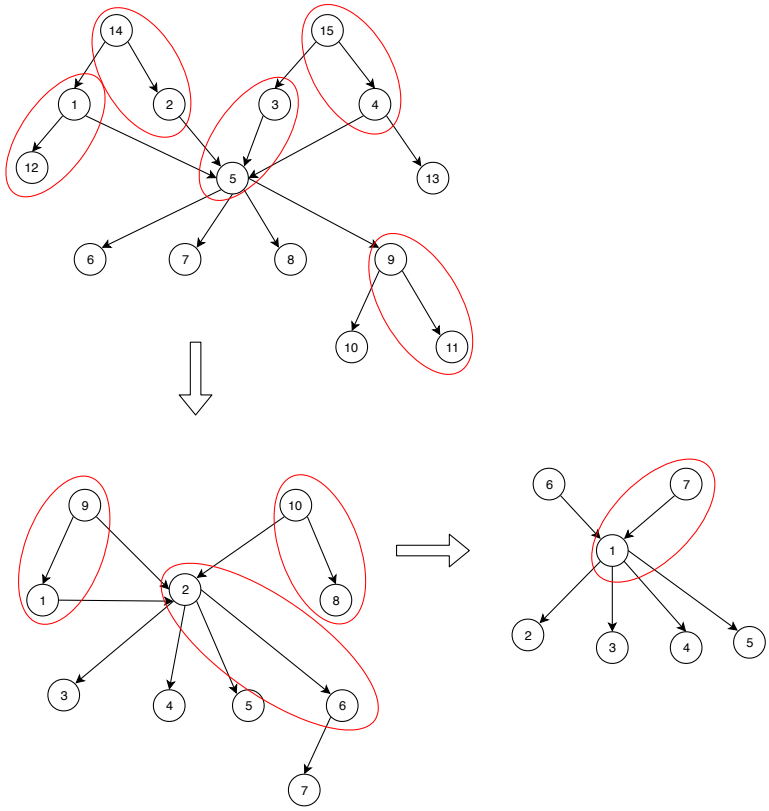


Figure 5.1: Matching example

according to the topological order. Likewise, node 5 is its only outgoing node and thus node 3 and node 5 are selected for the next matching. Likewise, nodes 14,2 and 1,12 are matched. In the end nodes 9 and 11 are matched.

After these nodes are matched, a new graph is generated with the matched nodes. If (u, v) is matched then u becomes the leader of v. That is how 14 is the leader of 2, 3 is the leader of 5. Then they are numbered consecutively to form the new coarse graph. We can see that the acyclicity is preserved in the new graph. Node numbering has been changed. Once 2,5 is matched in this graph, no other nodes 3,4,5,7 will be looked for. Eventually this algorithm will run into cases like the last graph where each time only one matching will be selected.

The problem that we faced with this matching algorithm was that the graph was eventually going to the last graph stage where each time we were getting a very few matching, making the entire matching process really slow. There is a threshold value for stopping the matching process but after initial coarsening, the whole graph becomes a big graph with small parts just like this last graph here.

To get rid of this issue, we added an extra pre-processing step before the actual matching. As we saw in the previous case that a lot of these kinds of nodes which has a lot og incoming edges and outgoing edges, most likely the topological levels of the nodes that are the sources of these high degree nodes will be same. Also, most likely the children of these high degree nodes will also have the same topological level. That is why during the pre-processing phase, we do an additional coarsening of the graph. We coarse every 3 nodes in the same topological level into one node. As they already maintain the topological order in a DAG, this coarsening does not lead to a cycle. The motivation for this change is to minimize the incoming and outgoing nodes for the high degree nodes. Let us see with a simple lobpcg DAG and its pre-

processed version. In Figure 5.2 we can see an actual lobpcg DAG generated by the DAG generator code. This DAG is for a matrix which is divided into 3*3 CSB blocks. Straightway we can see the high degree nodes in this graphs. Even in such a simple graph, there are multiple high degree graphs which might lead us to the issue we previously discussed. In graph 5.3 we see the preprocessed graph. Here we can see that every 3 nodes in the same topological level are coarsened into one single node. Although the edges are shown as they were in the previous graph, my opinion is we can merge them onto a single edge adding their costs to form a new cost.

5.2 Initial Partitioning

After the coarse graph is generated, it is fed to the initial partitioning function. They have a bunch of partitioning algorithms from where we can chose one. Although we initially had kernighan algorithm as the partitioning algorithm, we figured out that it takes a lot of time for kernighan algorithm because it traverses through all the nodes.

In there paper they discuss about a greedy approach where they define *free vertices* and *eligible vertices* where *free vertices* are those vertices which are not put in a part yet. *Eligible vertices* are those vertices whose predecessors are all not free. Gain is computed for eligible vertices to see in which part they can be moved. The gain is computed according to their algorithm in the paper and the objective function is edge cost. That means the minimization of the edge cut.In the end all the vertices are divided into two parts.

There was another approach named BFSGrowing which was actually a DFS traversal of nodes and from that traversal each time one vertex is being selected and it is added to the current filling part. The weight of that vertex is then adjusted accordingly. But this

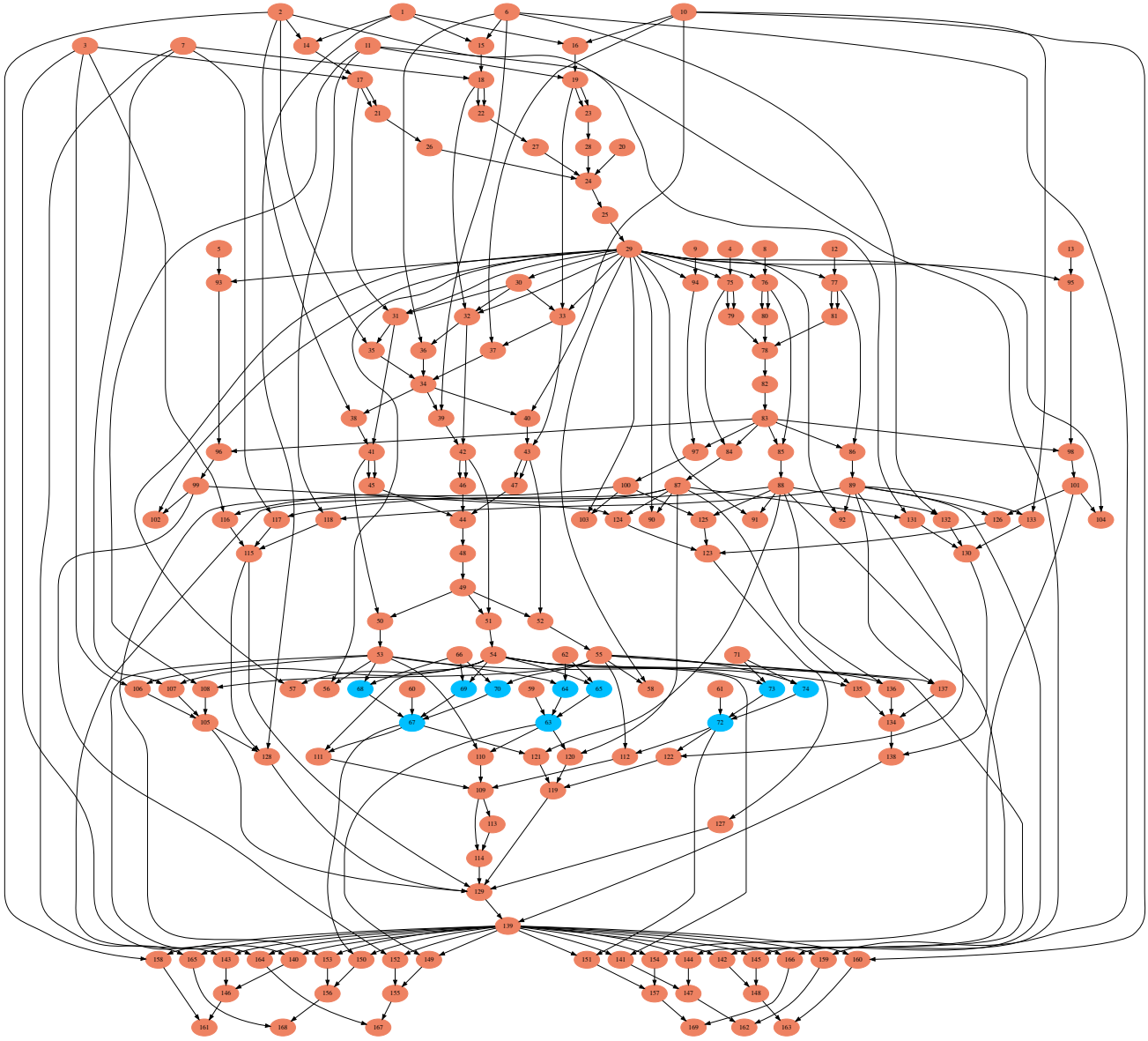


*Figure 5.2: Simple Lobpcg DAG with 3*3 blocks*

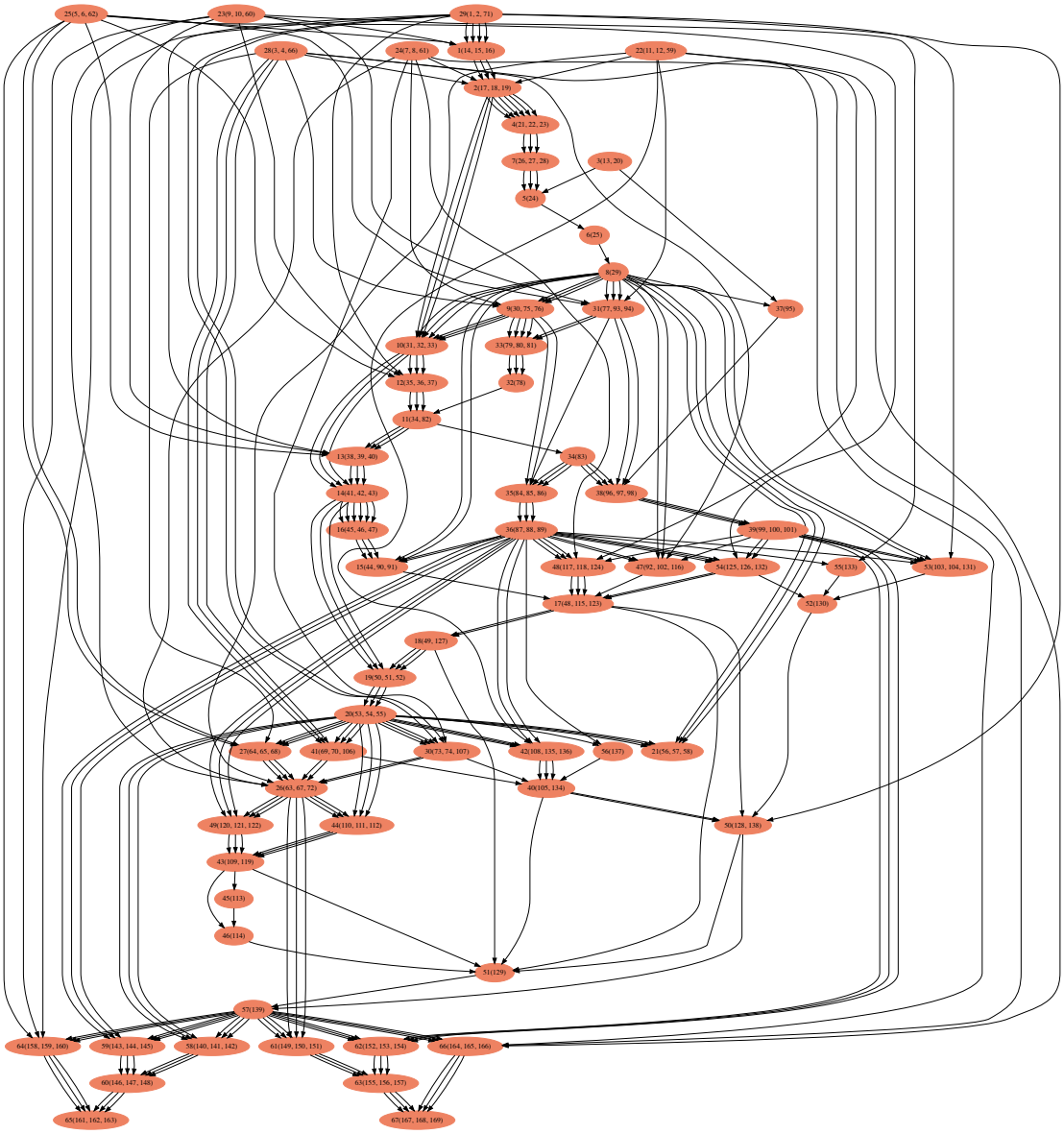


Figure 5.3: Lobpcg graph after pre-processing with every 3 nodes in the same topological level are coarsened into one single node and the edges are kept intact

algorithm does not compute any gains on the edgecut. It only fills the part as long as there is space in this part with the upper bound. We modified the DFS to BFS traversal but it did not have a major influence on the results.

It is only natural that the algorithms like kernighan DFS, Greedy Graph Growing which computes the gain and takes the edge cuts into account might not have an even partition. Hence, a forced balancing function is called afterwards to make them balanced according to the upperbounds. We did not modify anything in the forced balancing function.

5.3 Uncoarsening/Refinement

After the partitioning algorithm returns the coarsed graph with two partitions, this graph is uncoarsened and refined. Previously we stated that while coarsening, in each step the new node number in the new coarse graph is mapped with an old node number in the previous graph before coarsening. In each coarsening step, these records are kept. Also the leader information is kept in each coarsening step. The uncoarsening step is mainly a project back from a later graph to an older graph. The partitioning will return two partitions of the coarse graph with some vertices will be in part 0 and some will be in part 1. While projecting back, in each coarsening step, the matched node is assigned the part of its leader. For example, let us assume that in a coarsening step, (u, v) was matched and u became the leader of v and node u was renumbered in the next step. Suppose u is partitioned using partitioning algorithm and assigned in part 1. While projecting back, v is assigned in the same part as its leader which is 1. This is how the entire graph will be projected back to a previous coarsened version. In Figure 5.4 we can see a simplified example of how the projectback function assigns the part numbers accordingly. The third graph is the most coarsened graph and is fed to the

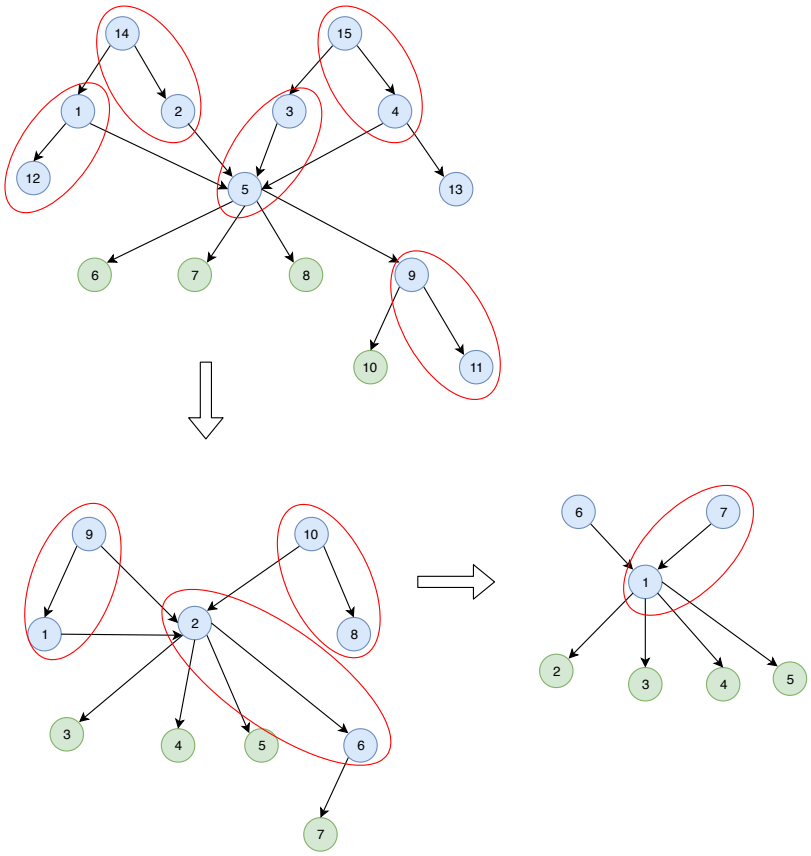


Figure 5.4: Coarsed graph Partition assignment example, blue is part 0 and green is part 1

partitioning algorithms. The algorithms returns a partitioning where $\{1, 6, 7\}$ in part 0 and $\{2, 3, 4, 5\}$ in part 1. While projecting back, the nodes which were matched in the previous coarsened step are assigned to the same part as the part of the leader node. Eventually in the end we can see that, among 15 nodes, only 4 will go into part 1 and node 11 will be assigned to part 0.

After this uncoarsening phase, a refinement step is called. We can clearly see that even in our small example, one part gets 11 nodes whereas the other part gets 4 nodes only. According to the vertex weight scheme, there is an imbalance here. That is why the refinement step is called. The refinement step moves some nodes from one part to the other part in a way that does not violate the acyclicity constraint.

First they compute where can each nodes be moved. As in actual code, only two partitions are generated, they check whether a node can be moved from this part to the other part. A list of boundary vertices are created and their gain are computed accordingly. A heap is maintained to extract the node with highest gain. This entire moving of nodes are continued until one part is greater than the upper bound. After the refinement phase, some nodes are moved to the other part.

5.4 Partitioned Graphs

For analyzing the quality of partitioning we are getting from this partitioner, we concentrated on the csb blocks that are accessed during a task. The sparse matrix is only accessed during the SPMM task and it is the dominating attribute that will occupy the majority of the memory. We wanted to see how are the matrix blocks are traversed in a part, or if there is any specific pattern present in their accessing order.

Let us see the csb block accessing order in different partitions. In Figure 5.5 we show the block access pattern for a sparse matrix with dimensions of 500,000*500,000 having 1000000 nonzeros. This is a sparse matrix generated by ParMAT library. In this case we assume each csb block to have 512*512 dimensions. That makes the entire sparse matrix a block matrix having 977*977 blocks. We ran a Lobpcg algorithm on this sparse matrix and generated a DAG, then fed this DAG to get a partition with 32 parts. For convenience, we color only part 1, 2 and 3 in this graph.

Straightway we start seeing a pattern here. The accessing of sparse blocks are in a column order in a part. Almost all the column blocks of a particular column falls in the same part. Another interesting thing is not necessarily all the adjacent columns will fall into the same part. For example, blocks in column 897,709,699 and 479 falls in the same part(Part 1).

Also let us see the partitioning result for our pre processed graph for a 61*61 sparse block matrix in Figure 5.6. Here we show the block access pattern for part 20,21 and 22. In this figure we can see that the sparse blocks are accessed in the same column order way. Although its more scattered now for some parts. As we are coarsening every 3 nodes in the same topological level without any relation among themselves, the access pattern becomes a bit more scattered.

The reason behind this kinds of pattern leads us to the kind of graph that we have. As we already have mentioned that we will have some nodes with high degrees. Let us see a small part of the DAG that is actually generated. Also we will see how the actual matching takes place.

In this Figure 5.7 we can see that from INV,RBR node, there will be a lot of outgoing edges. To be precise, if the block dimension is $nblocks * nblocks$, then there will be $nblocks$ number of outgoing edges. Also, from each DLACPY node there will be $nblocks$ number of

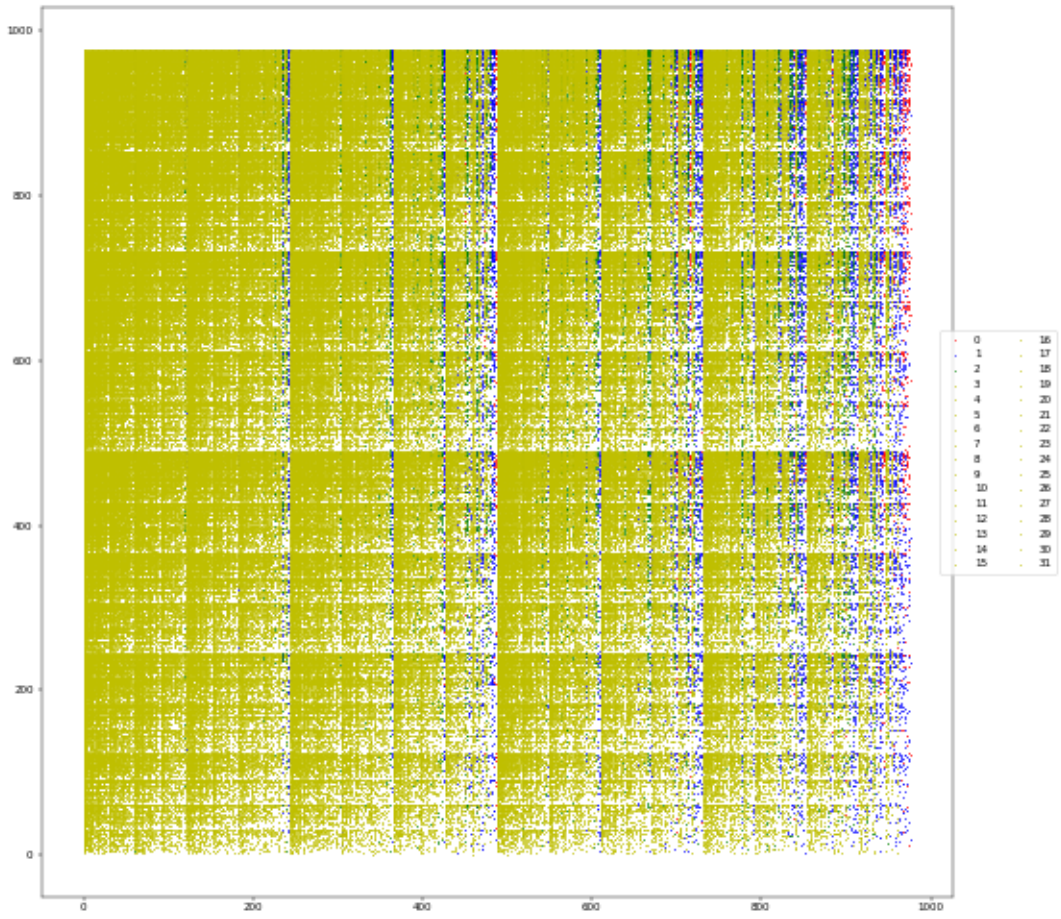


Figure 5.5: Sparse matrix block access patterns in different parts with matching

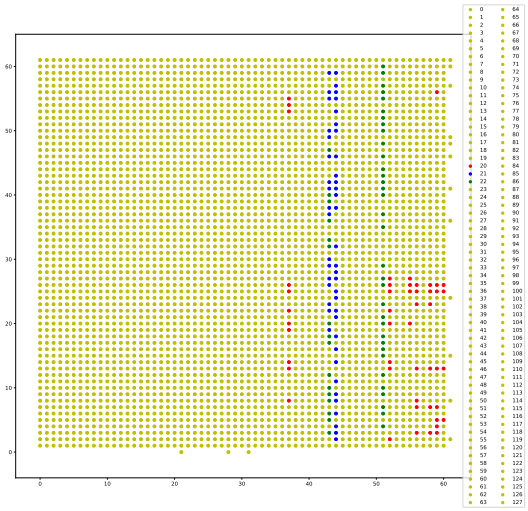


Figure 5.6: Sparse matrix block access patterns in different parts with pre-processing before matching

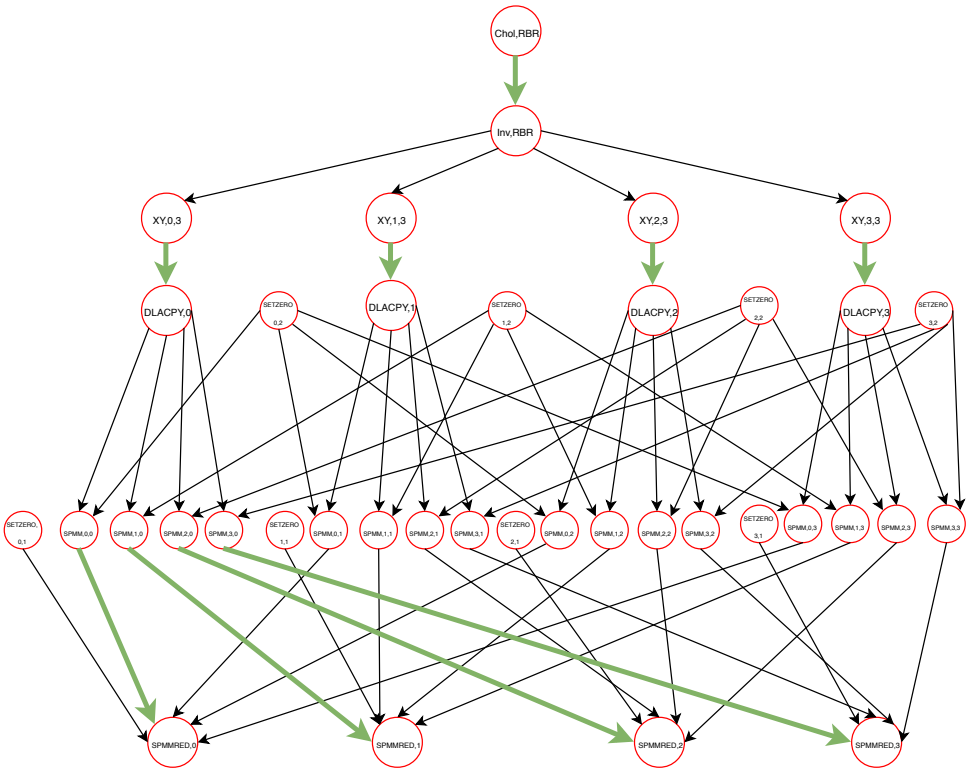


Figure 5.7: A Small part of the Original DAG that is generated. The matched edges are shown in green color

outgoing edges to all the SPMM tasks where the column blocks of the sparse matrix will be accessed. Also we can see there are *nblocks* number of SPMMRED nodes which have incoming edges from *nblock* SPMM nodes.

As we previously assumed, with higher degree nodes, what happens is that once these high degree nodes are matched, all the other edges incident with that node are not considered at all. This results in a lot of nodes to be left out from consideration. Hence, in the coarsened graph, the number of nodes are not reduced that much and gets kind of saturated and the matching process becomes slow.

Another very important phenomena is that, while matching the edge (u, v), u becomes the leader of v in the coarsened graph. In this graph we show the matched edges in the first coarsened level with thick green arrows. We can see that, CHOL,RBR node will become the leader of INV,RBR node. All the XY nodes will become the leaders of their subsequent DLACPY nodes. Since all the DLACPY nodes are matched now, all of the edges from DLACPY to SPMM nodes will not be considered. Rather, only the *nblock* number of edges from SPMM to SPMMRED will be added to matching set.

In the next coarsening level, shown in 5.8 we start seeing a pattern where clustered XY nodes (combined with XY and DLACPY) have outgoing edges to SPMM nodes and some of those edges are included in the matching set. For example the edge from XY,2,3 to SPMM,0,2 is added to the matching set making XY,2,3 the leader of SPMM,0,2 and resulting in not considering any other edges incident with XY,2,3.

In subsequent Figures 5.9, 5.10 and 5.11 we show the later coarsening stages. Here we start seeing that pattern where the edge from XY,2,3 to SPMM,1,2 and then SPMM,2,2 and then SPMM,3,2 are matched in consecutive coarsening levels. Making all the SPMM tasks along the same column clustered in the same XY node. This also happens with other

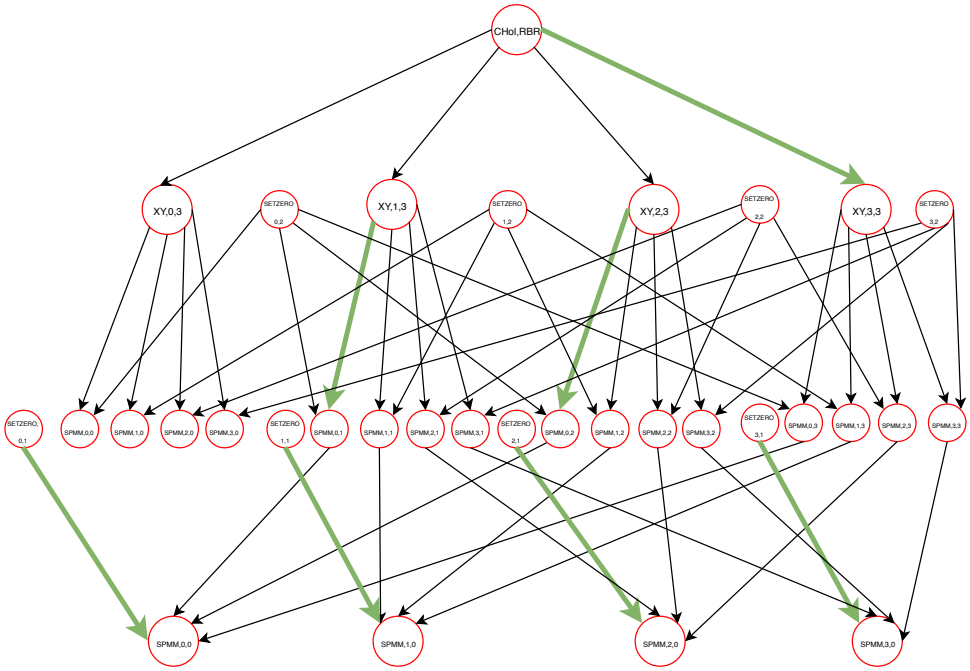


Figure 5.8: step 2 of the matching

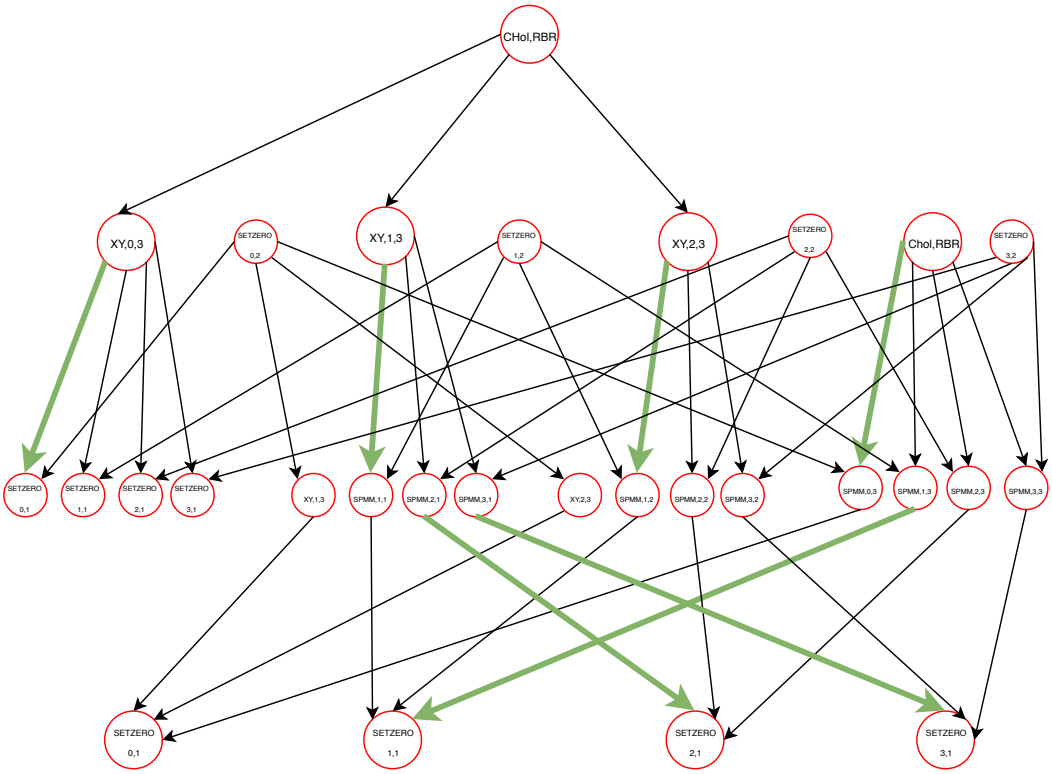


Figure 5.9: step 3 of the matching

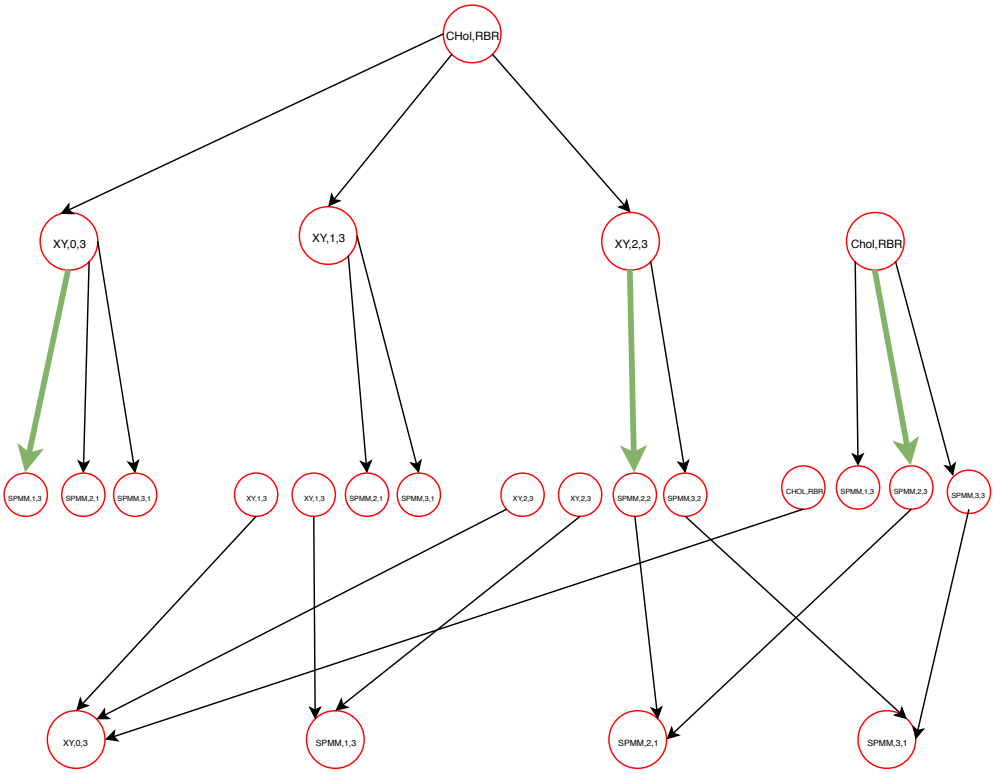


Figure 5.10: step 4 of the matching

XY nodes where all the SPMM tasks along the same column becomes clustered in the same node. Moreover, a large cluster forms with CHOL,RBR as the leader. This continues to happen until we reach to our step size.

5.5 Coarsening A Block of SPMM Nodes Into One Block

Since the SPMM tasks along the same column gets clustered in the same node, all the column blocks gets into the same partition. So we tried a different proprocessing approach. Rather than coarsening nodes in the same topological level, we concentrated only on tasks which access the main sparse matrix blocks. Before sending the graph to the regular matching

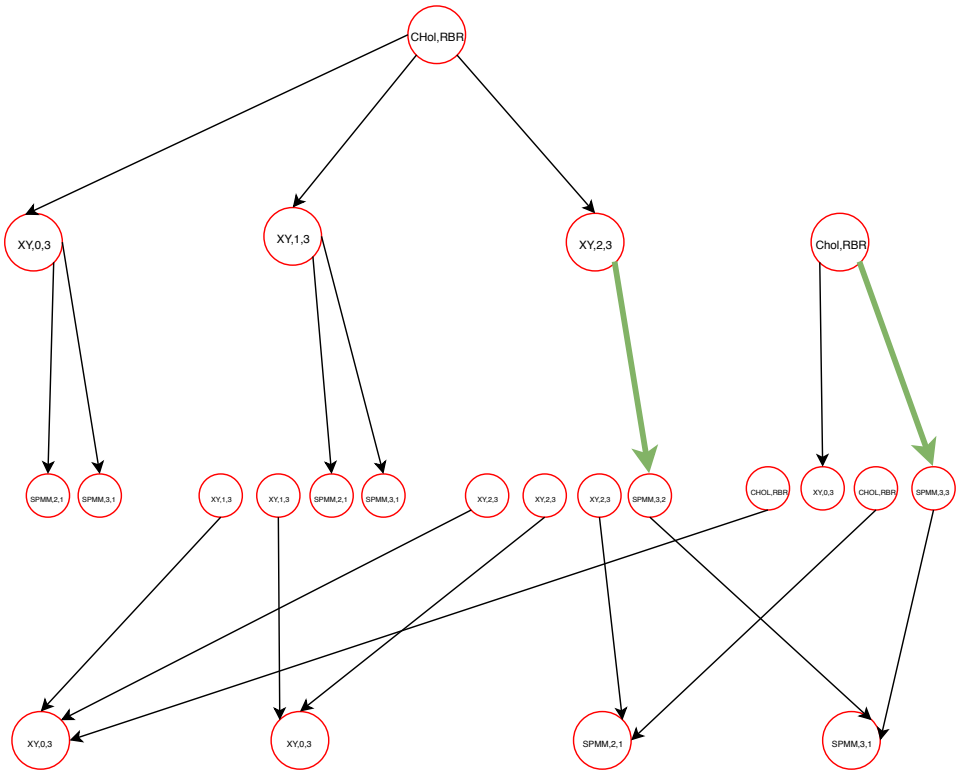


Figure 5.11: step 5 of the matching

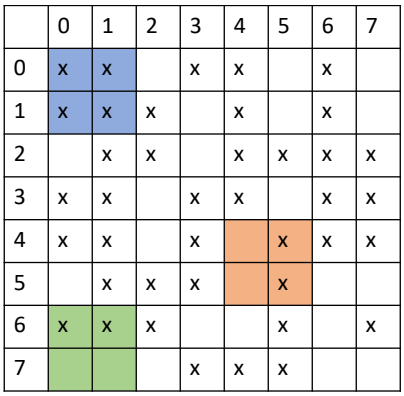


Figure 5.12: blocking of csb blocks to coarse multiple nodes into one node

algorithm, we coarsen the nodes which are accessing to the csb blocks which form a small block. Assuming we will coarse bxb nodes into one node, we divide the entire block matrix into $(nblocks/b) * (nblocks/b)$ small blocks. This will be the maximum number of nodes after coarsening. The block number for each SPMM node is calculated and the first visited SPMM node in a row-major traversal from that block is set as the leader. If there are some some blocks with no nonzeros, that means that SPMM node will not be present in the DAG. That node will not be considered anyway. Let us see our method in this Figure 5.12

Here let us assume we have a sparse block matrix with 8*8 csb blocks. x denotes that there is at least one non zero in this block. We take $b = 2$ for this example. That means there will be 4*4 small blocks. The blue block will have SPMM (0,0), SPMM(0,1), SPMM(1,0) and SPMM(1,1) nodes. In row major order, SPMM(0,0) will be the first node visited from this block. Hence we make the SPMM(0,0) the leader of the other nodes in this block.

There will be some small blocks where some csb blocks will have no non zeros. In the figure, the orange block only has two csb blocks with non zeros. For this kind of cases, first node traversed from this block(SPMM(4,5)) becomes the leader of the other node. In this case only 2 nodes will be coarsened.

The motivation behind trying this scheme was to find out if we can actually see any

difference in the 1d partitioning results because this time we are coarsening a small block into one node. In figure 5.13 we see the results for a similar matrix from figure 5.14. Here we have $b = 5$ that means we coarsen 5*5 nodes into one single node. We see that the thickness of the csb blocks accessed has somewhat increased. But overall it is still a 1d kind of partitioning. This makes sense as now some adjacent columns will also be included in the same part thats why the thickness increases. But those DLACPY nodes will still have outgoing edges to $(nblocks/b)$ nodes which will behave same.

5.6 Issues with the Partitioner

5.6.1 Upperboounds

issue is that they rely on the upperbounds and lower bounds while partitioning. These partitioning bounds are first decided based on the vertex weights. There initial examples had vertex weight of one. For example, if we have 100 vertices, and we want to create 4 parts using the partitioner, each part will have an upperbound of ~ 27. BFSGrowing fills one part until it is under this upper bound, regardless of the gains.

5.6.2 Refinement

In my opinion this is an important issue that is making the csb block accessing pattern more scattered. When we project back, each node is assigned the part number of its leader. Because of regular matching algorithm and also the pre processing that we tried, obviously one part becomes larger than the other part. In this case the refinement function calls a forced balance function. In refinement step, boundary vertices are selected and the gain for

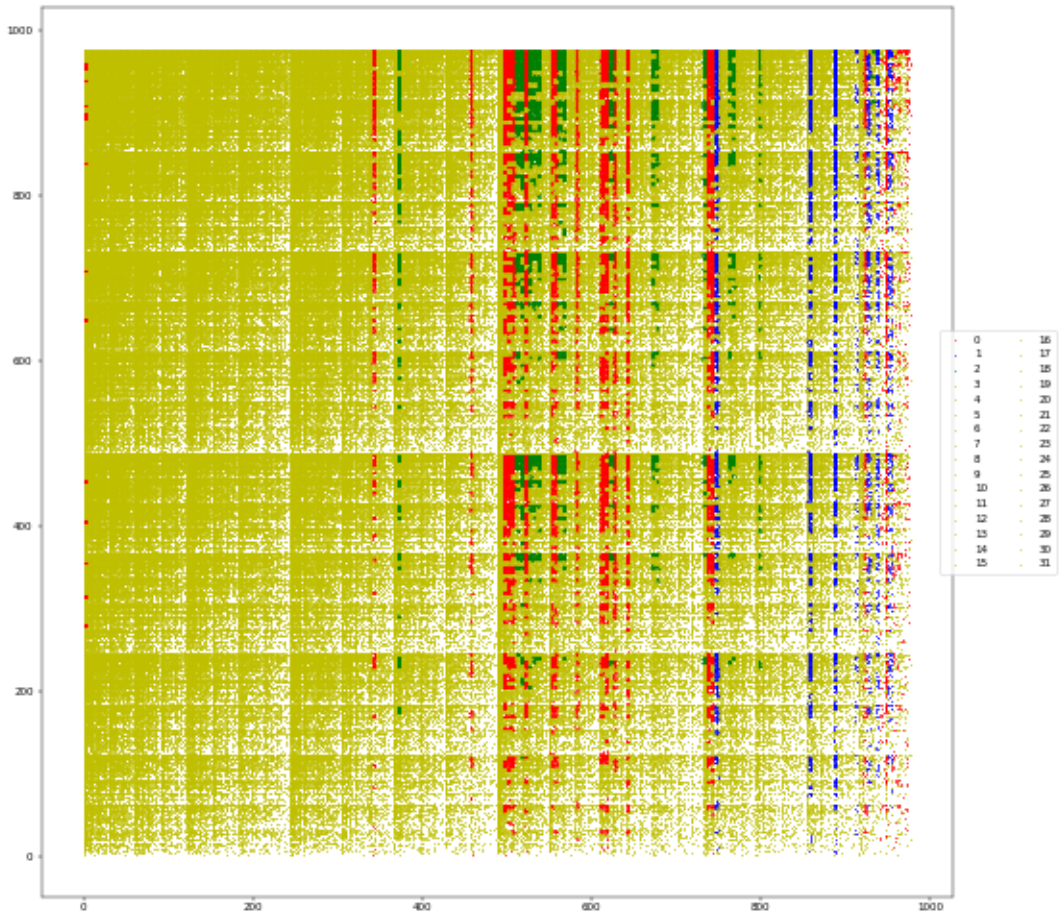


Figure 5.13: Sparse matrix block access patterns in different parts

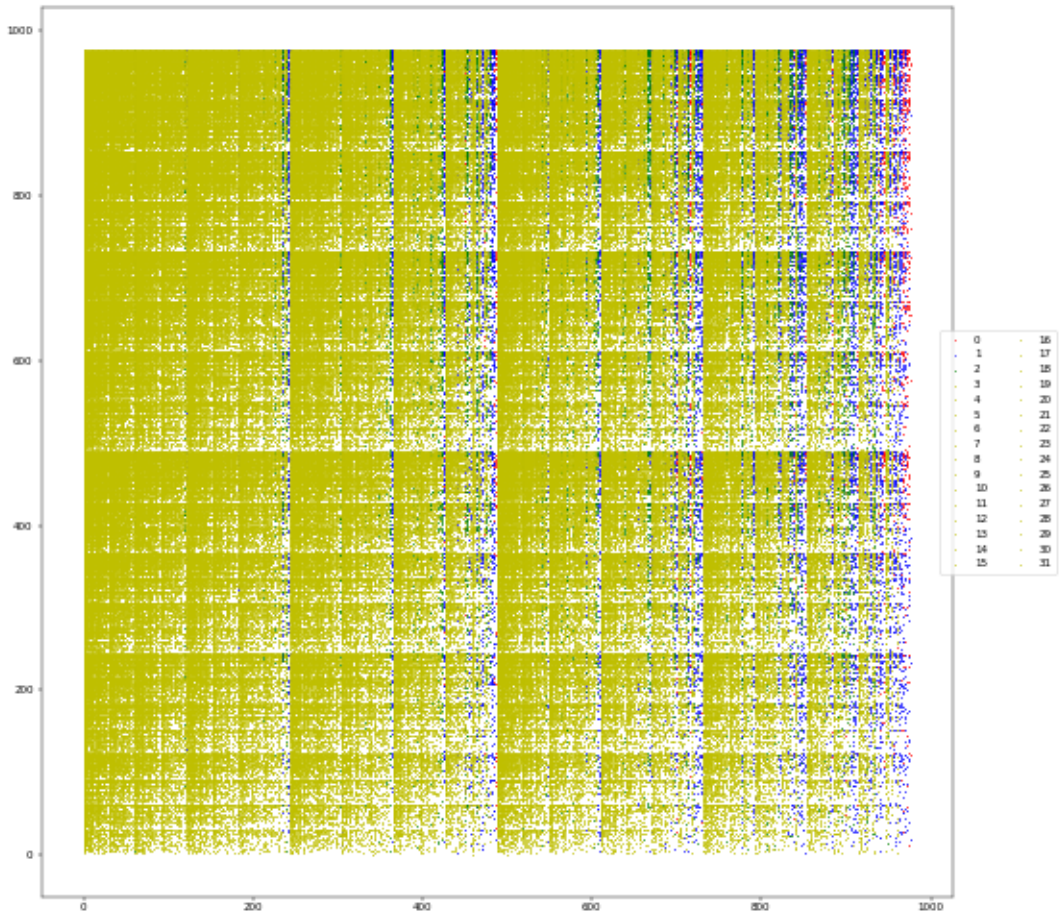


Figure 5.14: Sparse matrix block access patterns in different parts

their move to the other parts are calculated. Currently nodes from the big part are moved to the small part only. Hence some nodes gets moved to the other part anyway. At each uncoarsening step, after each project back call the refinement is called. As we have seen that SPMM column nodes are clustered into one node DLACPY most of the time, different DLCPY nodes fall into same part in a coarsened graph. While projecting back, the SPMM nodes associated with them gets into that same part making the non adjacent columns in the same part. Moreover, the refinement process moves some nodes to the smaller part making the accessing more scattered.

5.6.3 Graph structure

Obviously our graph structure leads us to different kinds of issues. What happens with a node with high indegree or high outdegree. The matching becomes slower because of this kinds of nodes. If we want to avoid this saturation in matching, then we need to increase the coarse graph size which might increase the time for partitioning. Also, because of this regular matching, the column blocks are getting clustered into one node which leads us to a 1d partitioning. 1d partitioning will not be the best for memory usage. We can think this graph as a hypergraph somehow. The codebase will need a huge change then. But may be with hypergraphs we can come up with some agglomerative approaches for matching.

5.6.4 Edgecut

EdgeCut is calculated on the entire graph in different stages of the algorithm. Weights of every edge which has its endpoints in different parts are added to the edgecut. While generating partitioning, the gains are considered. At this moment, the minimum of edgecuts

are considered. But for our purpose, the more the edgecut is, the more data reuse between the parts.

5.7 PowerLaw Graph Partitioning Attempts

There are two main issues that we faced with the DAGs we were working with. As we have already seen the structure follows a specific pattern where some nodes have a lot of outgoing edges and some nodes have a lot of incoming edges. This graph maintains a power law graph shape. We know from power law graphs that it follows a power law relation.

Abou-Rjeili and Karypis discusses a multilevel partitioner approach for the irregular graphs which follows power law distribution. They discuss similar problems that we face with our graph structure and matching. Once the nodes with higher degrees get matched, we can not hide a lot of edges and thus it makes the whole matching process slow. Hence, the coarsening steps become more and more slow resulting in increase of memory required to store these intermediate coarsened graphs. In our current method, we do not consider a vertex at all once it gets matched. But they propose a scheme where a vertex will still be considered for including in the matching set even if it is matched.

They have two edge visiting strategies - *globaly greedy* strategy (GG) and *globally random locally greedy* strategy(GRLG). In GG they order the edges according to some pieces of information and then take those edges in a greedy approach. In GRLG method, they randomly choses some vertices and then greedyly considers the edges incident to those vertices. As our graphs are directed, GG strategy will not for us because we need to maintain the acyclicity though out our approach. Hence we tried with GRLG approach.

We selected a random order for visiting the vertices. Then we ordered the edges incident

to that vertex with the edge weights. The highest weighted edge with that vertex gets the most priority. The purpose is to cluster the nodes which have a high data movement between them to be in the same cluster so that we can gain some data reuse. But this approach although worked for some random orderings, did not work for all the random orderes. It generated a cycle after some matching step. Here we present an example of this kind of case.

In this figure 5.15 we show a possible cycle generation using the above mentioned approach. We show an actual part of our graph which follows that kind of structure of one node having a lot of outgoing edges. For ease of explanation we are naming them as node numbers. Here, we see that in the first graph when node 8 is chosen, all of its adjacent edges are selected for matching. In the earlier case, only one of these edges would be chosen and all of them would be replicated in the coarsened graph hiding all those similar edges. But in this case, all the similar kind of edges incident to node 8, 10, 12 and 14 are matched. This was one of the main motivations behind the aforementioned paper. In the next graph we can see the coarsened graph after the first stage with updated vertex number. In the next graph similarly with a random order some vertices and their adjacent edges are matched. But this time it creates a problem. Since the vertex numbers are updated accordingly to the leader of the matched edge, in the third graph we can see that there is an edge from node 1 to node 9 and also from node 9 to node 1. This is a cycle so this approach will not work for us. With undirected graphs, these schemes work pretty well as they have showed but for our case unfortunately we cannot implement these ideas as they were stated.

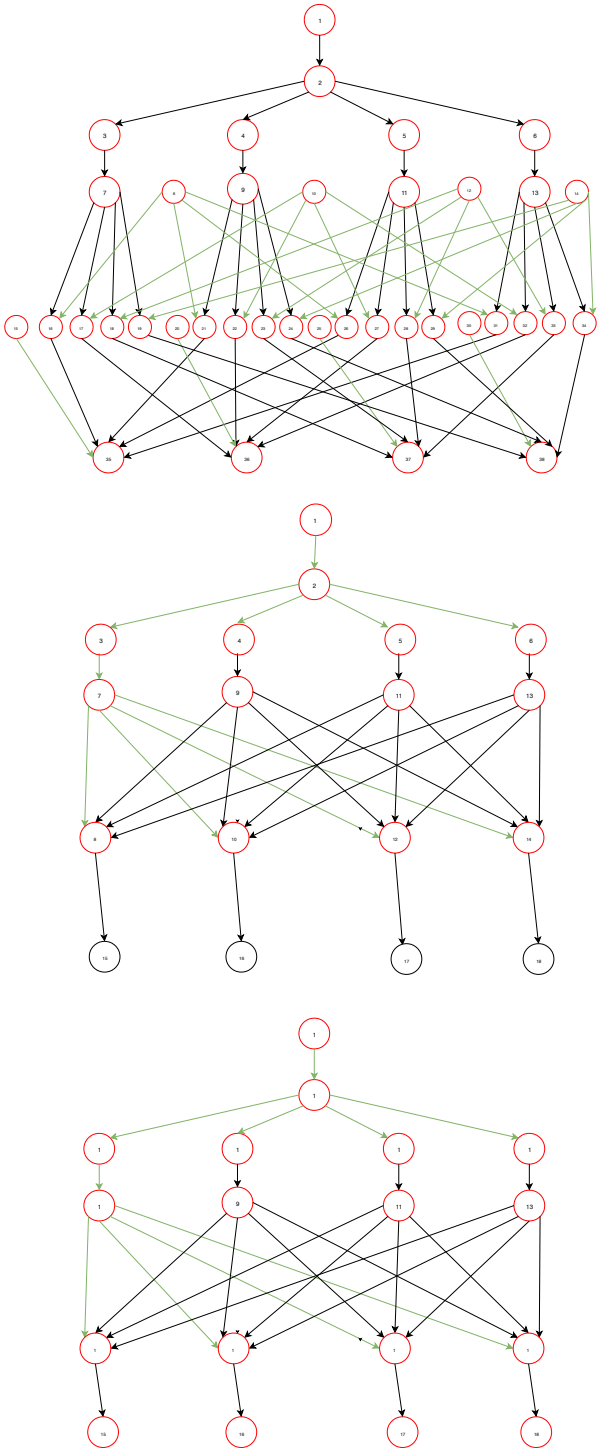


Figure 5.15: A cycle is created using GRLG approach

5.7.1 Lowest Common Ancestor

To get rid of this cycle issue, we came up with a different method. We calculated the lowest common ancestor of two nodes in the graph. Then while the matching process, we check whether the other parents of a children of a node we are considering for matching shares a lowest common ancestor. If they have a common ancestor, that means it might be possible that it creates a cycle. Because in some later coarsening steps, the vertex we are considering might be matched the common ancestor and resulting an edge from the other sibling to have an edge to the common ancestor and thus creating a cycle. Hence, if the other parents of this children have a lowest common ancestor with the vertex we are considering, we do not match this edge. We do it for all the vertices.

For example in figure 5.15 second graph, when we are considering node 7, its children is node 8 whose other parents 9, 11 and 13 all have a lowest common ancestor 2 which is also an ancestor of node 7. This means, if node 7 is matched in some coarsening step with node 2, then it will create a cycle. So in this case we do not match node 7 and 8.

Although this scheme never creates a cycle in the coarsened graph, but unfortunately it stops the coarsening process too early. The main reason is that our graph structure has a lot of nodes who share a common ancestor at a very high level of the graph even though those nodes are in the down level of the graph and actually might never be matched together. In the earlier coarsening steps we see some matching being done but after some steps, all the nodes have some common ancestors with some nodes resulting in no matching thus stopping the matching process. This scenario does not serve our purpose. Also, another important motivation for us is to match in such a way so that the matrix can be traversed in a blocked shape. That purpose is also not served.

5.7.2 Hierarchical partitioning attempts

We have already discussed that our graph maintains a certain structure which leads to a specific pattern of accessing along the columns of the large matrix in a partition. This kind of 1d partitioning will not be ideal for cache utilization since the entire output block vector needs to be kept in the memory for all the columns and so for almost all the partitions. We would like to access the matrix in a 2D kind of shape so that we can improve the cache utilization. Keeping this thing in mind, we have tried another approach which hierarchically breaks down a large tasks into small tasks.

We start our partitioning with tasks having a large matrix block size eg. 64K, 32K etc. With large block size, the number of nodes in the DAG will be relatively small. Hence our regular partitioning algorithms which struggled with partitioning large number of nodes can partition this graph with relatively small nodes with ease. Hence we will generate a partitioning using a large block size. The problem with large block size is that it will incur a large number of unnecessary calculations. For example in a $64K * 64K$ block, there might only be a few nonzeros but eventually we will be needing to apply all the operations into these blocks where almost all the values are zero. In figure 5.16 we show the partitioning achieved in the Z5 graph. We can clearly see that there are less number of blocks and they have mostly that 1D pattern.

But for efficient computation, we would like to have small blocks of the matrix. The main reason is that we can skip so many unnecessary calculations we were doing with large blocks. Also the cache utilization will increase with small block sizes. Hence we divide each large block into small blocks. We create a completely new graph with these small blocks. All the tasks which executes on a block whether a matrix block or a vector block will be

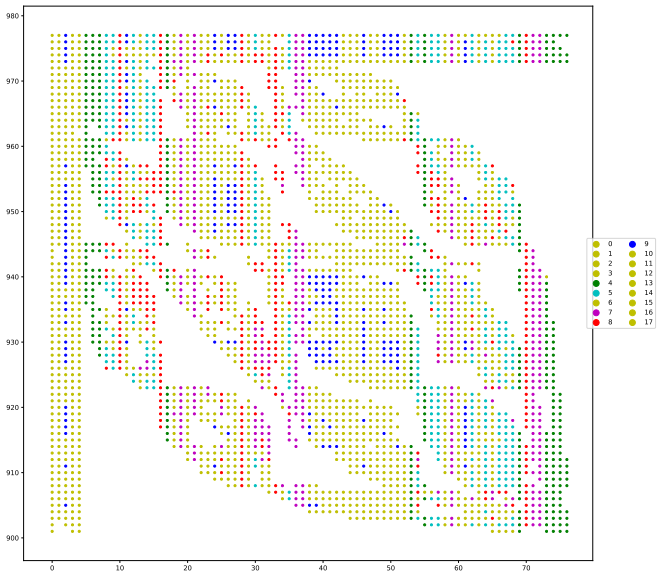


Figure 5.16: Partitioning of Z5 matrix with 64K block size

replicated into newly constructed task nodes with small blocks with appropriate incoming and outgoing edges.

In figure 5.17 we show an example of how the hierarchical blocking is actually done. Here we have a matrix which is divided into 4*4 blocks. We will use this graph for partitioning. After we get the partitioning, we will refine this graph into a different graph with small block size. Each of the blocks in the large blocks are divided into further small 4*4 blocks. Hence, for each large matrix blocks, we will have 4*4 = 16 new nodes and they will be renamed appropriately. Note that, when we refine the large blocks into small blocks, a lot of small blocks will have no nonzeros in them. We will consciously skip those nodes from our refined graph since those nodes will not incur any valid computation. Hence the original $spmm, 0, 0$ node will be divided into 16 new *spmm* tasks starting from $spmm, 0, 0$ to $spmm, 3, 3$. All the vector blocks will also be divided into small blocks. That means a $dgemm, 0$ task will be

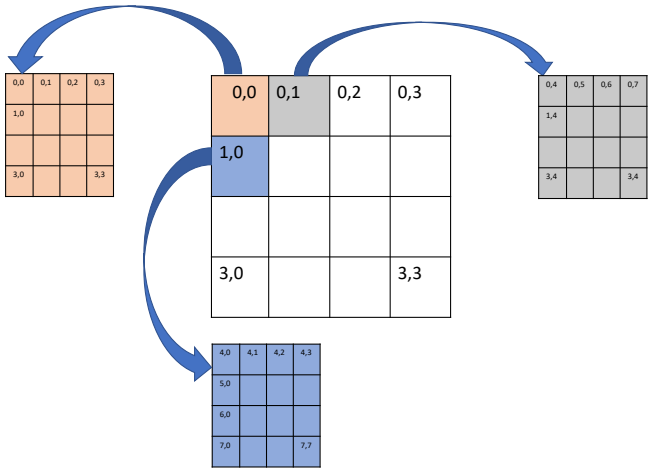


Figure 5.17: Hierarchical blocking scheme

divided into 4 *dgemm* tasks starting from $dgemm, 0$ to $dgemm, 3$ and so on. All the edges in the original graph will be converted to appropriate number of edges to appropriate nodes accordingly.

As we already know that the partitioner assigns a partition number to each task. Hence, the part number of the tasks in the original graph will be replicated to the part number of the tasks in the refined graph with small block size. Since each node in the original graph is divided into small blocks, we expect to see some 2D kind of shape in the refined graph. In figure 5.18 we have the refined graph where each block from the original graph has been divided into $64 * 64$ blocks of $1k$ size. We can see how the matrix is accessed in the refined graph. We still see a slight column based access but in this case there is a blocked shape in this case which should increase the data reuse that we are looking for.

5.8 Memory Bound Implementation

One of the most important aspects of our partitioner is that it ensures a constraint is strictly maintained in all partitions. In our case, the constraint is the fast memory. All our partitions

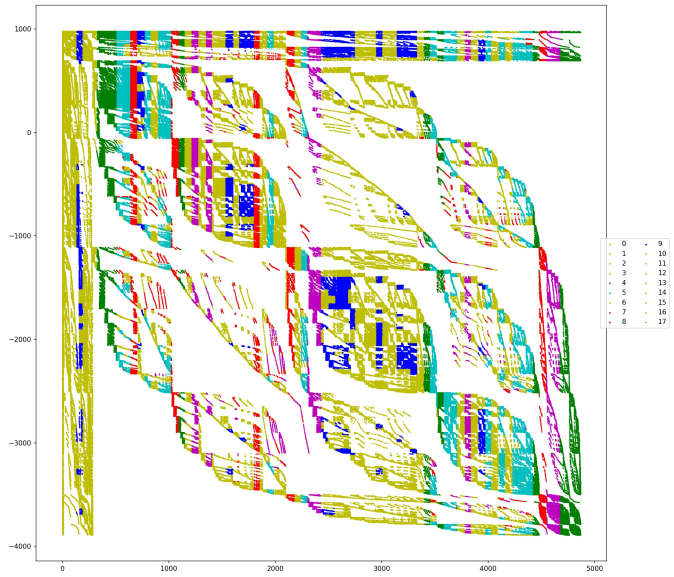


Figure 5.18: Partitioning of Z5 matrix with 1K block size

follow this memory bound. We have two important phenomena to consider while managing this constraint over all the partitions.

We should correctly identify all the active memories that will be needed during the execution of this part. As we design our partitioner in such a way that we assume all the tasks in this partition will be executed in a single node, that means all the active memory parts that will be needed, allocated for both input and output of individual tasks should contribute in the active memory calculation. Not only input and output memories which will be read or written, but also the amount of internal memories allocated for executing that task (Such as temporary memories allocated in heap through malloc which will have a scope on that task only) should be considered while active memory calculations.

Another important and motivating aspect behind our partitioner is to catch the data

reuse between the tasks whenever possible. The DAG generated has some dependencies between the tasks where the edge represents the data flow from one task to another. Note that the memory needed by a task as inputs are already written as output memory by another tasks. Hence, we can safely say that memory chunk needed for writing will be present in the fast memory and when a task that needs that memory chunk as an input memory for execution, it will find it in the fast memory. This memory chunk is active and it should not be counted multiple times.

Let us give an example of these two considerations. In figure 5.19 we can assume a partition consisting of some tasks named A, B, C, D, E and F which are represented as nodes and the edges here represent the memory chunk these tasks need as input and the memory generated by these tasks as output which goes to other tasks as input. For example, task D has incoming edges from task A and B and outgoing edge to task E. Task D receives the memory chunks a and b as input from tasks A and B respectively. It generates an output d which is then forwarded to the task E as its input.

While the tasks are being executed, we can see here that task B and C uses a common memory chunk $p2$ as input. If task B is executed first, memory $p2$ will be loaded in the fast memory. Hence, when C will be executed, $p2$ is already in the fast memory. Hence while calculating active memory, we should not include $p2$ again in our calculation. We will keep adding all the active memory needed for each of the tasks unless they are already in the active memory. When task E needs to be executed, the input memories it needs are d, b and x(from some other part). Since both d and b are already in the fast memory the values of these memory chunks will not be added to the active memory calculation. But since x has not been yet included in the active memory, it will be included in the calculation.

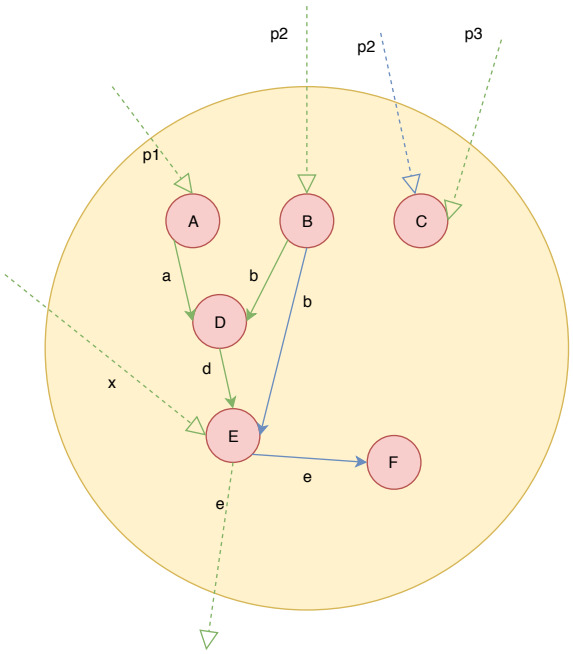


Figure 5.19: Memory Management in partitions

We will also include all the temporary memory allocated in each tasks which only has a scope on that task only but since they are allocated, those memories are also part of active memory in this part.

After all the active memory calculations, we can measure if the total active memory needed in this partition is under our fast memory amount. If it is less than the fast memory amount then we can easily go ahead with the execution of these tasks in this part as it will fit in the fast memory. If the amount of active memory is more than the amount of fast memory, that means we need to further partition this part.

5.9 Experimental Results

5.9.1 Experiment setup

We conducted all our experiments on Cori Phase I, a Cray XC40 supercomputer at NERSC, mainly using the GNU compiler. Each Cori Phase I node has two sockets with a 16-core Intel Xeon Processor E5-2698 v3 Haswell CPUs. Each core has a 64 KB private L1 cache (32 KB instruction and 32 KB data cache) and a 256 KB private L2 cache. Each CPU has a 40 MB shared L3 cache (LLC). We use thread affinity to bind threads to cores and use a maximum of 16 threads to avoid NUMA issues. We test DeepSparse using five matrices with different size, sparsity patterns and domains (see Table 4.2). The first 4 matrices are from The SuitSparse Matrix Collection and the Nm7 matrix is from nuclear no-core shell model code MFDn.

We compare the performance of partitioned schedule with two other library implementations: i) **libcsr** is implementation of the benchmark solvers using thread-parallel Intel MKL Library calls (including SpMV/SpMM) with CSR storage of the sparse matrix, ii) **libcsb** is an implementation again using Intel MKL calls, but with the matrix being stored in the CSB format and our DeepSparse implementation. Performance data for LOBPCG is averaged over 10 iterations, while the number of iterations is set to 50 for Lanczos runs. Our performance comparison criteria are L1, L2, LLC misses and execution times for both solvers. All cache miss data was obtained using the Intel VTune software.

5.9.2 Performance of the partitioner

We ran our custom scheduler in both Haswell and Knl nodes and measured the L1, L2 and last level of memory(in this case MCDRAM) misses using VTune in similar way to our

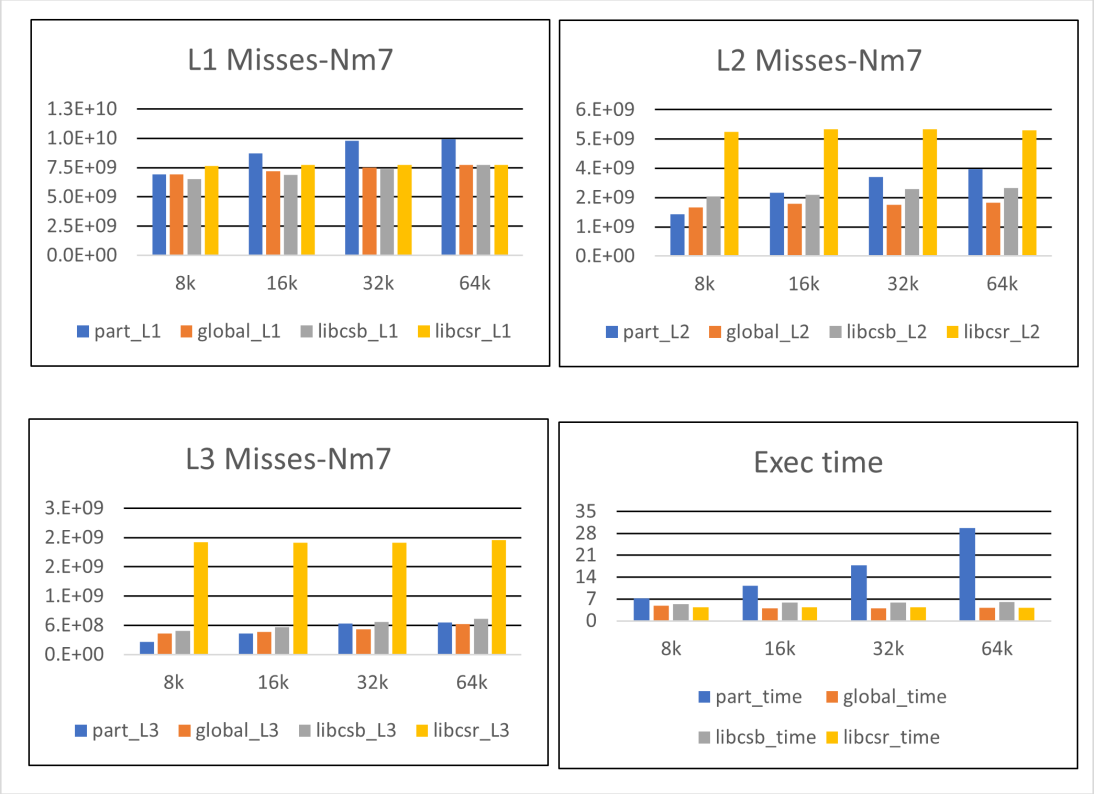


Figure 5.20: Performance comparisons between different cache levels and execution time of Nm7 matrix in Haswell nodes

previous project.

In Figure 5.20 we show the cache miss and execution time results for Nm7 matrix in Haswell nodes for different block sizes for our execution. For L1, we do not see much improvement for our custom scheduler. But for L2 and L3 caches, we see the custom scheduler achieves better cache performance over libcsr version. But the improvement over DeepSparse was not consistent and often it could not beat the DeepSparse performance. Also we noticed that the execution time is actually not as we expected for the custom scheduler.

In Figure 5.21 we show the cache miss and execution time results for Nm7 matrix in KNL nodes for different block sizes for our execution. We see similar traits here and additionally we get better L1 cache optimization in this case. But likewise, we could not outperform the DeepSparse for KNL nodes too.

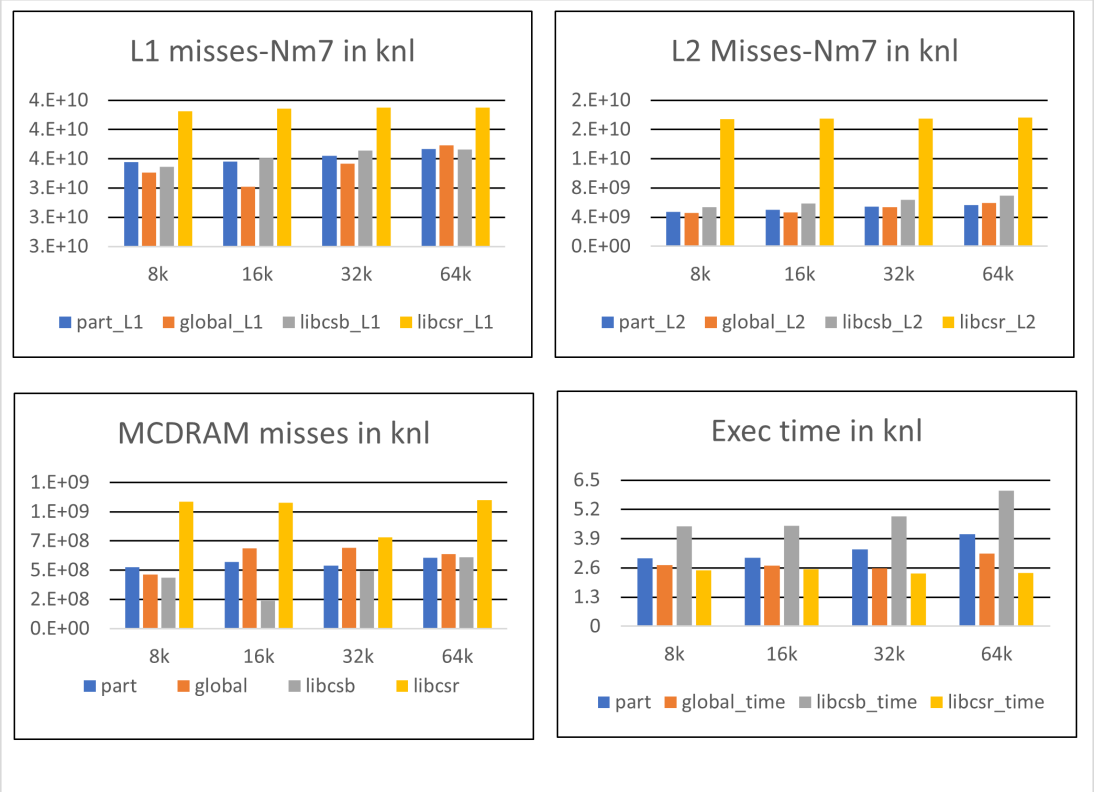


Figure 5.21: Performance comparisons between different cache levels and execution time of Nm7 matrix in knl nodes

5.10 Future work on the partitioner

We tested the performance of the custom scheduler in other matrices for both Haswell and knl machines. We saw some similar traits in all of our experiments. We achieve an improvement over the library versions as we expected. But we can not out perform the DeepSparse performance for all matrices with all the block sizes. That means the OpenMP default scheduler is doing a pretty great job as well as our custom scheduler is not performing as we expected. One important aspect is having to use a taskwait after each part is holding us back a little bit. Because if we do not use this taskwait between each partition, since we are still using OpenMP for executing all the tasks of a partition together, some other tasks from a different part can be scheduled before a task from this partitioner if its in out dependencies have been resolved, which will hinder our performance.

Hence we thought we will be needing some other heuristics if we want to use the partitioner with a better scheduler. An idea might be to use hypergraphs and partition the hypergraph. But converting the graphs to hypergraphs and converting every data structure to hypergraphs would be a cumbersome work. We still hope to continue trying to improve the partitioner. At this point we wanted to move on with the existing scheduler served by OpenMP and try and implement the distributed MFDn. Before that we wanted to see the communication pattern and their behavior in MFDn. We use a simulator called SST to simulate the networking topology in MFDn which we discuss in detail in the next chapter.

Chapter 6

SIMULATING THE COMMUNICATION PATTERNS IN A LARGE SCALE DISTRIBUTED APPLICATION

Our target application, MFDn [89, 90, 4] is used for ab initio calculations of the structure of atomic nuclei. The structure of an atomic nucleus with A nucleons can be described by solutions of the many-body Schrödinger equation

$$\hat{\mathbf{H}}\,\Psi(\vec{r}_1,\ldots,\vec{r}_A) \;=\; E\,\Psi(\vec{r}_1,\ldots,\vec{r}_A) \tag{6.1}$$

where $\hat{\mathbf{H}}$ is the nuclear Hamiltonian acting on the A-body wavefunction $\Psi(\vec{r}_1,\ldots,\vec{r}_A)$, $\vec{r}_j$ the single-nucleon coordinates, and E the energy. For stable nuclei, the low-lying spectrum is discrete. The solution associated with the algebraically smallest eigenvalue is the ground state. The nuclear Hamiltonian $\hat{\mathbf{H}}$ contains the kinetic energy operator $\hat{\mathbf{K}}$, and the potential term $\hat{\mathbf{V}}$ which describes the strong interactions between nucleons as well as the Coulomb repulsion between protons. Solving for nuclear properties with realistic nucleon–nucleon (NN) potentials, supplemented by three-nucleon forces (3NF) as needed, for more than a

few nucleons is recognized to be computationally hard [124].

Obtaining highly accurate predictions for properties of light nuclei using the No-Core Configuration Interaction (NCCI) approach requires computing the lowest eigenvalues and associated eigenvectors of a very large sparse symmetric many-body Hamiltonian matrix $\hat{\mathbf{H}}$. If A is the number of nucleons in a nucleus, this matrix is the projection of the nuclear many-body Hamiltonian operator into a subspace (configuration space) spanned by Slater determinants of the form

$$\Phi_a(\vec{r}_1, \ldots, \vec{r}_A) = \frac{1}{\sqrt{A!}} \det \begin{bmatrix} \phi_{a_1}(\vec{r}_1) & \ldots & \phi_{a_A}(\vec{r}_1) \\ \vdots & \ddots & \vdots \\ \phi_{a_1}(\vec{r}_A) & \ldots & \phi_{a_A}(\vec{r}_A) \end{bmatrix}, \tag{6.2}$$

where ϕ_a are orthonormal single-particle wavefunctions indexed by a generic label a. The dimension of the subspace or basis spanned by these many-body wavefunctions Φ_a depends on (1) the number of nucleons A; (2) the number of single-particle states; and (3) the (optional) many-body truncation.

In the NCCI approach, one typically works in a basis of harmonic oscillator single-particle states where the number of single-particle states is implicitly determined by the many-body truncation $N_{\max}$, which imposes a limit on the sum of the single-particle energies (oscillator quanta) included in each Slater determinant of A nucleons. In the limit of a complete (but infinite-dimensional) basis, this approach would give the exact bound state wave functions; in practice, increasingly accurate approximations to both the ground state and the narrow (low-lying) excited states of a given nucleus often require increasingly large values of $N_{\max}$. The dimension D of $\hat{\mathbf{H}}$ increases rapidly both with the number of nucleons, A, and with

the truncation parameter, $N_{\max}$. The sparsity of the matrix $\hat{\mathbf{H}}$ (and hence the memory requirements and computational load) depend on the nuclear potential. With NN-only potentials this matrix is extremely sparse, whereas with 3NFs there are significantly more nonzero matrix elements in the matrix, and with a (hypothetical) A-nucleon force one would get a dense matrix [91].

Many-Fermion Dynamics—nuclear, or MFDn, is a NCCI code for nuclear structure calculations using realistic NN and 3NFs forces [125, 126, 127, 128, 129] written in Fortran 90 using a hybrid OpenMP/MPI programming model. A typical calculation consists of constructing the many-body Hamiltonian matrix in chosen basis, obtaining the lowest eigenpairs, and calculating a set of observables from those eigenpairs. Efficiently utilizing the aggregate memory available in a cluster is essential because a typical basis dimension is several billion - corresponding to a very sparse matrix with tens of trillions of nonzero elements.

The lowest few eigenvalues and eigenvectors of the very large real sparse symmetric Hamiltonian matrix are found with iterative solvers - using either the Lanczos [92] or the LOBPCG [93] algorithms. The key kernels in iterative eigensolvers are Sparse Matrix–Vector (SpMV) and Sparse transposed Matrix–Vector (SpMV^T) products, as only half of the symmetric matrix is stored in order to reduce the memory footprint. The sparse matrix is stored in a CSB_COO format [22], which allows for efficient linear algebra operations on very sparse matrices, improved cache reuse on multicore architectures and thread scaling even when the same structure is used for both SpMV and SpMV^T (as is the case in this application).

6.1 MFDn Communication Motif

On a distributed-memory system, we would like to distribute the matrix among different processing units in such a way that each processing unit will perform roughly the same number of operations in parallel sparse matrix computations. To achieve this, we split the sparse matrix into $n_d \times n_d$ approximately square submatrices, each of approximately the same dimension and with the same number of nonzero matrix elements, for both CPU time and memory load-balancing purposes. As mentioned above, since the matrix $\hat{\mathbf{H}}$ is symmetric and the number of nonzero elements in $\hat{\mathbf{H}}$ increases rapidly for increasing problem sizes, we store only half of the symmetric matrix. This results in $n_p = n_d(n_d + 1)/2$ submatrices for e.g. the lower-triangle part of the Hamiltonian. One can then distribute these submatrices over n_p different processing units, perform local SpMV and SpMV^T operations with these local submatrices, and synchronize after every iteration along both the columns and the rows of this grid of $n_d \times n_d$ submatrices. However, in addition to CPU and memory load-balancing, we also have to consider communication load-balancing, and the naive distribution of just the lower triangle (or equivalently, just the upper triangle) leads to highly imbalanced communication patterns [90]; in particular, because the number of processing units per column (row) ranges from 1 to n_d for different columns (rows).

6.2 Simulation of a Distributed communication

To create a more efficient mapping for load-balanced communication, one can start from the $n_d \times n_d$ square grid of submatrices, and, taking into account that $\hat{\mathbf{H}}$ is symmetric, require that each column (row) in the $n_d \times n_d$ grid has the same number of submatrices (specifically $(n_d+1)/2$) assigned to one of the $n_p = n_d(n_d+1)/2$ processing units. There are

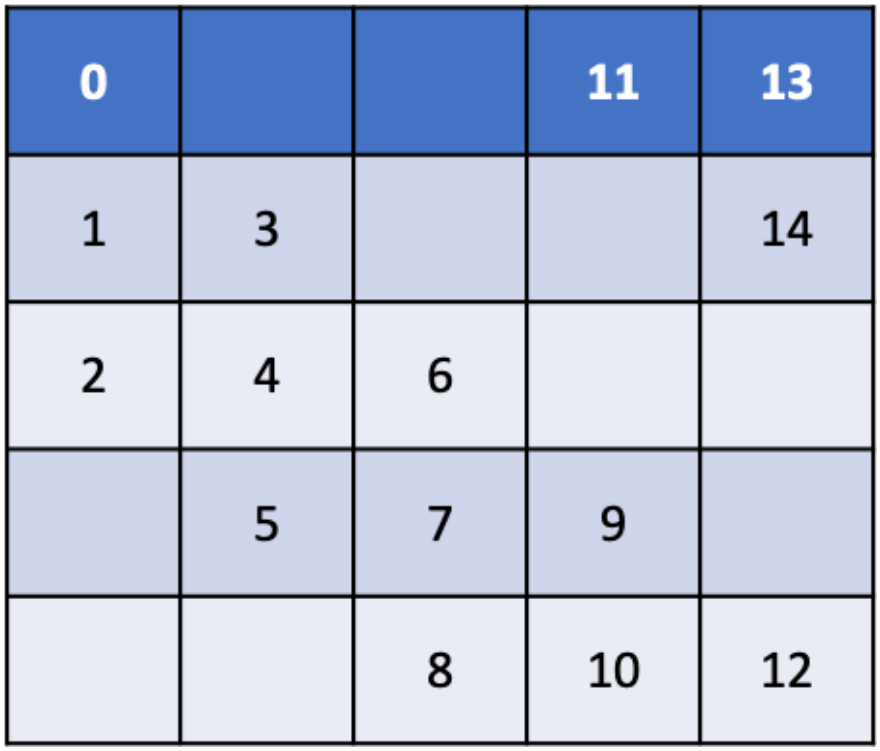


Figure 6.1: Processor topology with 15 processors numbered from 0-15. Distributed in an efficient manner where each row and column has the same number of processors.

many different ways to achieve this – the implementation that is used in MFDn [22, 90], is illustrated in the top panel of Fig. 6.2 for 15 processing units on a 5×5 grid of submatrices. After each local SpMV and SpMV^T, we have to perform two reductions: One along the processing units in the same column, and one along the processing units in the same row, as as indicated by the lower panels of Fig. 6.2. With the mapping of Fig. 6.2, all column- and row-communicator groups contain $(n_d + 1)/2)$ processing units per communicator, and have essentially the same communication volume as well. Thus, with this distribution the communication load associated with the SpMV or SpMM in the iterative solver is almost the same for all processing units and communicator groups, both in message sizes, and in number of processing units in each communicator, provided that the dimensions of each submatrix are approximately the same.

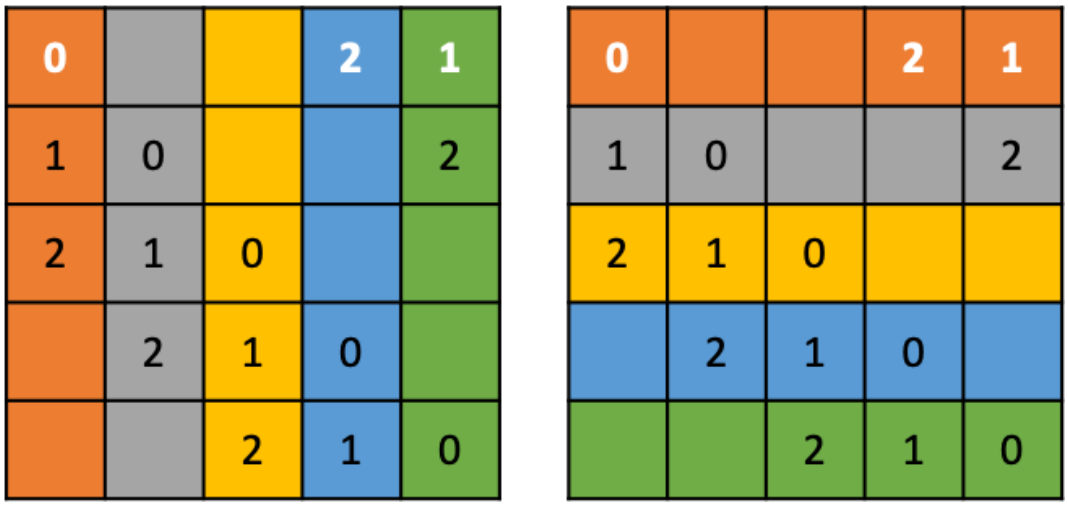


Figure 6.2: Processor distribution in MFDn: MPI_COMM_WORLD (top) and custom column (left) and row (right) communicator groups.

The additional steps in the iterative solvers, namely orthogonalization of (blocks of) vectors, preparation of the input vector (block) for the next iteration, and in the case of LOBPCG applying the pre-conditioner, are all distributed evenly over all n_p processors. For this purpose, we further divide each of the n_d (blocks of) vectors into $(n_d + 1)/2$ segments, and distribute these evenly over the column-communicator groups. Thus each processing unit deals with a (block of) vectors of length $D/(n_d(n_d + 1)/2)$ for the orthogonalization and preparation of the input for the next iteration, where D is the dimension of the Hamiltonian. These steps also involve additional communication, mainly reduce or all_reduce on all processing units, but the message sizes are small and the communication overhead during this step is negligible compared to the communication overhead during the SpMV/SpMM phase.

Schematically, the communication pattern for the SpMV/SpMM phase at each iteration is given in Fig. 6.3:

First, the (block of) vector segments of length $D/(n_d(n_d + 1)/2)$ are gathered on the 'diagonal' processing units within each column-communicator;

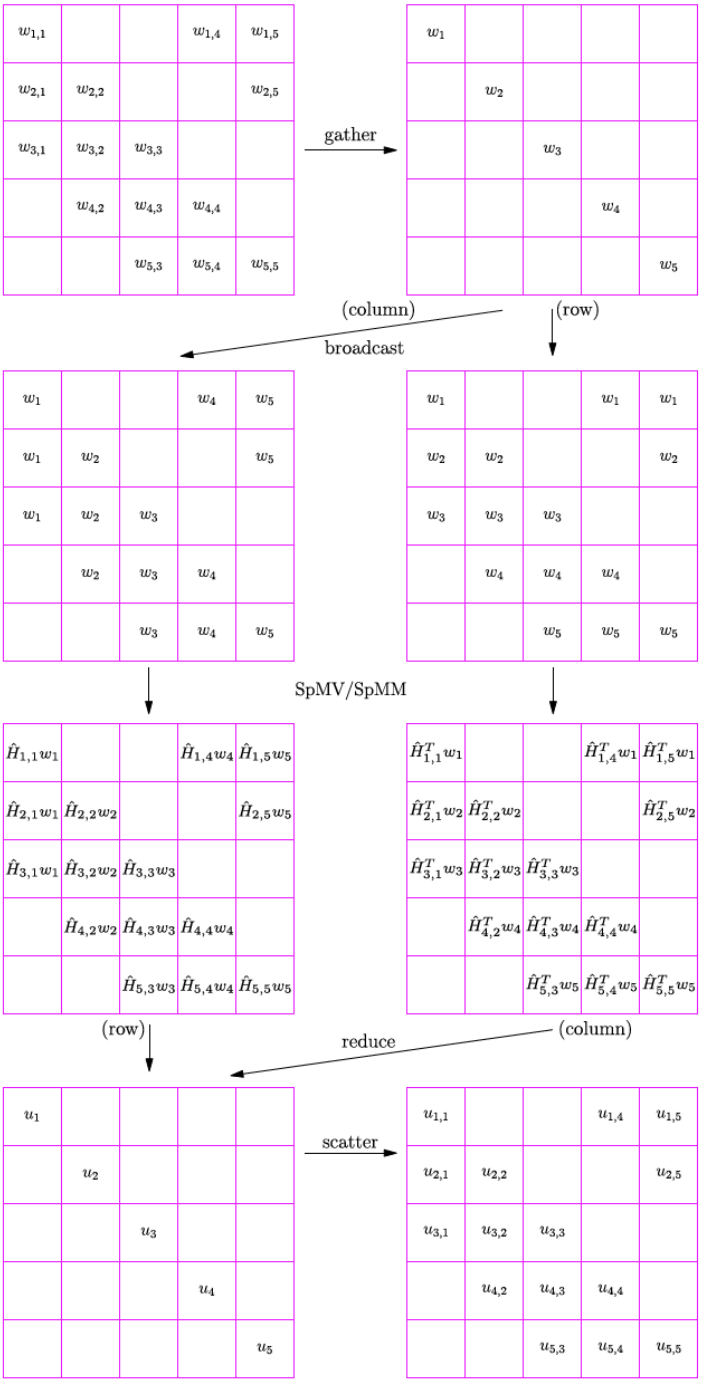


Figure 6.3: Communication pattern for distributed SpMV (Lanczos) or SpMM (LOBPCG) during iterative solver. In our actual implementation, we have replaced the initial Gather + Broadcast along the columns by a single call to AllGatherV, and similarly, the final Reduce + Scatter along the columns by a single call to ReduceScatter. Also, the Broadcast and Reduce along the rows is overlapping with the local SpMV and SpMV$^{\mathrm{T}}$. (Figure adapted from Ref. [4]

Next, the diagonal processors broadcast their (block of) sub-vectors of length D/n_d both along the column-communicators and along the row-communicators;

Each processing unit performs a local SpMV/SpMM and SpMV^T/SpMM^T;

The outputs of both the local SpMV/SpMM and SpMV^T/SpMM^T are reduced along the column and row communicators, respectively, onto the diagonal processors;

Finally, each diagonal processor scatters the final result among the $(n_d + 1)/2)$ processing units within its column-communicator, for further processing in preparation for the next iteration.

In practice, we perform the initial gathering of the vector segments onto the diagonals followed by the broadcast along the column-communicators in a single collective MPI call, namely MPI_AllGatherV. Similarly, the reduction along the column-communicators followed by the final scatter of the vector segments at the end can also be done in a single collective MPI call, namely MPI_ReduceScatter. Thus the entire SpMV/SpMM requires only four collective MPI routines: MPI_AllGatherV, MPI_Bcast, MPI_Reduce, and MPI_ReduceScatter.

It has been shown that this distribution of the data and implementation of the communication performs efficiently on current HPC platforms; furthermore, on multicore processors it allows us to hide the broadcast and reduction along the row-communicators during the SpMV/SpMM phase behind computation [22], depending on the local CPU performance, the actual network performance, and the problem size. However, for large-scale production runs, using thousands of processing units, and with a vector dimension in the tens of billions, the communication overhead does become a major factor, surpassing the time required for the local sparse matrix computations. For an application scientist preparing for the next

generation of HPC platforms, it is therefore very useful to be able to simulate just the communication overhead before the system becomes available so that any algorithm changes or optimizations can be worked out in advance.

6.3 Simulation Framework and Implementation

To evaluate the communication pattern we utilize the Structural Simulation Toolkit [88]. SST is a flexible simulation framework designed to explore trade-offs in the parallel and distributed system design space. SST is widely used within the academic, national laboratory and vendor communities [130, 131, 132]. Components included in SST represent the different subsystems of a supercomputer, including, CPUs, accelerators, memory, networks and software. These components are connected together by the core SST framework which can then simulate discrete events in parallel across multiple nodes using MPI. In this work, we utilize three existing modules within SST, namely Ember, Firefly and Merlin. These represent the workload/communication pattern, communication software stack (e.g., MPI) and the network fabric (routers, cables, topology, etc.), respectively.

6.3.1 Ember

Ember is one of the components of the SST libraries. This is a state-machine based event engine. This event engine replicates communication patterns in a scientific application at a simulation endpoint. Sets of application communication patterns are called *motifs*. A collection of motifs can be created within each point to create a complex workflow.

A sequence of events containing primitive communications and also collective communications, computations, timings, barriers are created in a motif. A queue is maintained

where these events are pushed and executed one by one until the queue is empty. Motifs are prompted to refill the queue with additional events once emptied.

The communication events are sent to the Firefly layer. A communication event is coded in Ember but converted into actual operations by Firefly component which keeps track of the parameters associated with the encoded Ember communication motif code. Ember uses short sprints of events. Ember component can scale to very large simulated node counts regardless of any constraints on the amount of memory or simulation related processing.

6.3.2 FireFly

Firefily is another component in SST which implements a state machine based data movement stack. It is the "MPI" equivalent in SST. The main purpose of Firefly is to help testing different network topologies at a larger scale than actual simulations would be allowed to run on by a network stack. Firefly can not be run stand alone. It requires a network component which in this case is Merlin and a driver component which in our case is Zodiac.

This component provides library supports for point-to-point communications like send, receive, wait etc. It also provides support for the collectives like alltoall, reduce etc operations. It follows an eager/rendezvous protocol model. The state machine functionality moves the data between hosts over a bus. The motifs are written in Ember component and sent to Firefly component where the network parameters along with the message sizes, bandwidth, latencies are processed.

6.3.3 Merlin

Merlin is a combination of low level networking components that be used to simulate high speed networks or on chip networks. Merline comprises of a range of different network topologies. Merlin also provides a set of tunable parameters like buffer size, latencies, routing models which can be used to create a new architecture that is not yet released. Merlin library currently supports DragonFly topology which we have used in our simulation, Torus, Fat Tree topologies. All topologies use deterministic routing.

6.4 Implementation

For comparisons with SST simulations, we implemented a communication only version of MFDn, which we call communications-skeleton code, because for very large matrices where large numbers of compute nodes are needed, communication overheads dominate the execution time of the MFDn eigensolver. Hence, our SST/MFDN motif does not simulate any computations either.

We implemented MFDn's communication motif (without any computations) using SST. We show the pseudocode in Algorithm 4. MPI communication routines implemented in SST/Ember have the following common syntax: $enQ\_ < MPI\_communication\_routine\_name >$. We tried to replicate the actual application code with only communication routines in our MFDn motif, but we could not use the exact Ember/SST equivalent of the MPI routines in a few cases where they lacked support as detailed below.

MFDn uses MPI_Comm_Split for creating its row and column communication groups. However, the corresponding enQ_Comm_Split function was not producing the correct communication groups for our motif, therefore we used the enQ_Comm_Create function which

Algorithm 4: Ember pseudocode for the MFDn communication motif.

```
1 enQ_Comm_Create(..., Comm_world,..., Col_Comm);
2 enQ_Comm_Create(..., Comm_world,..., Row_Comm);
3 for iter = 1 to maxIteration do
4     enQ_Allgather(...,vector_segments,...,Col_comm);
5     enQ_bcast(...,sub_vector,...,Row_comm);
6     enQ_reduce(...,SpMM_output,...,Row_comm);
7     enQ_reduce(...,final_result,...,Col_comm);
8     enQ_scatter(...,final_result,...,Col_comm);
```

creates a custom communicator for given a set of MPI ranks.

MFDn uses MPI_Reduce_Scatter along the column communicator in the final phase as pointed out in Figure 6.3 because MPI_Reduce_Scatter significantly reduces the execution time over using an MPI_Reduce followed by an MPI_Scatter. Unfortunately, SST currently does not have an equivalent enQ_Reduce_Scatter function. Hence, we used an enQ_Reduce followed by an enQ_Scatter in the MFDn motif. We added the simulation time for the reduction and the scatter operations together in all results in the following section where we compare the timings from the SST simulation and the real application. We acknowledge that this will not entirely capture the effect of MPI_Reduce_Scatter in the real runs.

Another important difference between MPI and Ember/SST is that unlike the MPI_Reduce operation, no local aggregations are applied in the enQ_Reduce function. In MFDn, reductions are actually performed in two different ways. One uses the default MPI_SUM aggregation, but this default MPI_SUM option does not make use of all available cores in a hybrid parallel MPI-OpenMP code. Since the dimensions of vectors in MFDn are very large, local aggregations during reductions are potentially very time consuming. Therefore, MFDn uses a customized reduction that uses OMP multithreading for aggregation; we refer to this version as OMP_SUM. For most moderates size problems, the OMP_SUM version outperforms the default MPI_SUM, but for very large number of ranks or if we use 16 or

more ranks per node (leaving only 4 or fewer threads for OMP parallelization), there is almost no difference or the default MPI_SUM is slightly more efficient. This illustrates that the actual reduction operation takes a non-negligible amount of time for the message sizes in MFDn. Since there are no actual summations involved in the Ember/SST reduction, we expect there will be a considerable performance difference between the real application runs and the SST simulation results for reduction operations.

6.4.1 Random Distribution of Processes

During a production run, when we try to allocate nodes from a system using batch scripts, getting nodes in close proximity is generally not guaranteed (unless if one uses the entire machine). In a large supercomputer, typically hundreds of jobs are running, starting and completing at different times. Resources are allocated according to various queuing policies. A set of nodes are created from the available nodes and those are allocated for the next eligible job. Hence, medium to large jobs are generally fragmented in a random manner across the set of available nodes.

In the MFDn code, the process distribution assumes that MPI ranks are in the range from 0 to np as discussed in Section 6.1. When we run the exact process distribution using SST, it assumes that the first np cores/nodes will be used for the simulation and the custom communicators are created accordingly, thus deviating from the random node allocation scheme. Therefore, we have introduced a similar random node selection in the SST simulation code for MFDn.

In Figure 6.4, we show a small example where we need 6 nodes with one MPI ranks per node for our simulation. We assume the machine has 32 nodes in total. When we ask for 6 nodes, a random set of 6 nodes are given. Then in the SST simulations, the custom

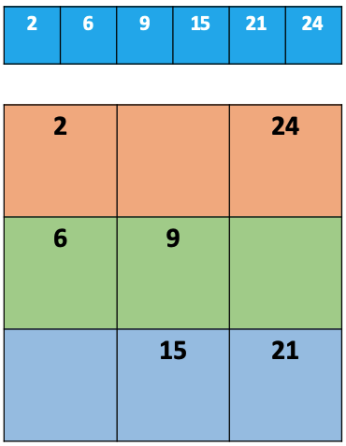


Figure 6.4: Random selection of the ranks.

communicators will be generated accordingly as shown in Figure 6.4. If all MPI ranks in the SST simulation are consecutive and start from 0, then it does not portray a real world scenario and it is possible that it does not catch the actual communication bottlenecks. Introducing this randomness helps us making the SST simulation close to the real world simulations.

Note that when we use more than one MPI rank per node, some of the MPI ranks will be bundled under the same node. We have created the random set keeping this practical issue in mind. Whenever we need more MPI ranks per node for our simulation, first a set of random nodes will be selected. Then the set of MPI ranks will be created using those nodes and the number of MPI ranks bundled together.

6.5 Evaluation and Results

For our experiments, we chose two different clusters. We used the Cori-KNL cluster as an existing machine for validation and another machine similar to the upcoming NERSC Perlmutter system for prediction. Although the detailed configuration information for this

system is not yet public, we use a reasonable approximation.

6.5.1 Hardware and Software

We conducted all our validation experiments on Cori Phase II (Cori-KNL), a Cray XC40 supercomputer at NERSC. Each Cori-KNL node is a single-socket Intel Xeon Phi Processor 7250 ("Knights Landing") processor with 68 cores per node @ 1.4 GHz. Each node has 96 GB DDR4 2400 MHz memory with 102 GiB/s peak bandwidth and also a 16 GB MCDRAM (multi-channel DRAM).

Although not all details of the Perlmutter system are released yet, it is scheduled to be delivered in two phases. Phase 1 will have 12 GPU-accelerated cabinets and 35 PB of all-Flash storage and Phase 2 will have 12 CPU cabinets. Each of Phase 1's GPU-accelerated nodes will have 4 NVIDIA A100 Tensor Core GPUs based on the NVIDIA Ampere GPU architecture, along with 256GB of memory for a total of over 6000 GPUs. In addition, the Phase 1 nodes will each have a single AMD Milan CPU. Each of Phase 2's CPU nodes will have 2 AMD Milan CPUs with 512 GB of memory per node. The system will contain over 3000 CPU-only nodes. For simulating a Perlmutter like machine, we adjusted our parameters accordingly since it will have larger memory.

We use SST_9.1.0 version for all our simulations. While installing SST, we used Open-MPI_4.0.2 version as a supporting flag. For compiling SST, we use NERSC's default programming environment modules, namely PrgEnv-Intel/6.0.5.

Table 6.1: Matrices used in this study, the dimensions and number of nonzero matrix elements of each matrix.

Nucleus	Dimension	Number of Nonzeros
11Be, Nm= 8	196,861,465	146,137,030,364
10Be, Nm= 9	430,062,264	409,045,051,874
10Be, Nm=10	1,343,536,728	1,600,272,603,633

6.5.2 Benchmark problems

For normal production runs of MFDn, the dimension of the matrices ranges from a few hundred to tens of billions; the largest runs to date, on nearly the full Cori-KNL machine, have a dimension of about 35 billion. For practical reasons, we restrict ourselves here to three problem sizes, as listed in Table 6.1. We also list the number of nonzero matrix elements in half of the symmetric matrix – this is what dominates the computational load.

It has been reported already that the wall time to simulate the motifs using SST varies depending on different factors. These factors include congestion, adaptive routing, number of events and distribution of cores across physical nodes. Simple motifs take small wall times. But complicated motifs require a large amount of wall time across a large number of nodes due to limited memory. For example, for the dimension 430,062,264 it took more than 33 hours to finish 2 iterations of our motif using 16 Cori-KNL nodes having 68 cores each. For comparison, a communication skeleton run on MFDn, performing only the communication during the eigensolver, took less than half an hour on 71 nodes for 5 runs on different number of MPI ranks, and 20 iterations per run. That is, an aggregate of about 30 node-hours for the communication skeleton run, compared to 512 node-hours for the SST simulation. Of course, the advantage of the SST simulation is that one can simulate the performance of machines that do not (yet) exist. Because of this issue, we could not compare the cases with larger dimension requiring larger number of nodes. We believe that our motif being complicated

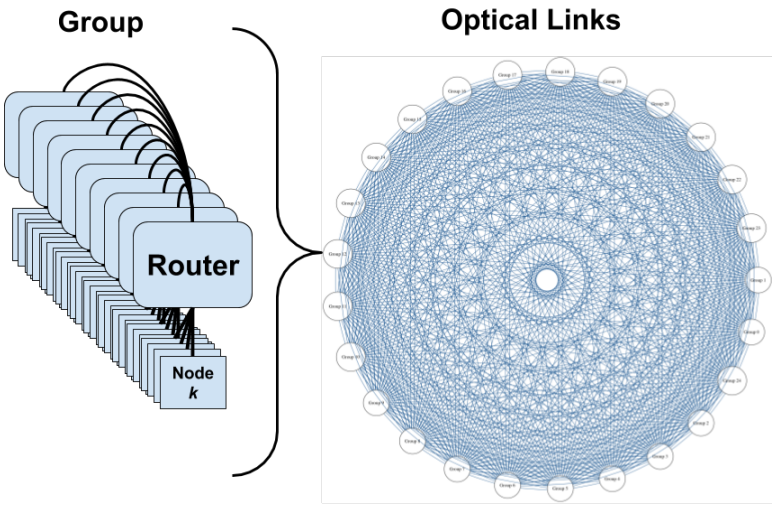


Figure 6.5: Illustration of a general Dragonfly topology with a single group shown on the left and the optical all-to-all connections of each group in a system shown on the right. Per the original definition of a dragonfly [5], the design within a group is not strictly specified.

with different communicators increased the SST walltime compared to more simpler motifs.

6.5.3 SST parameters for Cori-KNL simulations

In SST, there are a number of network topologies that we can use to simulate our motifs. Since both Cori-KNL and Perlmutter are based on the Dragonfly topology [5], we use the Dragonfly option in SST. Dragonfly networks combine high radix routers and create virtual routers called a group which are fully connected to other groups by optical links. Local ports connect a router to a compute node or NIC. Group ports connect routers within the same group together. Global ports facilitate inter-group traffic and use optical links so that they may reach larger distance than is practical for electrical cables. For a visual reference the reader may refer to Fig 6.5. As mentioned before, because of our interest in communication overheads, we have elected to utilize communication-only skeleton code and its SST implementation. Specifically, we utilize SST to accurately represent the packet-level routing, buffering, and internal switch characteristics of the Dragonfly network, as well as

the MPI semantics and message matching.

Below, we specify the Merlin and Firefly parameters we use to simulate Cori-KNL interconnect:

Network Topology: For simulating Cori-KNL System, we use a network having 12056 Nodes. In SST, we need to provide the shape of the network that we want to use as a parameter. We use 4 local ports and 96 as the group port value. To replicate the same peak bisection bandwidth as Cori-KNL, we use 10 global optical links among groups.

Note that we limit our runs to the minimum number of "switches" required. There are $(4 * 96) = 384$ nodes in each Cori-KNL switch. Hence, for example a simulation needing 1000 nodes, we use 3 groups for the corresponding SST simulation.

Router and NIC Parameters: In Table 6.2, we present the router and NIC parameters we used for simulating the Cori-KNL cluster. Each links has a bandwidth of 8 GB/s. The port input, output latency values are taken from the Cray documentation[133], where possible or otherwise estimated (as is the case with input and output buffer sizes).

In practice, there are subtle differences in the real system simulated that Merlin does not currently capture. For example in a real Cray Aries router, traffic from a single 8GBps NIC is divided across 48 router *tiles*. Depending on whether it is an optical tile or electrical tile, the bandwidth may vary between 4.7 and 5.25GBps. Though these architectural subtleties are more complex than can be simulated by Merlin currently, SST is widely used within industry to simulate the performance of large systems and we use parameters that provide as close an approximation as possible to the NERSC Cori network.

NERSC uses SLURM for job scheduling. With the "–switches" flag in SLURM, we can

Table 6.2: Router and NIC Parameters used for Simulating Cori-KNL

Parameter Name	**Value**
flitSize	6 Byte
port input latency	150ns
port output latency	150ns
link latency	150ns
packetSize	64Byte
link BW	8 GB/s
input buffer size	16KB
output buffer size	16KB

limit our runs to a certain number of Dragonfly groups when we need small number of nodes. This significantly reduces the communication time, as the global optical links are not used in these cases. We used the appropriate parameters in SST simulations in accordance with this. We have also implemented the random distribution of processes in SST in such a way that addresses this phenomena and the random set is created in the same range as the production runs. The matrices considered in this study all fit within a switch of 384 KNL nodes.

6.5.4 Simulation results in Cori-KNL

We run the simulations with 3 matrices from Table 6.1. In Cori-KNL the total memory per node is 96GB. To use half the memory of of cori KNL(48 GB) with the sparse matrix, we try to keep the number of nonzeros around 6 billion per node. We use different number of MPI ranks per node (1,2,4,8,16) for both the communication skeleton runs of MFDn and the SST simulations. So in our simulations, for each of the matrices, the nodes required remains similar but with the increased MPI ranks per node, total MPI rank increases, see Table 6.3. The communication volume between ranks also decreases for all MPI communication routines. We show the number of MPI ranks (np), the number of diagonal processors (nd)

Table 6.3: MPI ranks, number of diagonals, number of ranks per custom communicators, Message Size during Broadcast and Reduce and Message Size during Allgather and Reduce_Scatter.

Dimension = 196,861,465

np	nd	$\frac{nd+1}{2}$	Bcast & Reduce	AllGather&ReduceScatter
28	7	4	112 MB	28 MB
45	9	5	88 MB	18 MB
91	13	7	60 MB	8.7 MB
190	19	10	40 MB	4.2 MB
378	27	14	29 MB	2.1 MB

Dimension = 430,062,264

np	nd	$\frac{nd+1}{2}$	Bcast & Reduce	AllGather&ReduceScatter
66	11	6	144 MB	26 MB
120	15	8	116 MB	14 MB
276	23	12	76 MB	6.2 MB
496	31	16	56 MB	3.5 MB
1128	47	24	36 MB	1.5 MB

Dimension = 1,343,536,728

np	nd	$\frac{nd+1}{2}$	Bcast & Reduce	AllGather&ReduceScatter
276	23	12	232 MB	20 MB
496	31	16	172 MB	11 MB
1128	47	24	116 MB	4.8 MB
2016	63	32	84 MB	2.7 MB
4560	95	48	56 MB	1.2 MB

which represents how many column or row communicators will we have, number of processors in each custom communicator ($(nd+1)/2$) in Table 6.3 as well (see also Figs. 6.2 and 6.3 for the different custom communicators and the communication pattern).

In Fig. 6.6, we show the execution times per iteration for the communication skeleton of our application with default MPI_SUM and custom OMP_SUM, and also with the SST simulation. We can see that there is a difference between the real application result and the simulation results. We attribute this difference to a combination of (a) the non-negligible reduction operation which is not present in SST simulation but is included in the commu-

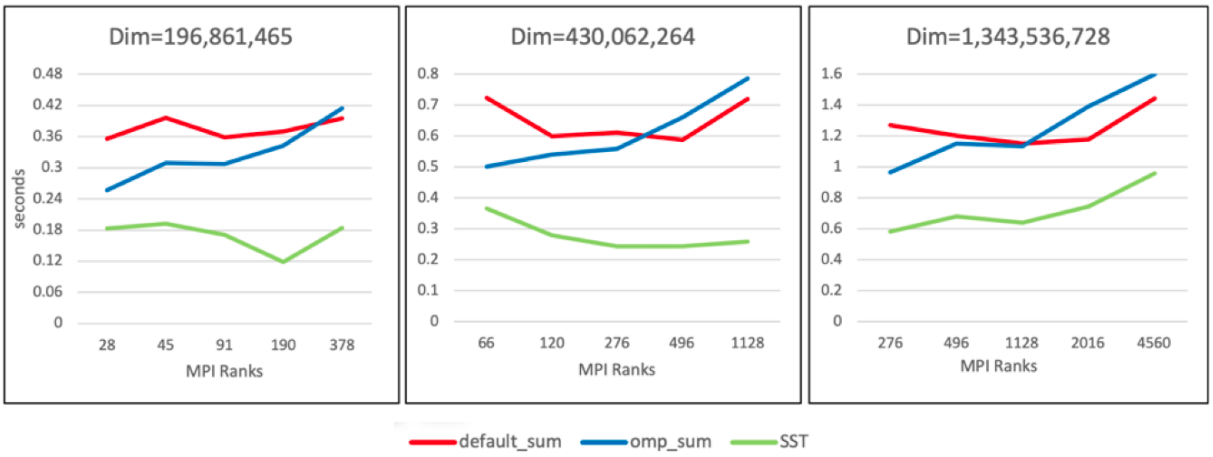


Figure 6.6: Total Execution time per iteration in Cori-KNL for real application using default MPI_SUM, custom OMP_SUM and SST Simulation

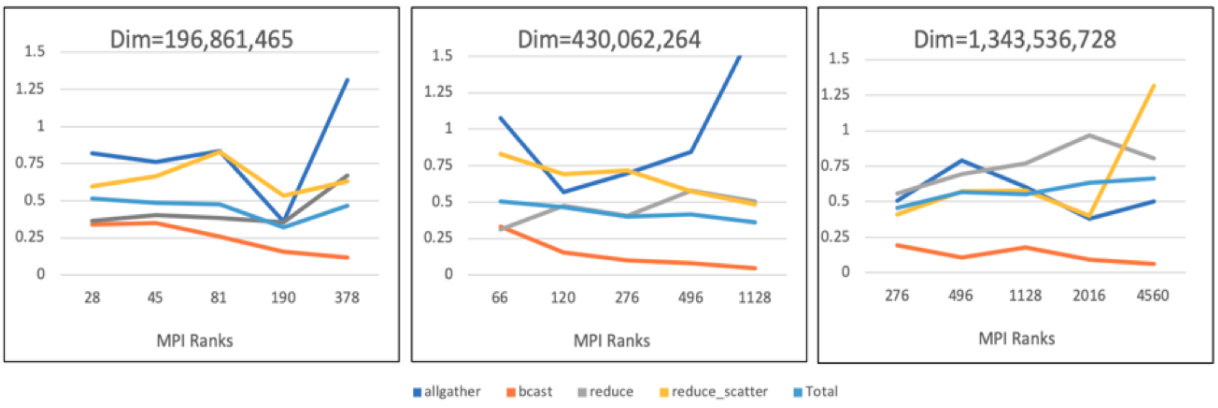


Figure 6.7: Ratio of different MPI communication routines between SST simulation and communication skeleton runs with MPI_SUM. i.e. $\frac{SST\_time}{Real\_run\_time}$

nication skeleton; (b) the lack of an MPI_Reduce_scatter equivalent in the SST implementation; and (c) possible network congestion in the communication skeleton runs due to the communication pattern of the workload.

In Fig. 6.7, we show the ratio of SST simulation results with real application runs for all communication routines in MFDn. As mentioned before, we add the times of the Reduce and the Scatter calls in SST and compare it with the Reduce_Scatter time of the communication-skeleton runs. We can see in all cases that broadcasts in SST are predicted to be much faster than the real runs. Our hypothesis is that this is a result of how the Aries network manages congestion compared to SST Merlin. It has been pointed out in recent work that congestion on the Aries system can severely degrade performance in ways that the Merlin does not capture. For example in a real system if congestion reaches threshold network quiesce operations may occur that are not captured in SST simulations. In the severe cases the slowdowns on real Cray XC systems have resulted in a 99% reduction in bandwidth [134].

Network congestion on real systems vs simulation: The message sizes and communication patterns of our motif mean that the motif is largely bisection bandwidth bound. To better understand the differences between simulated and real network congestion, we created a simple benchmark to communicate across the bisection bandwidth of the real Cori network and compare the congested vs peak performance. This benchmark divides the network into two partitions and then creates pairs of nodes between pairs of groups, such that no node-pair share a group or partition. Each pair of nodes then measures achievable bandwidth. We ran this across the entire Haswell and KNL partition of Cori during a maintenance window and observed the following: Peak bandwidth between any set of nodes was 3893 MBps, while average bandwidth was 621 MBps showing a roughly 6X difference in bandwidth.

This suggest router ports on NERSC Cori were spending approximately 84% of time stalled on average for the evaluation. The factor of slowdown due to congestion is very similar to what we observe in the difference between SST simulated and real broadcast operations in Figure 6.8.

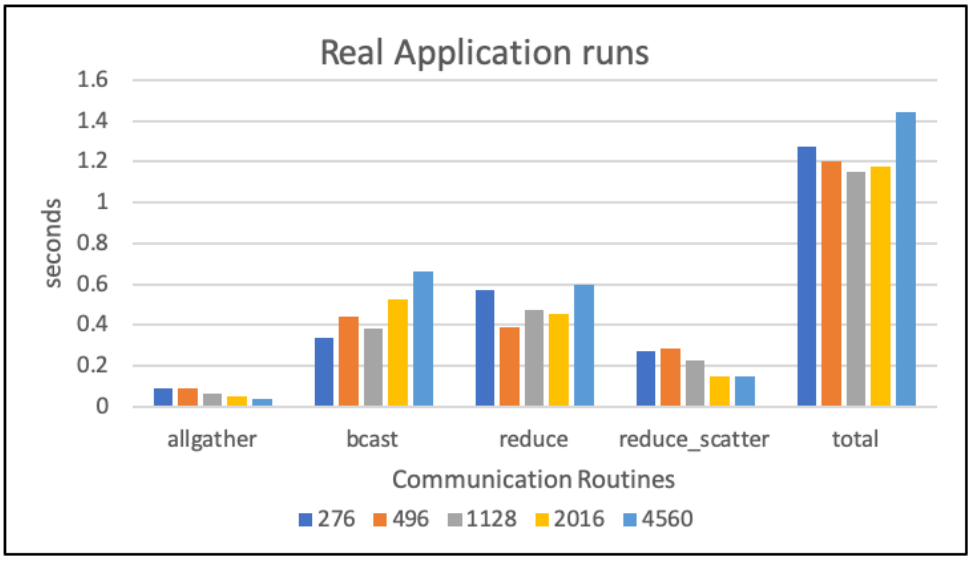
Existing work has examined the impact of congestion in similar bisection bound motifs (3DFFT and AllPingPong) for dragonfly topologies simulated in SST [130]. In that work the peak percentage of time that router ports were stalled was approximately 20%. This analysis suggest that there is a gap between how congestion is simulated in SST and production systems. Improving models of congestion within SST would be valuable to future studies.

6.6 Simulation for A Future Network

We also used SST to simulate the communication motifs on the Perlmutter machine which will be installed at NERSC later this year. Below, we specify the interconnect parameters that we anticipate for this system.

Network Topology: Perlmutter will have a Dragonfly topology like Cori-KNL. However, for this network, we assume the network to have 16 local ports and 32 group ports, and we assume the number of global links among groups to be 4. As in the Cori-KNL results, for the simulations requiring less than $(16 * 32) = 512$ nodes, runs are limited to one group and likewise.

Memory per node: We assume this new machine to have a much larger memory per node value, likely 4 times the memory of Cori-KNL nodes. For the same simulations that we used in Cori-KNL, we reduce the required nodes by 1/4th and increase the MPI Ranks per core to 4 times the values we used in Cori.



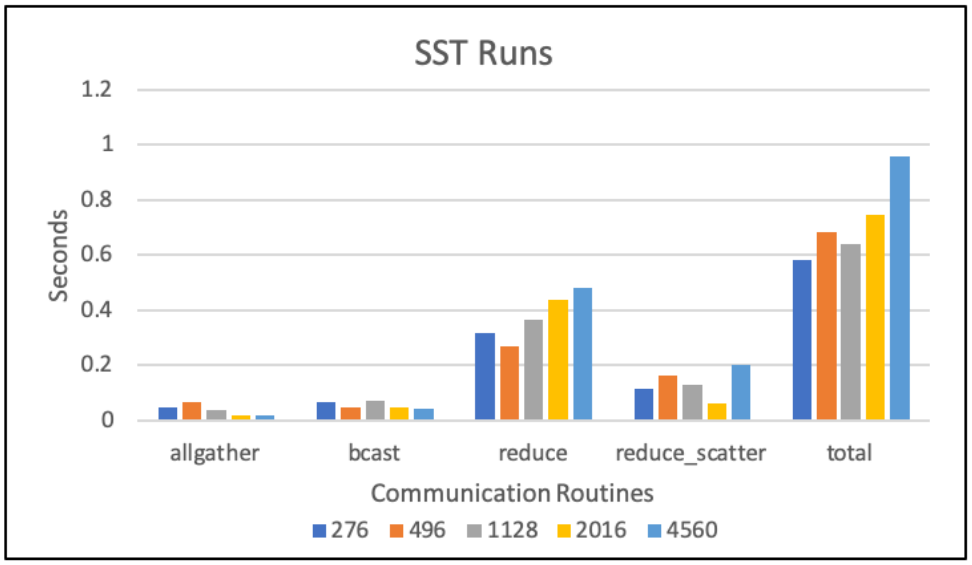


Figure 6.8: Communication time breakdown for a real run and SST simulation for dimension = 1,343,536,728

Table 6.4: Router and NIC Parameter used for simulating Perlmutter's predicted network.

Parameter Name	Value
flitSize	6 Byte
port input latency	150ns
port output latency	150ns
link latency	150ns
packetSize	64Byte
link BW	25 GB/s
input buffer size	64KB
output buffer size	64KB

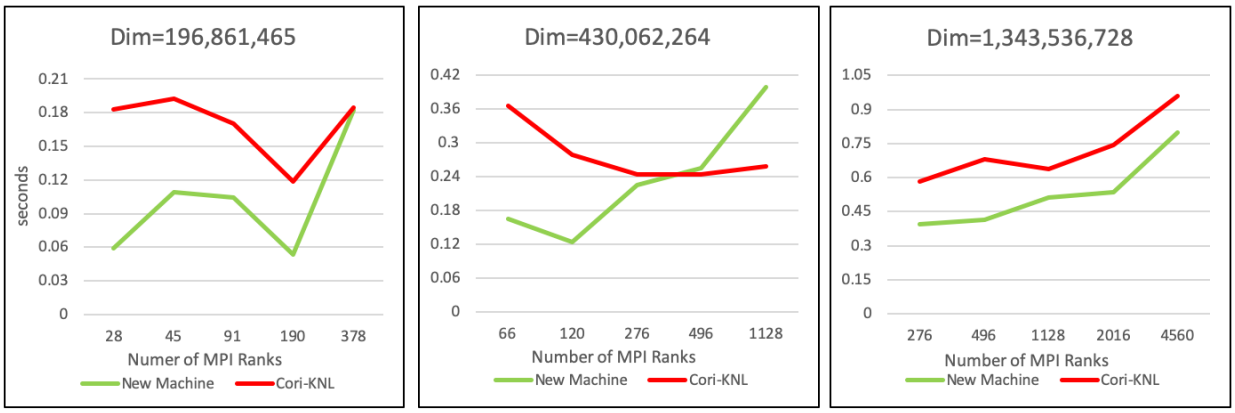


Figure 6.9: Timing comparison of the simulation of MFDn motif in Cori-KNL and the soon-to-be-installed Perlmutter machine with our predicted parameters.

Router and NIC Parameters: In Table 6.4, we give the router and NIC parameters we used for simulating Perlmutter using SST. We assume the network to have a link bandwidth of 25GB/s. We assume the input/output buffer sizes for this new system to be 4 times as those of Aries to account for the increase in the bandwidth-delay-product.

In Figure 6.9, we show the results that we obtained using the parameters we predict for the soon-to-be-installed Perlmutter system and compare them with the SST simulations on Cori-KNL. Since we assumed the bandwidth to be more in the new machine along with high in node memory and increased buffer size, we predict that the execution time in the future machine will bring communication overhead improvements over Cori-KNL in most cases.

Expected impact of congestion in future systems: Network congestion has received greater attention in the design of future networks such as the Cray Slingshot system. Existing work has shown that the impact of congestion is significantly reduced on Slingshot networks compared to the Cori Aries network. Because of this, we expect the SST simulated performance of Perlmutter to be a closer match to the real performance than was observed in Figure 6.7.

6.6.1 Conclusions of this work

In this work, we introduced a new application motif which corresponds to the communication operations in the distributed eigensolver algorithm used in the MFDn code. We compare the simulations of the SST motif with actual runs of a skeleton code written only using communication routines for an existing architecture for validation. We point out to the differences between real runs and simulation results, and gave possible reasons behind those differences. We also evaluated our motif in a future architecture and compare its results with the existing architecture. We also discussed the features we used and some shortcomings of SST. Moreover we also have contributed in introducing new ember motifs by the developers in the SST open source community. With these observations we got a better idea on how to go ahead and implement the distributed MFDn using communication tasks which we will decide in the next chapter.

Chapter 7

OPTIMIZING A DISTRIBUTED MEMORY APPLICATION USING DEEPSPARSE

7.1 Motivation

In the quest for a task parallel implementation of an iterative eigensolver, we first dived into the shared memory architecture. We developed the framework DeepSparse which we described in Chapter 4. In that work we implemented two different algorithms Lanczos and LOBPCG algorithms used executed them using our DeepSparse framework. The implementation is based on task parallelism and was an on node optimization. We observed that DeepSparse achieves 2× - 16× fewer cache misses across different cache layers (L1, L2 and L3) over implementations of the same solvers based on optimized library function calls. We also achieve 2× - 3.9× improvement in execution time when using DeepSparse over the same library versions.

In the shared memory implementation of DeepSparse, we only have computational kernels. In all the kernels we used, there was some kind of computation/assignment operations. Looking at the success of the framework in a shared memory architecture, we were moti-

vated to extend the framework into a distributed memory architecture and extending the framework into an actual scientific application. With the experience on working with MFDn code which is a distributed memory CI code I decided to take this application to build a distributed memory version for DeepSparse.

Our target application, MFDn [89, 90, 4] is used for ab initio calculations of the structure of atomic nuclei. The structure of an atomic nucleus with A nucleons can be described by solutions of the many-body Schrödinger equation

$$\hat{\mathbf{H}}\,\Psi(\vec{r}_1,\ldots,\vec{r}_A) \;=\; E\,\Psi(\vec{r}_1,\ldots,\vec{r}_A) \tag{7.1}$$

where $\hat{\mathbf{H}}$ is the nuclear Hamiltonian acting on the A-body wavefunction $\Psi(\vec{r}_1,\ldots,\vec{r}_A)$, $\vec{r}_j$ the single-nucleon coordinates, and E the energy. This code already consists of the LOBPCG algorithm that we explored and the Sparse Matrix Multiple Vector Multiplication(SpMM) being the main kernel which is also optimized by us in a prior project motivated us to use this application.

7.1.1 Introducing communication tasks

As we discussed in the previous Chapter 6 about the communication pattern in MFDn while doing the distributed matrix multiplication. We noticed that we have an allgather, a broadcast, a reduction and a reduce scatter operation in the MFDn code. The detailed explanation is given in Section 6.2 in Chapter 6. In practice, it is seen that for MFDn, when run in an architecture like knl, the communication usually takes over the computation for a very large simulation consisting of a very high number of mpi ranks involved from a lot of compute nodes. We simulated the performance of the communication patterns and tried to

find out a possible cause using a simulator named SST.

We observed that, the broadcast operation takes a much longer time in real life because of possible network congestion and the messages being very large in size also fuels into this behavior. Since our deepsparse framework is a task based parallel framework where each task performs a particular matrix or vector operation on a matrix or a vector block, we were motivated to use blocked communication tasks.

Our motivation was to introduce custom communication tasks where each communication tasks will communicate with other nodes and only transmit a block of the matrix or a vector between themselves. This will help us in multiple ways.

Reduced Message Size: Being blocked communication will reduce the size of the messages during the communication much less than the actual code which we expected will help in case of network congestion.

Overlapping Communication and Computation: The other motivation was to overlap the communication with the computations in the matrix multiplication. Whenever a particular block is is received or ready to compute, the other kernels waiting for this particular block of matrix or a vector can start immediately rather than waiting for the entire matrix or vector to be transmitted and then starting its execution.

7.1.2 Better pipelining of matrix and vector operations

In the shared memory implementation of DeepSparse we observed a nice pipelined execution of different kinds of kernels. The matrix multiplication SpMM and the vector operations like vector vector multiplication or vector vector transpose multiplication. in figure 4.8 we can

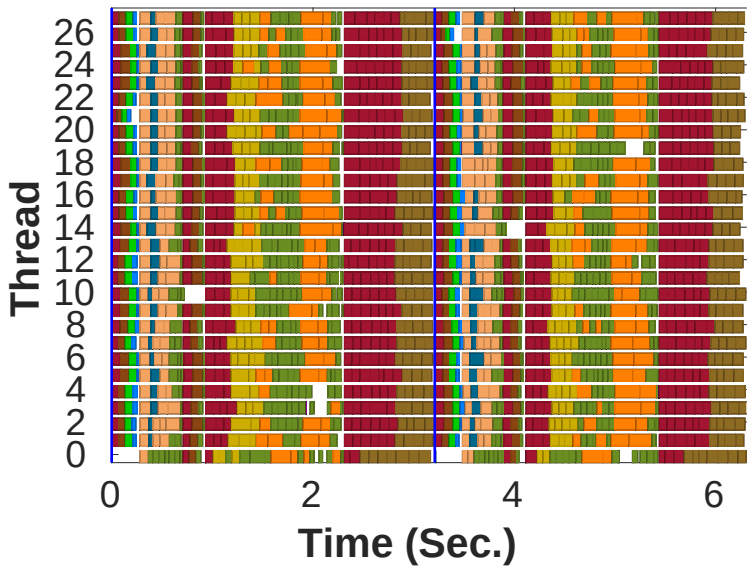


Figure 7.1: LOBPCG two iteration execution flow graph of nlpkkt240 matrix. SpMM is represented using orange color, XY operation is Maroon and XTY is using green color palette

see a pipelined execution of an actual iteration of LOBPCG where the tasks are different kernels but they use the same datastructure and ultimately improves the performance. This test was done in a haswell architecture.

We also did similar tests on Broadwell machines with a different matrix to validate the framework and the pipelined execution. In figure 7.1 we show this pipelined execution for the nlpkkt240 matrix in a broadwell architecture. Here the SpMM is represented using orange color, XY operation is Maroon and XTY is using green color palette. We can clearly observe

that the matrix and vector operations are well pipelined. Another interesting observation was that the ration of time spent on the SpMM and vector operations are somewhat in the similar range. We observed that since the matrix and vector operations are taking similar amount of time during an iteration, a well pipelined execution of these kernels will improve the cache performance which it did and we saw the execution time is actually improved in a shared memory architecture. We were motivated to extend this idea for a distributed application like MFDn which also has similar matrix and vector operations. With the introduction of communication tasks, we were motivated to use the idea from shared memory to distributed memory.

7.2 Methodology

In our previous works we implemented our custom kernels for a task parallel implementation of LOBPCG and Lanczos algorithms for a shared memory architecture. I also worked on the actual MFDn code which is a distributed memory application. For this work the goal was to add communication tasks since other kernels were already implemented.

Using MPI routines as OpenMP task is not a very common practice. There are a few reasons behind this. This is a tricky task, as features of both languages might easily interact in an unexpected way, resulting in dead-locks, incorrect results, fatal errors or performance issues. IntertWine project from Barcelona supercomputing center have been trying to merge MPI routines with their OmpSs programming model[135]. OmpSs uses a task based parallel programming model which is thoroughly similar to OpenMP. They also report some pitfalls of using MPI with OpenMP.

MPI programs start with a function call which initializes the message passing library.

MPI standard defines two different functions for this purpose:

MPI_Init(), used when no multi-threading support is needed.

MPI_Init_thread(), specifies a desired level of multi-threading support.

These routines must be called by one thread only. That thread is called the main thread and must be the thread that calls MPI_Finalize(). The programmer must provide a desired level of multi-threading support to the MPI_Init_thread() which, in turn, may return a value lower than requested. This is because different library implementations may be restricted to different levels (e.g. absence of locking mechanisms for efficiency in single-threaded programs).

We used MPI_Init_Thread with MPI_THREAD_MULTIPLE multithreaded level value as multiple threads may call MPI, with no restrictions if we use this value.

7.2.1 Issue with blocking MPI calls

An important pitfall is using blocking MPI routines as OpenMP tasks.In figure 7.2 we show a small example of this case. Figure 7.2 shows a case where a specific order of task scheduling can produce the single thread execution to hang: all processes execute task C first, making the thread wait for a message that is never sent. This thread enters the MPI routine and cannot leave it until the communication is completed. Thus a deadlock is created.

Using non blocking MPI calls we can get rid of these kinds of scenario. In figure 7.3 we show the kind of call that we will make inside an OpenMP task. To stop the thread still waiting inside the task, we can suspend the execution of this task and allow other tasks using the taskyield directive.

```
// Task A
#pragma omp task out(a) priority(1)
a = 10;

// Task B
#pragma omp task in(a) priority(1)
MPI_Send(&a, 1, MPI_INT, buddy, 0/*tag*/, MPI_COMM_WORLD);

// Task C
#pragma omp task out(b) priority(2)
MPI_Recv( &a, 1, MPI_INT, buddy, 0/*tag*/, MPI_COMM_WORLD,
         MPI_STATUS_IGNORE );

#pragma omp taskwait
```

Figure 7.2: Example code for a blocking mpi call as an OpenMP task

```
int flag = 0;
MPI_Request req;
MPI_Irecv( ..., &req );
while( flag == 0 ) {
  MPI_Test( &req, &flag,
            MPI_STATUS_IGNORE );
}
```

Figure 7.3: Example code for a non blocking mpi call

Although this way helps us to get rid of deadlocks but there is a caveat. In this technique modification of every single MPI routine of an application is cumbersome. Also, tasks can be resumed even though the requests they are waiting for have not been completed yet. Hence, they are resumed to check the completion of the communication and this is not efficient because there is chance that they may be suspended again.

7.2.2 Issue with absence of TAG fields in MPI collectives

Another known problem is the absence of TAG field in MPI Colletive routines. I tried to use collectives like MPI_Ibcast and MPI_Ireduce inside OpenMP tasks but eventually deadlock and segmentation fault occurred because the individual tasks need to know exactly which message it is either sending or receiving with a specific tag. Hence I had to use point to

point MPI routines like MPI_Isend and MPI_Irecv for communications.

Here we are working with blocks of the vectors. During a collective call either all the processes sends some messages to a specific destination process or the root processor send some messages to all other process. The blocks in our vectors are not uniform in size. One thread might send a particular block to the destination and another thread might send another block in separate tasks. In the destination process, the threads in that process are waiting in receive tasks. Without a proper tag field, the destination will start receiving the message but because of nonuniform size of the blocks, it creates a size mismatch and eventually creates a segmentation fault.

This is where I had to use point to point sends and receives instead of collective routines. In the point to point send and receive routines the TAG field is set as the starting offset of the block. The receive routine also expects the message with this particular tag and size. Hence the size and offset of the messages match and the operation completes as expected.

7.2.3 Distributed SpMM

We use an efficient mapping for load-balanced communication. For this, we start from the $n_d \times n_d$ square grid of submatrices, and, taking into account that $\hat{\mathbf{H}}$ is symmetric, require that each column (row) in the $n_d \times n_d$ grid has the same number of submatrices (specifically $(n_d+1)/2$) assigned to one of the $n_p = n_d(n_d+1)/2$ processing units. There are many different ways to achieve this – the implementation that is used in MFDn [22, 90], is illustrated in the top panel of Fig. 7.4 for 15 processing units on a 5×5 grid of submatrices. After each local SpMV and SpMV^T, we have to perform two reductions: One along the processing units in the same column, and one along the processing units in the same row, as as indicated by the lower panels of Fig. 6.2.

With the mapping of Fig. 7.4, all column- and row-communicator groups contain $(n_d + 1)/2)$ processing units per communicator, and have essentially the same communication volume as well. Thus, with this distribution the communication load associated with the SpMV or SpMM in the iterative solver is almost the same for all processing units and communicator groups, both in message sizes, and in number of processing units in each communicator, provided that the dimensions of each submatrix are approximately the same.

The additional steps in the iterative solvers, namely orthogonalization of (blocks of) vectors, preparation of the input vector (block) for the next iteration, and in the case of LOBPCG applying the pre-conditioner, are all distributed evenly over all n_p processors. For this purpose, we further divide each of the n_d (blocks of) vectors into $(n_d + 1)/2$ segments, and distribute these evenly over the column-communicator groups. Thus each processing unit deals with a (block of) vectors of length $D/(n_d(n_d+1)/2)$ for the orthogonalization and preparation of the input for the next iteration, where D is the dimension of the Hamiltonian. These steps also involve additional communication, mainly reduce or all_reduce on all processing units, but the message sizes are small and the communication overhead during this step is negligible compared to the communication overhead during the SpMV/SpMM phase. In the right side of figure 7.4 we show how the vectors are also divided into subvectors and saved in each processors.

7.2.4 Blocked communication

In figure 6.3 we showed the nature of communication done during the SpMM and how the local SpMM and the transpose version of it are executed and also how the results are accumulated eventually. Our goal is to make these communications also blocked.

Since the SpMM and SpMMT both are executed according to the blocksize of the matrix

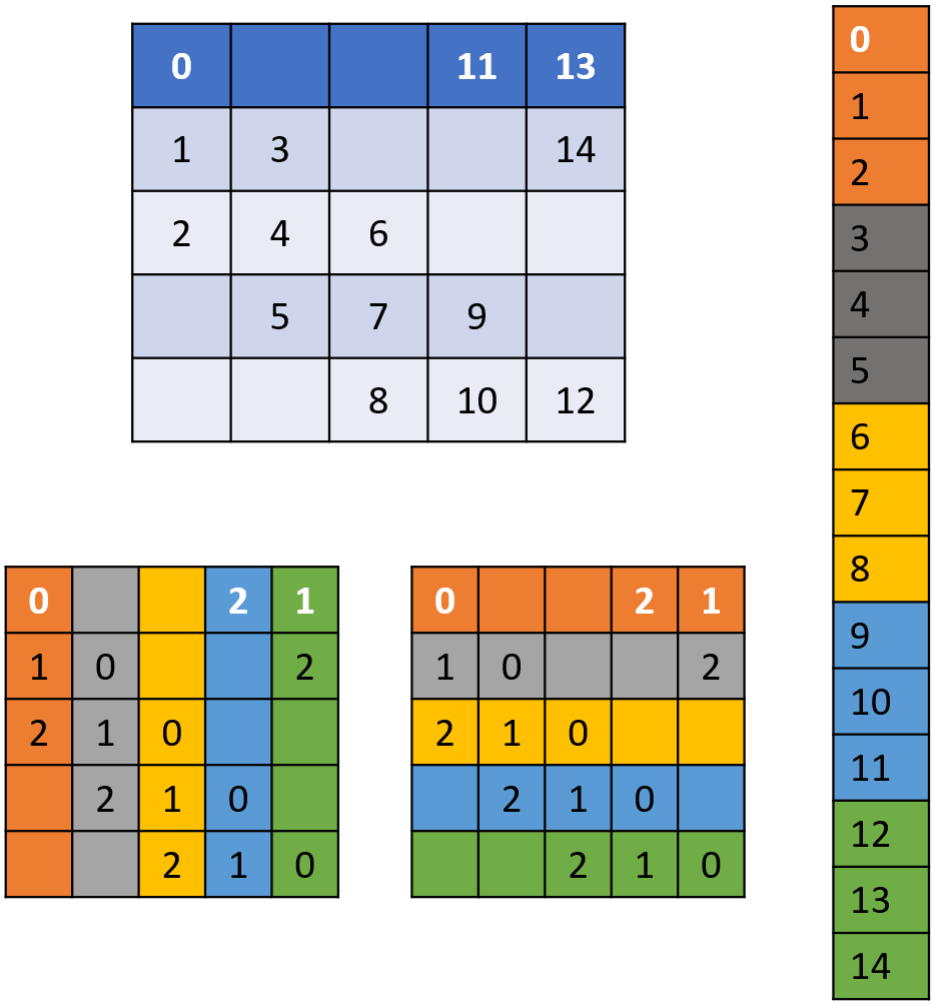


Figure 7.4: Matrix and Vector distribution in MPI ranks and efficient processor topology

blocks. The offsets of these blocks are kept as a metadata which is used during the multiplications to calculate correct addresses. This block size is not uniform. Since our SpMM kernel has a specific OpenMP input output dependencies based on these blockoffsets and custom blocksizes, the same offsets and sizes need to be used when we introduce the communication tasks.

In figure 7.5 we show how do we implement the blocked communication tasks. Since our vectors are divided mpi ranks accross the column communicator, we need to do an allgather accross the column communicator and gather the subvectors in the diagonal processor. Then we do the Bcast along the row communicator. The offsets and sizes being exactly same among the processors, I use the starting offset as the TAG field for the point to point communication routine which helps us to get rid of any possible deadlock or message size mismatch which might result in a segmentation fault.

In figure 7.5 we can see that whenever the communication is done for a particular block, we can immediately execute a local SpMM or a local SpMMT on that block since we have already received that block and now it is data safe to execute an SpMM kernel.

After the local SpMM is finished, it is safe to execute the reduction tasks. Hence, whenever an SpMM task is completed, the reduction task which is depending on the output of this task is called(or expected to get called by OpenMP).

First I implemented a version where I used each individual blocks for a task. But after our implementation, we noticed that a lot of communication tasks are generated and the performance is poor. Hence, I implemented a hierarchical blocking of tasks. In figure 7.6 I show how did we divide the entire matrix into a block of sparse matrix blocks. In this example I show a sparse matrix with 16X16 blocks. We use a higher level block consisting of 4x4 blocks of the original block and use this as our offsets and sizes for the kernel OpenMP dependencies.

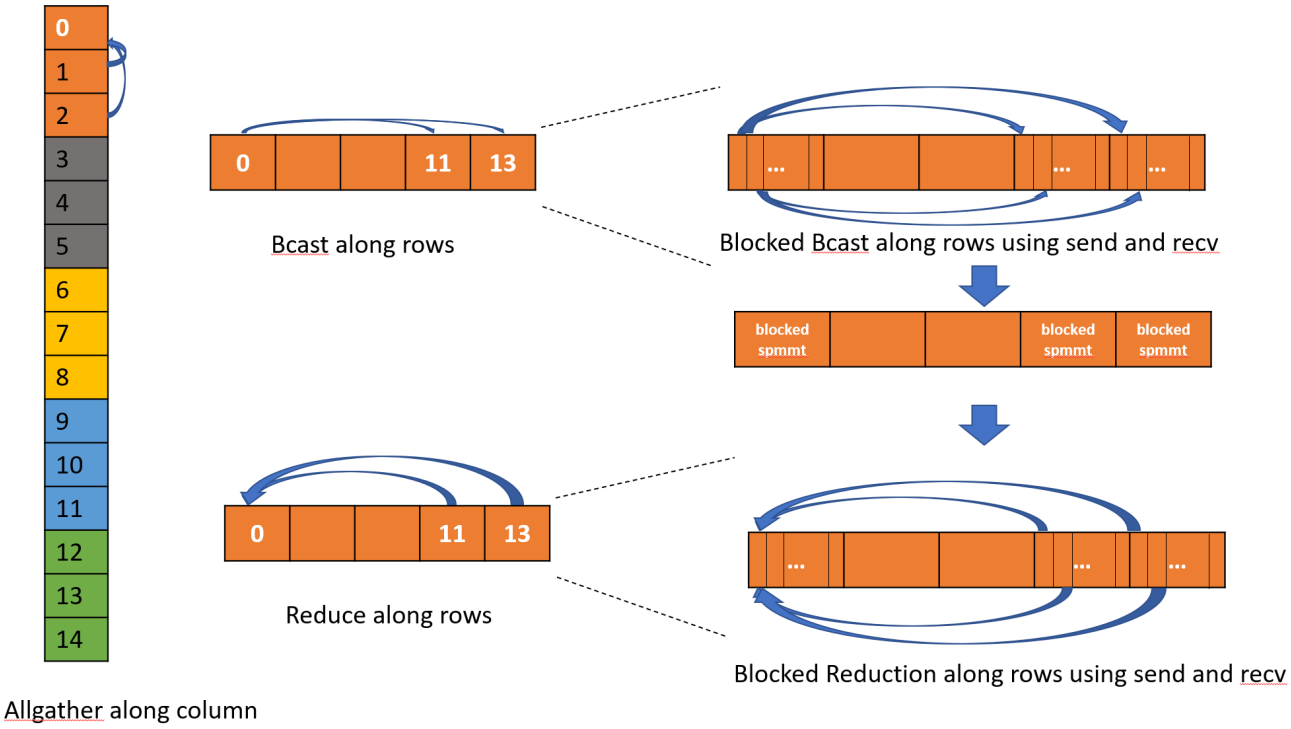


Figure 7.5: Blocked Communication along the processes in the same row communicator

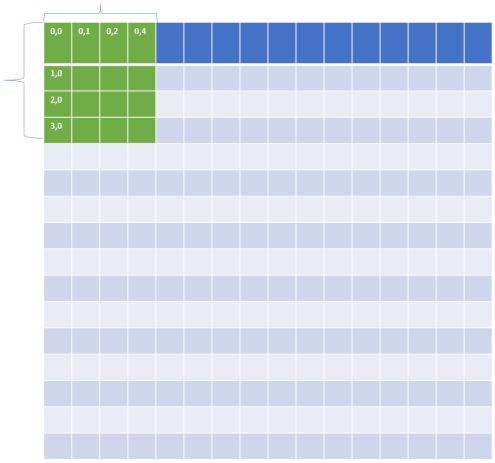


Figure 7.6: Hierarchical blocked communication

```
        if(myRowrank <= 0){
            for(send_rank = 1 ; send_rank < rowCommsize; send_rank++)
            {
                //fprintf(stderr, "rank %d send rank %d rowCommsize %d\n",world_rank,send_rank,rowCommsize);
                for(ib = 0 ; ib < *nrowblks ; ib += taskblk)
                {
                    ilim = (ib+taskblk)>*nrowblks?*nrowblks:ib+taskblk ;
                    isize = (rowblkoffset[ilim]-rowblkoffset[ib])*8 ;
#pragma omp task firstprivate(ilim,isize,taskblk,ib,send_rank) private(flag) shared(reqs_row_comm)\
                    depend(in: ampT[rowblkoffset[ib]*8:isize])
                    {
                        flag = 0 ;
                        MPI_Isend(&ampT[rowblkoffset[ib]*8] , isize , MPI_FLOAT , send_rank , ib , row_comm_c , &reqs_row_comm[ib/taskblk]);
                        MPI_Test(&reqs_row_comm[ib/taskblk],&flag , MPI_STATUS_IGNORE) ;
                        while(flag == 0 ){
#pragma omp taskyield
                            MPI_Test(&reqs_row_comm[ib/taskblk],&flag , MPI_STATUS_IGNORE) ;

                        }

                    }


                }

            }
        }
        else{
            for(ib = 0 ; ib < *nrowblks ; ib += taskblk)
            {
                ilim = (ib+taskblk)>*nrowblks?*nrowblks:ib+taskblk ;
                isize = (rowblkoffset[ilim]-rowblkoffset[ib])*8 ;
#pragma omp task firstprivate(ilim,isize,taskblk,ib) shared(reqs_row_comm)\
                depend(in: ampT[rowblkoffset[ib]*8:isize])
                {
                    //fprintf(stderr,"bcast recv rank %d tag %d\n",world_rank,ib);
                    MPI_Recv(&ampT[rowblkoffset[ib]*8] , isize , MPI_FLOAT , 0 , ib , row_comm_c , &status_row_comm[ib/taskblk]);

                }


            }



        }
```

Figure 7.7: Blocked broadcast code

Doing this way reduced the high number of communication tasks and eventually improved the overall performance. For our runs, a task block size of 8 or 16 is used because they performed the best.

The blocked broadcast is shown in the code snippet 7.7. Since we only use MPI point to point non blocking routines, the root process sends the block to all the processes serially. All the other processes receive that message from the root process with the same TAG that the message is sent with.

In the code snippet 7.8 the task dependencies are shown for the SpMM code. Here it can be seen that we use the same kind of memory footprint for the OpenMP tasks. These dependencies need to be exactly coherent among the tasks otherwise there will be deadlocks or race conditions might not give us the correct result. For example, if a dependency for an

```
        for(ib = 0 ; ib < *nrowblks ; ib += taskblk)
        {
            for(jb = 0 ; jb < *ncolblks ; jb += taskblk)
            {
                ilim = (ib+taskblk)>*nrowblks?*nrowblks:ib+taskblk ;
                jlim = (jb+taskblk)>*ncolblks?*ncolblks:jb+taskblk ;
                isize = (rowblkoffset[ilim]-rowblkoffset[ib])*8 ;
                jsize = (colblkoffset[jlim]-colblkoffset[jb])*8 ;
#pragma omp task firstprivate(ib,jb,taskblk,isize,jsize)\
                private(i,j,k,k1,k2,m,r,c,xcoeff)\
                depend(in: Hamp[rowblkoffset[ib]*8:isize])\
                depend(in: amp[colblkoffset[jb]*8:jsize])
                {
```

Figure 7.8: Blocked SpMM code

MPI_Recv task is not coherent with its prior dependencies, this task will get pulled from the task pool and OpenMP will try to execute that task regardless of an actual send has been processed for the recv or not. This results in a deadlock. Resolving the dependencies appropriately was one of the most important challenges during this implementation. An example code snippet for reduction is shown in figure 7.9 and it can be clearly seen that the same dependencies are kept coherent with the SpMM kernels. Although since it is a reduction operation and the values need to be added with the values of the root processor, a buffer needs to be kept to receive the results from other processors.

When all the OpenMP dependencies are resolved appropriately, the result matches and the MFDn code converges. OpenMP looks at the dependencies of each individual tasks and pulls the tasks whose dependencies have been resolved from a task pool and creates its own DAG underneath and its own scheduler.

7.2.5 Custom reduction

After the SpMM and SpMMt have been executed, then their results are reduced and then scattered in the individual processors for the next iteration. This is done using a MPI_Reduce_scatter call in the actual MFDn code which results in a better performance. The displacements while

```
        if(myRowrank > 0){
            for(ib = 0 ; ib < *nrowblks ; ib += taskblk)
            {
                ilim = (ib+taskblk)>*nrowblks?*nrowblks:ib+taskblk ;
                isize = (rowblkoffset[ilim]-rowblkoffset[ib])*8 ;
#pragma omp task firstprivate(ilim,isize,taskblk,ib) shared(reqs_row_comm)\
                depend(inout: Hamp[rowblkoffset[ib]*8:isize])
                {
                    MPI_Isend(&Hamp[rowblkoffset[ib]*8] , isize , MPI_FLOAT , 0 , ib , row_comm_c , &reqs_row_comm[ib/taskblk]);
                }


            }


        }
        else{
            for(recv_rank = 1 ; recv_rank < rowCommsize ; recv_rank++)
            {
                //fprintf(stderr, "rank %d recv rank %d rowCommsize %d\n",world_rank,recv_rank,rowCommsize);
                for(ib = 0 ; ib < *nrowblks ; ib += taskblk)
                {
                    ilim = (ib+taskblk)>*nrowblks?*nrowblks:ib+taskblk ;
                    isize = (rowblkoffset[ilim]-rowblkoffset[ib])*8 ;
#pragma omp task firstprivate(ilim,isize,taskblk,ib,recv_rank) private(i,j)shared(status_row_comm)\
                    depend(inout: hampBuffer[rowblkoffset[ib]*8:isize])
                    //depend(in: Hamp[rowblkoffset[ib]*8:isize])
                    {
                        MPI_Recv(&hampBuffer[rowblkoffset[ib]*8] , isize , MPI_FLOAT , recv_rank , ib , row_comm_c , &status_row_comm[ib/taskblk]);

                        for(i = rowblkoffset[ib] ; i < rowblkoffset[ilim] ; i++){
#pragma omp simd
                            for(j = 0 ; j < 8 ; j++){
                                Hamp[i*8+j] = Hamp[i*8+j] + hampBuffer[i*8+j];
                            }
                        }
                    }


                }

            }

        }
```

Figure 7.9: Blocked Reduction code

doing the reduce-scatter is actually not uniform. During the reduce scatter we move from a dimensions related to the offsets and sizes for the sparse matrix blocks to a local dimension for the vectors in each individual. Since we need to have the memory chunk sizes coherent for a successful and efficient task parallel implementation, implementing reduce scatter like a reduce and a scatter would not be efficient.

For this reason, we implement a custom reduction. Rather than doing a reduction followed by a scatter, only a reduction will be done in each process on specific blocks. Since we have a displacement array for doing the scatter which keeps track of the vector blocks being in a specific process, using this information, each individual block will determine which process it belongs to. If a block belongs to a particular mpi rank, it will receive the locally calculated results from other processors for this block and accumulate the result. If this

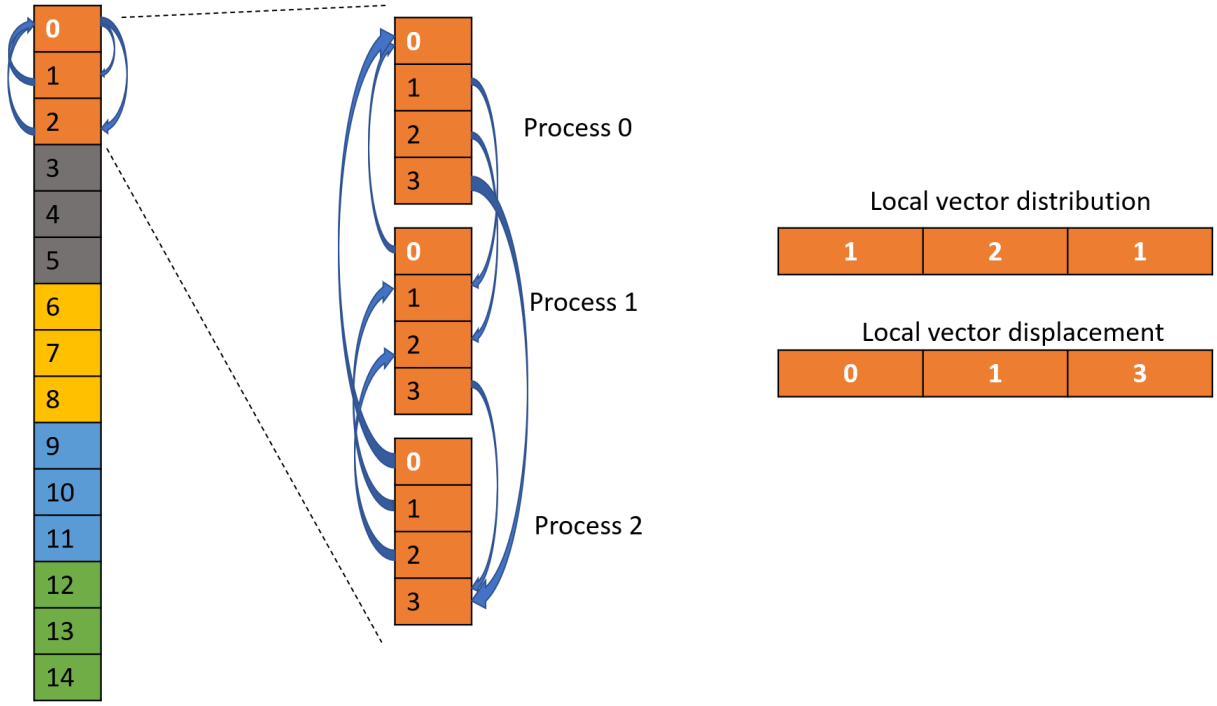


Figure 7.10: Custom Reduction depending on the local vector distribution

block does not belong to this mpi rank but a different one, it sends the locally calculated values for this block to that mpi rank which will eventually calculate the local value of its own.

In figure 7.10 an example of this custom reduction is shown. In this example, the entire subvector in the diagonal processor is divided into 4 blocks with a local distribution of 1, 2 and 1 blocks respectively. When the result of block 0 is calculated which belongs to the mpi rank 0, it will receive the local results for that task from the other processors. For blocks 1 and 2, it will send the locally calculated results to mpi rank 1 and for the block 3, it will send the result to process 3.

Processor	Xeon E5-2698 v3	Xeon Phi 7250
Core	**Haswell**	**Knights Landing**
Clock (GHz)	2.3	1.4
Data Cache (KB)	64(32+32)+256	64(32+32) + 512
Memory-Parallelism	HW-prefetch	HW-prefetch
Cores/Nodes	32	68
Threads/Nodes	64	272
Last-level Memory	40 MB L3	16GB MCDRAM
SP TFlop/s	1.2	3
DP TFlop/s	0.6	1.5
Available Memory	128 GB	96 GB
Interconnect	Aries(Dragonfly)	Aries (Dragonfly)
Global BW	120 GB/s	490 GB/s

Table 7.1: Overview of Evaluated Platforms. [1] *With hyper threading, but only 12 threads were used in our computations.* [2] *Based on the* `saxpy1` *benchmark in [1].* [3] *Memory bandwidth is measured using the STREAM copy benchmark.*

7.3 Experiments and Results

7.3.1 Experimental setup

We conducted all our experiments on Cori Phase I, a Cray XC40 supercomputer at NERSC, mainly using the GNU compiler. Each Cori Phase I node has two sockets with a 16-core Intel Xeon Processor E5-2698 v3 Haswell CPUs. Each core has a 64 KB private L1 cache (32 KB instruction and 32 KB data cache) and a 256 KB private L2 cache. Each CPU has a 40 MB shared L3 cache (LLC). We use thread affinity to bind threads to cores and use a maximum of 16 threads to avoid NUMA issues.

We also conducted all our validation experiments on Cori Phase II (Cori-KNL), a Cray XC40 supercomputer at NERSC. Each Cori-KNL node is a single-socket Intel Xeon Phi Processor 7250 ("Knights Landing") processor with 68 cores per node @ 1.4 GHz. Each node has 96 GB DDR4 2400 MHz memory with 102 GiB/s peak bandwidth and also a 16 GB MCDRAM (multi-channel DRAM).

MPI ranks	Dimension	local dim	nonzero
6	3354349	1677320	1063773871
15	3369600	1123336	1025614613
45	5771535	1154406	1044593229
66	6189415	1031593	958230159
120	8161289	1020212	968138528

Table 7.2: Matrices used in this experiment. Number of MPI ranks, dimensions and number of nonzeroes per rank.

For our initial implementations, we received a standalone code consisting of the Lobpcg algorithms, the preconditioning and the actual SpMM code from our collaborators. we did our implementations in a C wrapper code and merged it with the existing FORTRAN code.

We received 5 different matrices from our collaborators which are different in sizes. The matrices mainly varies in the MPI ranks they use and the number of diagonals. In table 7.2 we show the dimensions for SpMM multiplication, the average local dimensions of the vectors in each mpi ranks and number of nonzeroes per node. We can see the the number of nonzeroes are kept somewhat close to 1 billion per rank. Hence we expect a weak scaling in our experiments.

Since we have 128 GB per haswell nodes and 96 GB per knl nodes, if we want to use half the memory of the entire nodes for the matrix, we can have 8 MPI ranks per haswell nodes(8*8 GB = 64 GB). But empirically we have seen that the code performs best when we use 2 mpi ranks per node. Hence for our matrices, we use 3, 8, 23, 33 and 60 Haswell and Knl nodes and 2 mpi ranks per node. Since we have 32 cores in Haswell nodes and 68 cores in knl nodes, we can use 16 OpenMP threads and 32 OpenMP threads for Haswell and knl runs repectively. Also we have used hyperthreading to use 32 and 64 OpenMP threads respectively in Haswell and knl runs.

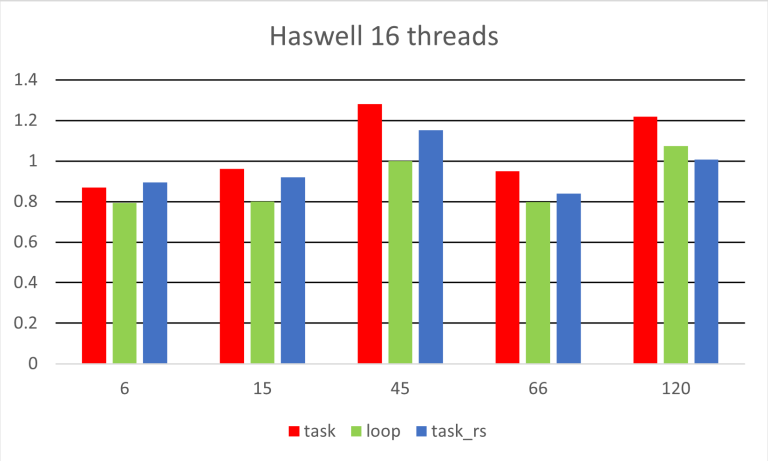


Figure 7.11: Comparison of execution time per iteration in Haswell 16 threads between loop parallel, task parallel and task parallel with custom reduce-scatter

7.3.2 Impact of blocked communications

In figures 7.11,7.12,7.13 and 7.14 we show the performance per LOBPCG iteration in MFDn code. We compare the execution time between the loop parallel version and our task parallel implementation. We show the results for both Haswell nodes and knl nodes with and without hyperthreading. We observe that the task parallel implementation is slightly slower than the loop parallel version for almost all the experiments. The huge number of tasks generated and the dependencies being resolved regularly might be generating a tasking overhead which results in a slightly slower performance.

7.3.3 Improvement with custom reduction

We also implemented the reduce-scatter with a custom reduction and measured the results and show them in the same plots. For this case, the tasks in reduce scatter are also included in the single omp loop thus there are more overlaps between computation and communication. This is also evident in our results. In almost all the experiments, the version with custom reduction is faster than the version without this custom implementation. But even this

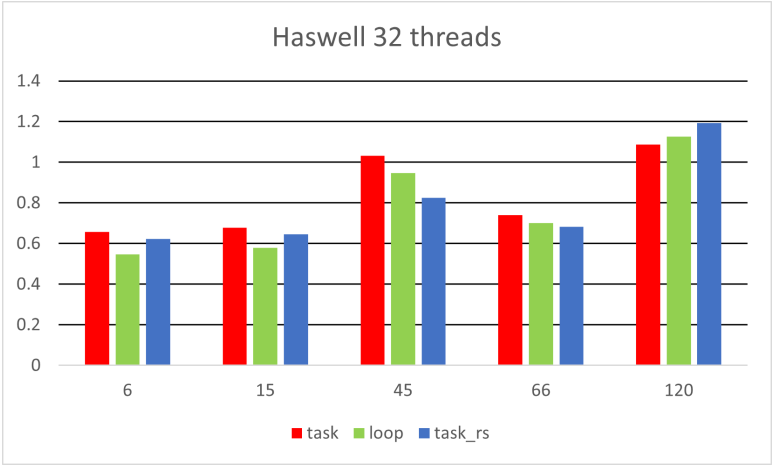


Figure 7.12: Comparison of execution time per iteration in Haswell 32 threads between loop parallel, task parallel and task parallel with custom reduce-scatter

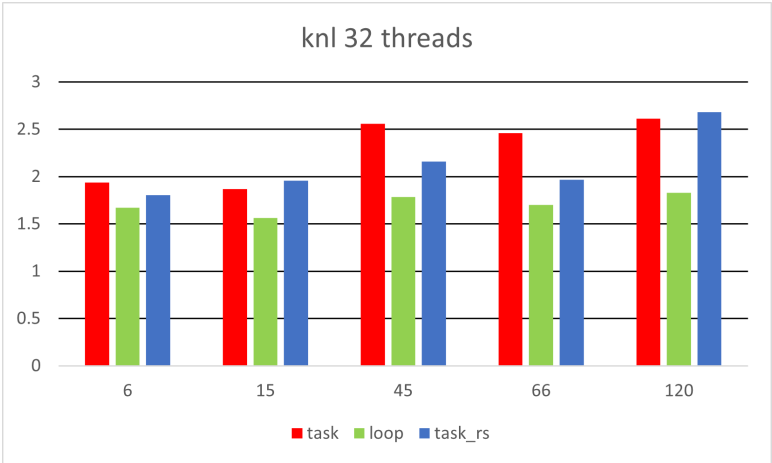


Figure 7.13: Comparison of execution time per iteration in knl 32 threads between loop parallel, task parallel and task parallel with custom reduce-scatter

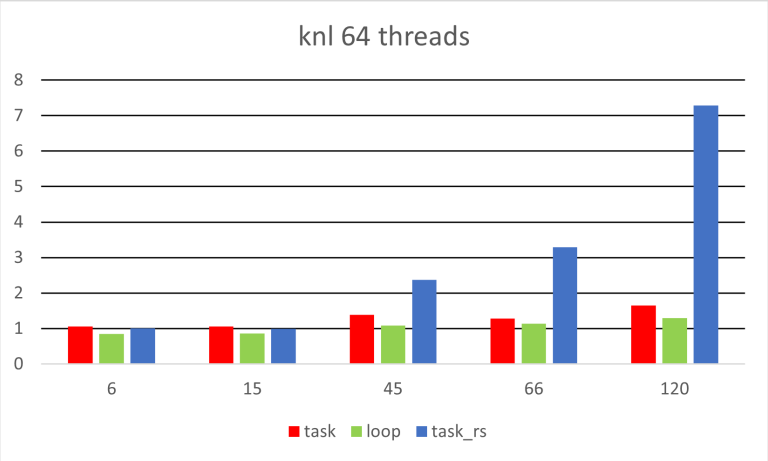


Figure 7.14: Comparison of execution time per iteration in knl 64 threads between loop parallel, task parallel and task parallel with custom reduce-scatter

improved task parallel implementation could not always beat the loop parallel times.

7.3.4 Breakdown of individual kernel performance

In MFDn we mainly have four expensive and important kernels. The bcast, the SpMM, the reduction and the SpMMt. We broke down the time needed to execute these 4 kernels and compared those between the loop parallel version and the task parallel version. We broke down the timings for 6 MPI ranks and 45 MPI ranks cases. In the loop parallel version, the communications are done by only one thread. And all the other threads take part in the SpMM and SpMMt. Whereas our entire implementation is task parallel and every thread takes part in all four kernels. In figures 7.15, 7.16, 7.17 and 7.18 we can see the comparison. It is clear that the communication times improved in task parallel version over the loop parallel version since more threads are taking part in the communication now.

Since the number of threads taking part in SpMM and SpMMt have been somewhat balanced for task parallel version, their execution time increases slightly. Eventually with this increase and also because of tasking overhead, the final execution time per iteration is

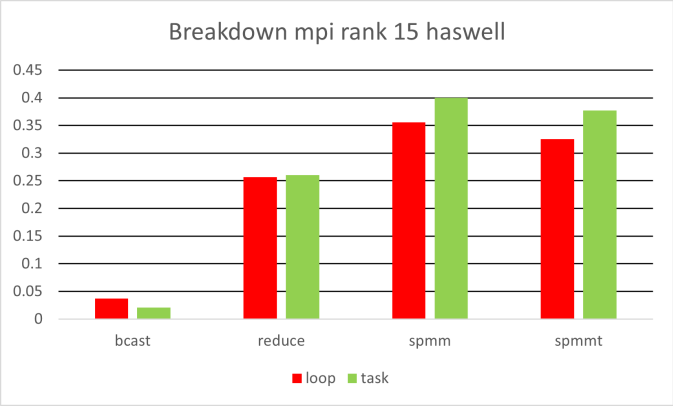


Figure 7.15: Breakdown of communication and computation operations with 6 mpi ranks in Haswell nodes

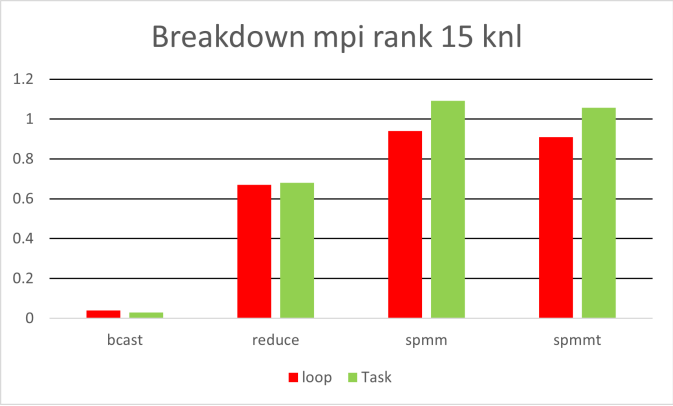


Figure 7.16: Breakdown of communication and computation operations with 6 mpi ranks in KNL nodes

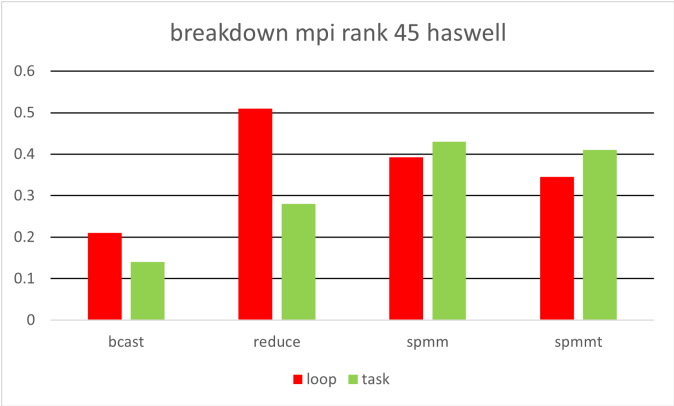


Figure 7.17: Breakdown of communication and computation operations with 45 mpi ranks in haswell nodes

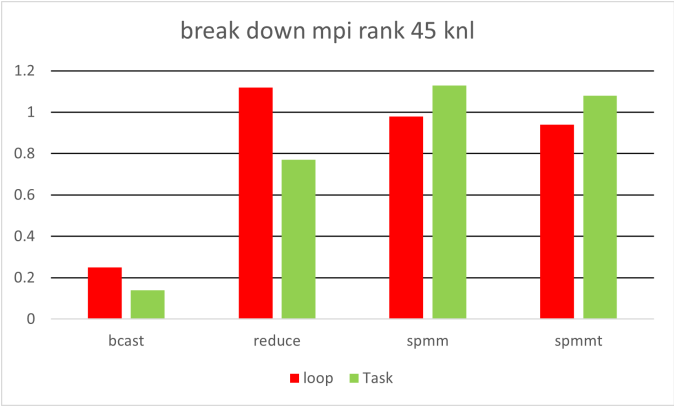


Figure 7.18: Breakdown of communication and computation operations with 45 mpi ranks in knl nodes

increased slightly.

7.3.5 Expensive matrix multiplication compared to vector operations

As mentioned earlier, one of our motivations behind this work is to overlap computation with communication. Since we saw a better pipelined execution flow results in improvements in performance for a shared node where the matrix and vector operations have a balance, we thought this will lead to better performance in MFDn too.

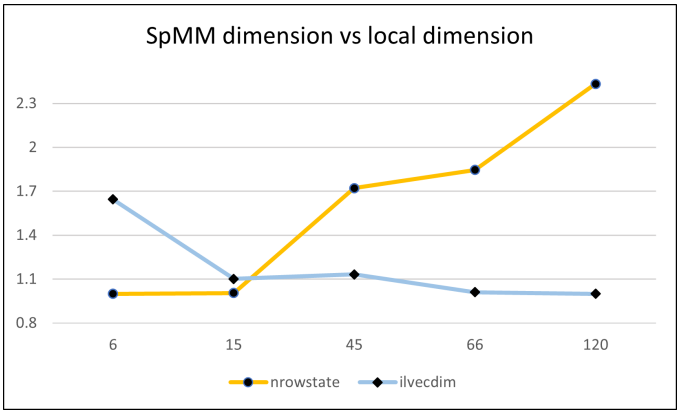


Figure 7.19: Change in SpMM dimension and local dimensions with the increase of mpi ranks

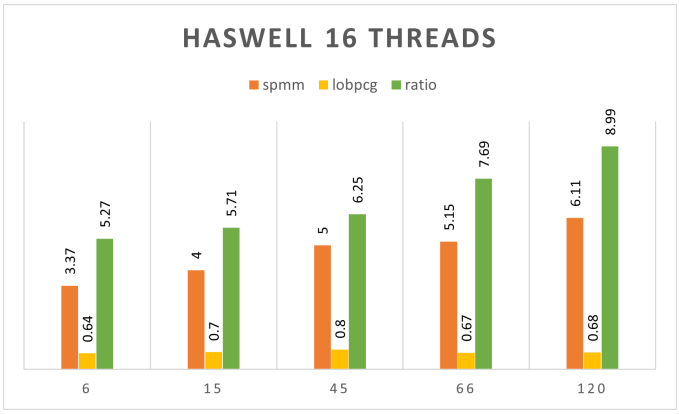


Figure 7.20: Ratio of LOBPCG compared to SpMM in Haswell nodes with 16 threads

As seen in table 7.2 and also pictorially shown in figure 7.19, we can see that with the increase of MPI ranks, the local dimensions for the vector actually decreases. This means the individual matrix dimensions actually increases and since the number of non zeros are kept similar per mpi rank, the matrises are more and more sparser with the increase of mpi ranks.

Also, with the decrease of the local vector dimensions, the vector operations become smaller compared to the matrix multiplications.

We can see in the breakdown sections that with the increase in dimensions, the communication also becomes expensive in SpMM making the SpMM more and more expensive compared to the vector operations.

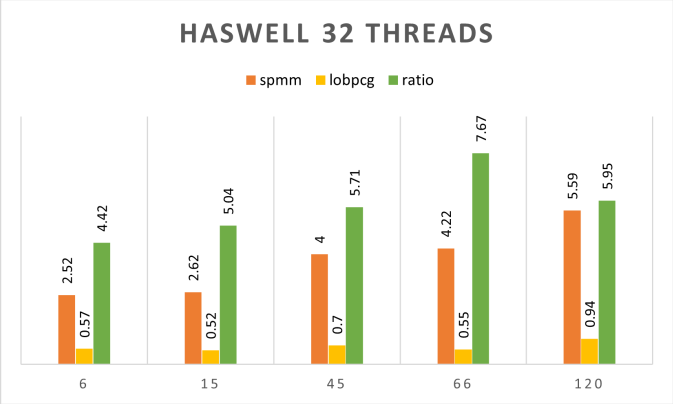


Figure 7.21: Ratio of LOBPCG compared to SpMM in Haswell nodes with 32 threads

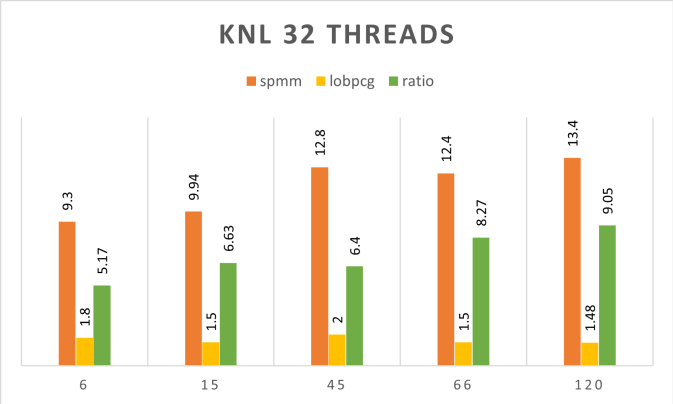


Figure 7.22: Ratio of LOBPCG compared to SpMM in knl nodes with 32 threads

In figures 7.20, 7.21, 7.22 and 7.23 we show the ratio between SpMM execution time and LOBPCG execution time which consists of vector operations. Unlike out shared memory version, the collective vector vector and vector transpose multiplication is almost 9 times slower than SpMM which was not the case for shared memory. This was an interesting observation for us and making the improvement not as expected as we thought it would be.

7.3.6 Conclusions of this work

To conclude this work, we introduced the communication tasks. While we faced several issues with the merging OpenMP tasks with MPI routines, we eventually succeeded. The MPI collective routines do not welcome the use of blocked tasks as we use in DeepSparse.

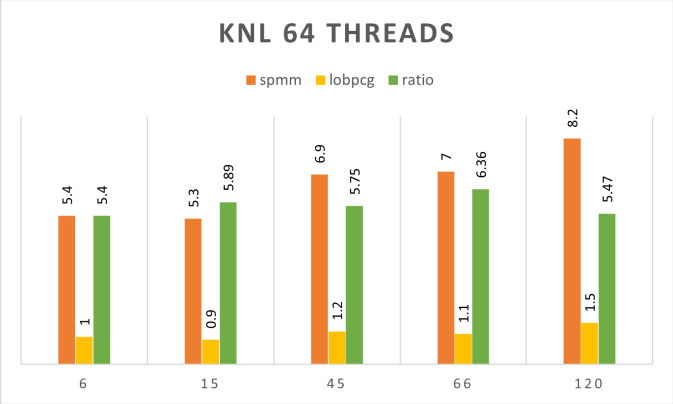


Figure 7.23: Ratio of LOBPCG compared to SpMM in knl nodes with 64 threads

We had to use point to point MPI communication routines as a result. We also had to introduce our custom reduce-scatter blocked routine. Eventually we observed that bulk synchronous implementation performs well compared to the task parallel implementation. A better adaptability in the newer MPI library would improve the implementation and also using a relatively newer MPI+Threads might actually improve the performance compared to the bulk synchronous process.

Chapter 8

CONCLUSION AND FUTURE WORK

In this thesis, we presented techniques and algorithms to optimize large scale iterative eigensolvers in deep memory architectures using task parallel approach. Although we started with a dense matrix code, eventually our work focused towards sparse matrix iterative eigensolvers.

At first we worked with a dense matrix in the Sky3D code and optimized its performance by setting up the 1d and 2d partitionings accordingly. With the help of an already rich library ScaLAPACK we observed great scaling. We also observed the performance of a pure MPI implementation and a MPI+OpenMP hybrid implementation. Dense matrices already have a pretty impressive collection of optimized multithreaded libraries which are widely used across different scientific fields.

But when it comes to sparse iterative solvers, we do not see a lot of optimized libraries as for the dense matrices. There are a lot of scopes to optimize the sparse eigensolvers. We discuss about such an approach that we implemented blocked version of sparse matrix matrix multiplication where the matrices are stored in a novel way of blocks rather than traditional compressed row or compressed columns. We discuss the implementation and the benefit of using the blocked storage version. We also discuss about the roofline model and the performance achieved by our implementation in Intel Xeon Phi architectures and Intel

Ivy Bridge machines.

We only concentrated on a single kernel for this case but for our next work we targeted an entire eigensolver and looked at all the steps in an eigensolvers rather than looking only a single kernel. We introduced a novel task-parallel framework named DeepSparse which takes all the steps of an eigensolver and distributes it into tasks on blocked matrices and executed these tasks in parallel using OpenMP tasks. We observed a reduction in runtime and a huge reduction of cache misses in all level of memory using our task parallel approach.

In this work we depended on the scheduler generated by OpenMP. We also wanted to use our own custom scheduler. With this goal we used a graph partitioner to partition graphs with a tight memory bound for active memories as inputs and outputs at a particular phase. After creating those phases, we created partitions by sorting topologically. Although we convincingly improved over the library implementations, we could not outperform the openmp scheduler. We need to find a different heuristic to work on with the paritioner.

With the success of a single node implementation of DeepSparse, we targeted the distributed MFDn algorithm for DeepSparse with a view to optimize distributed iterative eigensolvers. Before diving straight into the implementations we wanted to simulate the behaviour of communications in MFDn. We created a communication motif and simulated it using Structural Simulation Toolkit(SST). In the process we found out a network congestion issue present in the actual machines for large message sizes for communication routines. We plan to introduce task based blocked communication routines and we expect it to improve the performance of the distributed eigensolver.

Eventually we implemented a distributed memory version of DeepSparse and implemented a task parallel SpMM in the MFDn code. For the distributed memory application, communication plays a big role in SpMM being dominant over vector operations which was

not the case for shared memory version. The vector operations still need to be executed in the same openmp block with SpMM. Also at the moment the communication tasks are point to point and the collective communications are done serially. A tree based communication can be introduced for further improvement.

A more adaptable MPI library with OpenMP tasks might improve the performance of distributed MFDn implementation. A recent approach of MPI+Threads might also make the implementation faster and a more simplified implementation. We will keep looking for better approaches to improve the custom scheduler using partitioning and also to improve the distributed implementation of communication routines.

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] J. Fang, H. Sips, L. Zhang, C. Xu, Y. Che, and A. L. Varbanescu, "Test-driving Intel Xeon Phi," in *5th ACM/SPEC International Conference on Performance Engineering*. ACM, 2014, pp. 137–148.

[2] P. Maris, M. Sosonkina, J. P. Vary, E. Ng, and C. Yang, "Scaling of ab-initio nuclear physics calculations on multicore computer architectures," *Procedia Computer Science*, vol. 1, no. 1, pp. 97 – 106, 2010.

[3] P. Sternberg, E. G. Ng, C. Yang, P. Maris, J. P. Vary, M. Sosonkina, and H. V. Le, "Accelerating configuration interaction calculations for nuclear structure," in *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*. IEEE Press, 2008, pp. 1–15.

[4] M. Shao, H. Aktulga, C. Yang, E. G. Ng, P. Maris, and J. P. Vary, "Accelerating nuclear configuration interaction calculations through a preconditioned block iterative eigensolver," *Computer Physics Communications*, 2017. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465517302904

[5] J. Kim, W. J. Dally, S. Scott, and D. Abts, "Technology-driven, highly-scalable dragonfly topology," in *2008 International Symposium on Computer Architecture*. IEEE, 2008, pp. 77–88.

[6] N. Satish, N. Sundaram, M. M. A. Patwary, J. Seo, J. Park, M. A. Hassaan, S. Sengupta, Z. Yin, and P. Dubey, "Navigating the maze of graph analytics frameworks using massive graph datasets," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*. ACM, 2014, pp. 979–990.

[7] Y. Saad, *Iterative methods for sparse linear systems*. siam, 2003, vol. 82.

[8] M. Feit, J. Fleck Jr, and A. Steiger, "Solution of the schrödinger equation by a spectral method," *Journal of Computational Physics*, vol. 47, no. 3, pp. 412–433, 1982.

[9] A. Y. Ng, M. I. Jordan, and Y. Weiss, "On spectral clustering: Analysis and an algorithm," in *Advances in neural information processing systems*, 2002, pp. 849–856.

[10] I. Jollife, *Principal Component Analysis*. Wiley Online Library, 2002.

[11] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web." Stanford InfoLab, Tech. Rep., 1999.

[12] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," *Communications of the Association for Computing Machinery*, 2009.

[13] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proceedings of the conference on high performance computing networking, storage and analysis*. ACM, 2009, p. 18.

[14] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson, "Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks," in *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. ACM, 2009, pp. 233–244.

[15] E.-J. Im and K. A. Yelick, *Optimizing the performance of sparse matrix-vector multiplication*. University of California, Berkeley, 2000.

[16] B. C. Lee, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, "Performance models for evaluation and automatic tuning of symmetric sparse matrix-vector multiply," in *Parallel Processing, 2004. ICPP 2004. International Conference on*. IEEE, 2004, pp. 169–176.

[17] R. Nishtala, R. W. Vuduc, J. W. Demmel, and K. A. Yelick, "When cache blocking of sparse matrix vector multiply works and why," *Applicable Algebra in Engineering, Communication and Computing*, vol. 18, no. 3, pp. 297–311, 2007.

[18] A. Pinar and M. T. Heath, "Improving performance of sparse matrix–vector multiplication," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*. ACM, 1999, p. 30.

[19] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. Philadelpha, PA: SIAM, 2003.

[20] R. Vuduc, J. W. Demmel, and K. A. Yelick, "OSKI: A library of automatically tuned sparse matrix kernels," in *Journal of Physics: Conference Series*, vol. 16, no. 1. IOP Publishing, 2005, p. 521.

[21] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. IEEE, 2007, pp. 1–12.

[22] H. M. Aktulga, A. Buluç, S. Williams, and C. Yang, "Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*. IEEE, 2014, pp. 1213–1222.

[23] W. D. Gropp, D. K. Kaushik, D. E. Keyes, and B. F. Smith, "Toward realistic performance bounds for implicit cfd codes," in *Proceedings of parallel CFD*, vol. 99. Citeseer, 1999, pp. 233–240.

[24] X. Liu, E. Chow, K. Vaidyanathan, and M. Smelyanskiy, "Improving the performance of dynamical simulations via multiple right-hand sides," in *Parallel & Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*. IEEE, 2012, pp. 36–47.

[25] M. Röhrig-Zöllner, J. Thies, M. Kreutzer, A. Alvermann, A. Pieper, A. Basermann, G. Hager, G. Wellein, and H. Fehske, "Increasing the performance of the jacobi–davidson method by blocking," *SIAM Journal on Scientific Computing*, vol. 37, no. 6, pp. C697–C722, 2015.

[26] A. E. Sarıyüce, E. Saule, K. Kaya, and Ü. V. Çatalyürek, "Regularizing graph centrality computations," *Journal of Parallel and Distributed Computing*, vol. 76, pp. 106–119, 2015.

[27] A. Buluç and J. R. Gilbert, "Parallel sparse matrix-matrix multiplication and indexing: Implementation and experiments," *SIAM Journal on Scientific Computing*, vol. 34, no. 4, pp. C170–C191, 2012.

[28] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*. Elsevier, 2011.

[29] P. Kogge and J. Shalf, "Exascale computing trends: Adjusting to the" new normal" for computer architecture," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 16–26, 2013.

[30] A. V. Knyazev, "Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method," *SIAM Journal on Scientific Computing*, vol. 23, no. 2, pp. 517–541, 2001.

[31] H. A. Bethe, "Supernova mechanisms," *Reviews of Modern Physics*, vol. 62, no. 4, pp. 801–866, 10 1990. [Online]. Available: http://link.aps.org/doi/10.1103/RevModPhys.62.801

[32] H. Suzuki, *Physics and Astrophysics of Neutrinos*, M. Fukugita and A. Suzuki, Eds. Springer, Tokyo, 1994.

[33] D. G. Ravenhall, C. J. Pethick, and J. R. Wilson, "Structure of Matter below Nuclear Saturation Density," *Phys. Rev. Lett.*, vol. 50, no. 26, pp. 2066–2069, 6 1983. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevLett.50.2066

[34] M. Hashimoto, H. Seki, and M. Yamada, "Shape of Nuclei in the Crust of Neutron Star," *Prog. Theor. Phys.*, vol. 71, no. 2, pp. 320–326, 1984. [Online]. Available: http://ptp.ipap.jp/link?PTP/71/320/

[35] A. S. Schneider, C. J. Horowitz, J. Hughto, and D. K. Berry, "Nuclear "pasta" formation," *Phys. Rev. C*, vol. 88, no. 6, p. 65807, 2013. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevC.88.065807

[36] A. S. Schneider, D. K. Berry, C. M. Briggs, M. E. Caplan, and C. J. Horowitz, "Nuclear "waffles"," *Phys. Rev. C*, vol. 90, no. 5, p. 55805, 2014.

[37] C. J. Horowitz, D. K. Berry, C. M. Briggs, M. E. Caplan, A. Cumming, and A. S. Schneider, "Disordered nuclear pasta, magnetic field decay, and crust cooling in neutron stars," *Phys. Rev. Lett.*, vol. 114, no. 3, 2015.

[38] G. Watanabe, K. Sato, K. Yasuoka, and T. Ebisuzaki, "Microscopic study of slablike and rodlike nuclei: Quantum molecular dynamics approach," *Phys. Rev. C*, vol. 66, no. 1, p. 6, 2002.

[39] G. Watanabe and others, "Structure of cold nuclear matter at subnuclear densities by quantum molecular dynamics," *Phys. Rev. C*, vol. 68, no. 3, p. 35806, 2003.

[40] G. Watanabe, T. Maruyama, K. Sato, K. Yasuoka, and T. Ebisuzaki, "Simulation of transitions between "pasta" phases in dense matter," *Phys. Rev. Lett.*, vol. 94, no. 3, 2005.

[41] G. Watanabe, H. Sonoda, T. Maruyama, K. Sato, K. Yasuoka, and T. Ebisuzaki, "Formation of Nuclear "Pasta" in Supernovae," *Phys. Rev. Lett.*, vol. 103, no. 12, p. 121101, 2009.

[42] R. D. Williams and S. E. Koonin, "Sub-saturation phases of nuclear matter," *Nucl. Phys.*, vol. 435, no. 3?4, pp. 844–858, 1985. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0375947485901915

[43] K. Oyamatsu, "Nuclear shapes in the inner crust of a neutron star," *Nuclear Physics A*, vol. 561, no. 3, pp. 431–452, 8 1993. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/037594749390020X

[44] M. Okamoto, T. Maruyama, K. Yabana, and T. Tatsumi, "Nuclear "pasta" structures in low-density nuclear matter and properties of the neutron-star crust," *Phys. Rev. C*, vol. 88, p. 25801, 2013.

[45] H. Pais, S. Chiacchiera, and C. Providência, "Light clusters, pasta phases, and phase transitions in core-collapse supernova matter," *Phys. Rev. C*, vol. 91, no. 5, p. 55801, 2015. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevC.91.055801

[46] P. Magierski and P.-H. Heenen, "Structure of the inner crust of neutron stars: Crystal lattice or disordered phase?" *Phys. Rev. C*, vol. 65, no. 4, p. 45804, 4 2002. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevC.65.045804

[47] P. Gögelein and H. Müther, "Nuclear matter in the crust of neutron stars," *Phys. Rev. C*, vol. 76, no. 2, p. 24312, 8 2007. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevC.76.024312

[48] W. G. Newton and J. R. Stone, "Modeling nuclear "pasta" and the transition to uniform nuclear matter with the 3D Skyrme-Hartree-Fock method at finite temperature: Core-collapse supernovae," *Phys. Rev. C*, vol. 79, no. 5, p. 55801, 2009.

[49] H. Sonoda, G. Watanabe, K. Sato, K. Yasuoka, and T. Ebisuzaki, "Phase diagram of nuclear "pasta" and its uncertainties in supernova cores," *Phys. Rev. C*, vol. 77, no. 3, p. 35806, 2008. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevC.77.035806

[50] B. Schuetrumpf, K. Iida, J. A. Maruhn, and P. G. Reinhard, "Nuclear "pasta matter" for different proton fractions," *Phys. Rev. C*, vol. 90, no. 5, 2014.

[51] B. Schuetrumpf and W. Nazarewicz, "Twist-averaged boundary conditions for nuclear pasta Hartree-Fock calculations," *Phys. Rev. C*, vol. 92, no. 4, 2015.

[52] I. Sagert, G. I. Fann, F. J. Fattoyev, S. Postnikov, and C. J. Horowitz, "Quantum simulations of nuclei and nuclear pasta with the multiresolution adaptive numerical environment for scientific simulations," *Phys. Rev. C*, vol. 93, no. 5, 2016.

[53] F. J. Fattoyev, C. J. Horowitz, and B. Schuetrumpf, "Quantum nuclear pasta and nuclear symmetry energy," *Phys. Rev. C*, vol. 95, no. 5, p. 055804, 5 2017. [Online]. Available: http://arxiv.org/abs/1703.01433http://dx.doi.org/10.1103/PhysRevC.95.055804http://link.aps.org/doi/10.1103/PhysRevC.95.055804

[54] C. J. Horowitz, D. K. Berry, C. M. Briggs, M. E. Caplan, A. Cumming, and A. S. Schneider, "Disordered Nuclear Pasta, Magnetic Field Decay, and Crust Cooling in Neutron Stars," *Phys. Rev. Lett.*, vol. 114, no. 3, p. 31102, 1 2015. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevLett.114.031102

[55] J. Erler, N. Birge, M. Kortelainen, W. Nazarewicz, E. Olsen, A. M. Perhac, and M. Stoitsov, "The limits of the nuclear landscape," *Nature*, vol. 486, no. 7404, pp. 509–512, jun 2012. [Online]. Available: http://dx.doi.org/10.1038/nature11188http://www.nature.com/nature/journal/v486/n7404/abs/nature11188.html{#}supplementary-information

[56] M. Bender, P.-H. Heenen, and P.-G. Reinhard, "Self-consistent mean-field models for nuclear structure," *Reviews of Modern Physics*, vol. 75, no. 1, pp. 121–180, 1 2003. [Online]. Available: http://link.aps.org/doi/10.1103/RevModPhys.75.121

[57] M. V. Stoitsov, N. Schunck, M. Kortelainen, N. Michel, H. Nam, E. Olsen, J. Sarich, and S. Wild, "Axially deformed solution of the Skyrme-Hartree-Fock-Bogoliubov equations using the transformed harmonic oscillator basis (II) hfbtho v2.00d: A new version of the program," *Computer Physics Communications*, vol. 184, no. 6, pp. 1592–1604, 2013.

[58] N. Schunck, J. Dobaczewski, J. McDonnell, W. Satuła, J. A. Sheikh, A. Staszczak, M. Stoitsov, and P. Toivanen, "Solution of the Skyrme–

Hartree–Fock–Bogolyubov equations in the Cartesian deformed harmonic-oscillator basis.: (VII) hfodd (v2.49t): A new version of the program," *Computer Physics Communications*, vol. 183, no. 1, pp. 166–192, 2012. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465511002852

[59] J. A. Maruhn, P.-G. Reinhard, P. D. Stevenson, and A. S. Umar, "The {TDHF} code Sky3D," *Comput. Phys. Commun.*, vol. 185, no. 7, pp. 2195–2216, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0010465514001313

[60] T. Ichikawa, J. A. Maruhn, N. Itagaki, K. Matsuyanagi, P.-G. Reinhard, and S. Ohkubo, "Existence of an Exotic Torus Configuration in High-Spin Excited States of 40Ca," *Phys. Rev. Lett.*, vol. 109, p. 232503, 2012. [Online]. Available: http://dx.doi.org/10.1103/PhysRevLett.109.232503

[61] M. Stein, J. A. Maruhn, A. Sedrakian, and P.-G. Reinhard, "Carbon-oxygen-neon mass nuclei in superstrong magnetic fields," *Phys. Rev. C*, vol. 94, p. 35802, 2016. [Online]. Available: 10.1103/PhysRevC.94.035802

[62] P.-G. Reinhard, L. Guo, and J. A. Maruhn, "Nuclear Giant Resonances and Linear Response," *Eur. Phys. J. A*, vol. 32, p. 19, 2007. [Online]. Available: http://dx.doi.org/10.1140/epja/i2007-10366-9

[63] B. Schuetrumpf, W. Nazarewicz, and P. G. Reinhard, "Time-dependent density functional theory with twist-averaged boundary conditions," 3 2016. [Online]. Available: http://arxiv.org/abs/1603.03743http://dx.doi.org/10.1103/PhysRevC.93.054304

[64] J. A. Maruhn, P.-G. Reinhard, P. D. Stevenson, and M. R. Strayer, "Spin-excitation mechanisms in Skyrme-force time-dependent Hartree-Fock calculations," *Phys. Rev. C*, vol. 74, no. 2, p. 027601, 8 2006. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevC.74.027601

[65] N. Loebl, J. A. Maruhn, and P.-G. Reinhard, "Equilibration in the time-dependent Hartree-Fock approach probed with the Wigner distribution function," *Phys. Rev. C*, vol. 84, p. 34608, 2011. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevC.84.034608

[66] L. Guo, P.-G. Reinhard, and J. A. Maruhn, "Conservation Properties in the Time-Dependent {H}artree {F}ock Theory," *Phys. Rev. C*, vol. 77, p. 41301, 2008. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevC.77.041301

[67] G. H. Golub and C. F. Van Loan, *Matrix computations*, 4th ed. Johns Hopkins University Press, 2013.

[68] R. B. Lehoucq, D. C. Sorensen, and C. Yang, *ARPACK Users' Guide: Solution of Large-Scale Eigenvalue Problems with Implicitly Restarted Arnoldi Methods*. SIAM, 1998.

[69] K. Wu and H. Simon, "Thick-restart Lanczos method for large symmetric eigenvalue problems," *SIAM Journal on Matrix Analysis and Applications*, vol. 22, no. 2, pp. 602–616, 2000.

[70] A. Stathopoulos and Y. Saad, "Restarting techniques for the (Jacobi–)Davidson symmetric eigenvalue methods," *Electronic Transactions on Numerical Analysis*, vol. 7, pp. 163–181, 1998.

[71] G. H. Golub and R. Underwood, "The block Lanczos method for computing eigenvalues," *Mathematical Software*, vol. 3, pp. 361–377, 1977.

[72] A. Stathopoulos and J. R. McCombs, "A parallel, block, Jacobi–Davidson implementation for solving large eigenproblems on coarse grain environment," in *PDPTA*, 1999, pp. 2920–2926.

[73] A. V. Knyazev, "Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method," *SIAM Journal on Scientific Computing*, vol. 23, no. 2, pp. 517–541, 2001.

[74] Y. Saad, "On the rates of convergence of the Lanczos and the block-Lanczos methods," *SIAM Journal on Numerical Analysis*, vol. 17, no. 5, pp. 687–706, 1980.

[75] S. Williams, A. Watterman, and D. Patterson, "Roofline: An insightful visual performance model for floating-point programs and multicore architectures," *Comm. of the ACM*, April 2009.

[76] J.-H. Byun, R. Lin, K. A. Yelick, and J. Demmel, "Autotuning sparse matrix–vector multiplication for multicore," Technical report, EECS Department, University of California, Berkeley, Tech. Rep., 2012.

[77] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and E. Leiserson, "Parallel sparse matrix–vector and matrix-transpose–vector multiplication using compressed sparse blocks," in *SPAA*, 2009, pp. 233–244.

[78] P. Maris, H. M. Aktulga, S. Binder, A. Calci, Ü. V. Çatalyürek, J. Langhammer, E. Ng, E. Saule, R. Roth, J. P. Vary *et al.*, "No-Core CI calculations for light nuclei with chiral 2-and 3-body forces," *Journal of Physics: Conference Series*, vol. 454, no. 1, p. 012063, 2013.

[79] W.-Y. Chen, Y. Song, H. Bai, C.-J. Lin, and E. Y. Chang, "Parallel spectral clustering in distributed systems," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 33, no. 3, pp. 568–586, 2011.

[80] U. Von Luxburg, "A tutorial on spectral clustering," *Statistics and Computing*, vol. 17, no. 4, pp. 395–416, 2007.

[81] M. W. Berry, "Large-scale sparse singular value computations," *International Journal of Supercomputer Applications*, vol. 6, no. 1, pp. 13–49, 1992.

[82] S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman, "Indexing by latent semantic analysis," *JASIS*, vol. 41, no. 6, pp. 391–407, 1990.

[83] H. Zha, O. Marques, and H. D. Simon, "Large-scale SVD and subspace methods for information retrieval," in *Solving Irregularly Structured Problems in Parallel*. Springer, 1998, pp. 29–42.

[84] J. Kepner, D. Bade, A. Buluç, J. Gilbert, T. Mattson, and H. Meyerhenke, "Graphs, matrices, and the graphblas: Seven good reasons," *arXiv preprint arXiv:1504.01039*, 2015.

[85] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammerling, A. McKenney *et al.*, "Lapack users' guide, vol. 9," *Society for Industrial Mathematics*, vol. 39, 1999.

[86] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic linear algebra subprograms for fortran usage," *ACM Transactions on Mathematical Software (TOMS)*, vol. 5, no. 3, pp. 308–323, 1979.

[87] A. OpenMP, "Openmp application program interface version 4.0," 2013.

[88] A. R. Gilbert Hendry, "SST: A simulator for exascale co-design," in *n ASCR/ASC Exascale Research Conf.*, 2012.

[89] P. Maris, M. Sosonkina, J. P. Vary, E. Ng, and C. Yang, "Scaling of ab-initio nuclear physics calculations on multicore computer architectures," *Procedia Computer Science*, vol. 1, no. 1, pp. 97 – 106, 2010, iCCS 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S187705091000013X

[90] H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, and J. P. Vary, "Improving the scalability of a symmetric iterative eigensolver for multi-core platforms," *Concurrency Computat. Pract. Exper.*, vol. 26, no. 16, pp. 2631–2651, 2014.

[91] B. R. Barrett, P. Navratil, and J. P. Vary, "Ab initio no core shell model," *Prog. Part. Nucl. Phys.*, vol. 69, pp. 131–181, 2013.

[92] C. Lanczos, "An iteration method for the solution of the eigenvalue problem of linear differential and integral operators," *J. Res. Nat'l Bur. Std.*, vol. 45, pp. 255–282, 1950.

[93] A. V. Knyazev, "Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method," *SIAM J. Sci. Comput.*, vol. 23, no. 2, pp. 517–541, 2001.

[94] M. Kortelainen, J. McDonnell, W. Nazarewicz, P.-G. Reinhard, J. Sarich, N. Schunck, M. V. Stoitsov, and S. M. Wild, "Nuclear energy density optimization: Large deformations," *Phys. Rev. C*, vol. 85, no. 2, p. 24304, 2 2012. [Online]. Available: http://link.aps.org/doi/10.1103/PhysRevC.85.024304

[95] P. Klüpfel, P. G. Reinhard, T. J. Bürvenich, and J. A. Maruhn, "Variations on a theme by Skyrme: A systematic study of adjustments of model parameters," *Phys. Rev. C*, vol. 79, no. 3, 2009.

[96] V. Blum, G. Lauritsch, J. Maruhn, and P.-G. Reinhard, "Comparison of coordinate-space techniques in nuclear mean-field calculations," *Journal of Computational Physics*, vol. 100, no. 2, pp. 364–376, 6 1992. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/002199919290242Q

[97] A. Sunderland, S. Pickles, M. Nikolic, A. Jovic, J. Jakic, V. Slavnic, I. Girotto, P. Nash, and M. Lysaght, "An analysis of fft performance in prace application codes," *PRACE whitepaper*, 2012.

[98] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet *et al.*, *ScaLAPACK users' guide*. SIAM, 1997.

[99] M. Frigo and S. G. Johnson, "Fftw: An adaptive software architecture for the fft," in *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, vol. 3. IEEE, 1998, pp. 1381–1384.

[100] R. A. Van De Geijn and J. Watts, "Summa: Scalable universal matrix multiplication algorithm," *Concurrency-Practice and Experience*, vol. 9, no. 4, pp. 255–274, 1997.

[101] I. S. Dhillon, B. N. Parlett, and C. Vömel, "The design and implementation of the mrrr algorithm," *ACM Transactions on Mathematical Software (TOMS)*, vol. 32, no. 4, pp. 533–560, 2006.

[102] P. Löwdin, "On the Non-Orthogonality Problem Connected with the Use of Atomic Wave Functions in the Theory of Molecules and Crystals," *The Journal of Chemical Physics*, vol. 18, no. 3, 1950.

[103] H. M. Aktulga, C. Yang, E. Ng, P. Maris, and J. Vary, "Topology-aware mappings for large-scale eigenvalue problems," in *Euro-Par 2012 Parallel Processing*, ser. Lecture Notes in Computer Science (LNCS), vol. 7484. Springer Berlin/Heidelberg, 2012, pp. 830–842.

[104] H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, and J. P. Vary, "Improving the scalability of symmetric iterative eigensolver for multi-core platforms," *Concurrency and Computation: Practice and Experience*, vol. 26, pp. 2631–2651, 2013.

[105] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix–vector multiplication on emerging multicore platforms," in *Proc. SC2007: High Performance Computing, Networking, and Storage Conference*, 2007.

[106] S. Williams, "Auto-tuning performance on multicore computers," Ph.D. dissertation, University of California, Berkeley, 2008.

[107] M. Afibuzzaman, F. Rabbi, Y. Ozkaya, H. M. Aktulga, and U. V. Çatalyürek, "DeepSparse: A Task-parallel Framework for Sparse Solvers on Deep Memory Architectures," pp. 373–382, 12 2019.

[108] S. Wold, K. Esbensen, and P. Geladi, "Principal component analysis," *Chemometrics and intelligent laboratory systems*, vol. 2, no. 1-3, pp. 37–52, 1987.

[109] A. El Guennouni, K. Jbilou, and A. Riquet, "Block krylov subspace methods for solving large sylvester equations," *Numerical Algorithms*, vol. 29, no. 1-3, pp. 75–96, 2002.

[110] Z. Zhou, E. Saule, H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, J. P. Vary, and U. V. Catalyürek, "An out-of-core eigensolver on ssd-equipped clusters," in *2012 IEEE International Conference on Cluster Computing*. IEEE, 2012, pp. 248–256.

[111] H. M. Aktulga, M. Afibuzzaman, S. Williams, A. Buluç, M. Shao, C. Yang, E. G. Ng, P. Maris, and J. P. Vary, "A high performance block eigensolver for nuclear configuration interaction calculations," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 6, pp. 1550–1563, 2016.

[112] M. Shao, H. M. Aktulga, C. Yang, E. G. Ng, P. Maris, and J. P. Vary, "Accelerating nuclear configuration interaction calculations through a preconditioned block iterative eigensolver," *Computer Physics Communications*, vol. 222, pp. 1–13, 2018.

[113] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov, "Numerical linear algebra on emerging architectures: The plasma and magma projects," in *Journal of Physics: Conference Series*, vol. 180, no. 1. IOP Publishing, 2009, p. 012037.

[114] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," *Parallel Computing*, vol. 35, no. 1, pp. 38–53, 2009.

[115] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5-6, pp. 232–240, Jun. 2010.

[116] S. Tomov, R. Nath, H. Ltaief, and J. Dongarra, "Dense linear algebra solvers for multicore with GPU accelerators," in *Proc. of the IEEE IPDPS'10*. Atlanta, GA: IEEE Computer Society, April 19-23 2010, pp. 1–8, DOI: 10.1109/IPDPSW.2010.5470941.

[117] J. Dongarra, M. Gates, A. Haidar, J. Kurzak, P. Luszczek, S. Tomov, and I. Yamazaki, "Accelerating numerical dense linear algebra calculations with gpus," *Numerical Computations with GPUs*, pp. 1–26, 2014.

[118] I. S. Barrera, M. Moretó, E. Ayguadé, J. Labarta, M. Valero, and M. Casas, "Reducing data movement on large shared memory systems by exploiting computation dependencies," in *Proceedings of the 2018 International Conference on Supercomputing*. ACM, 2018, pp. 207–217.

[119] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures," in *Euro-Par - 15th International Conference on Parallel Processing*, ser. Lecture Notes in Computer Science, vol. 5704. Delft, The Netherlands: Springer, Aug. 2009, pp. 863–874. [Online]. Available: http://hal.inria.fr/inria-00384363

[120] E. Saule, H. M. Aktulga, C. Yang, E. G. Ng, and Ü. V. Çatalyürek, "An out-of-core task-based middleware for data-intensive scientific computing," in *Handbook on Data Centers*. Springer, 2015, pp. 647–667.

[121] C. Lanczos, *An iteration method for the solution of the eigenvalue problem of linear differential and integral operators*. United States Governm. Press Office Los Angeles, CA, 1950.

[122] A. V. Knyazev, "Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method," *SIAM journal on scientific computing*, vol. 23, no. 2, pp. 517–541, 2001.

[123] J. W. Demmel, *Applied Numerical Linear Algebra*. SIAM, 1997.

[124] S. Bogner *et al.*, "Computational Nuclear Quantum Many-Body Problem: The UNEDF Project," *Comput. Phys. Commun.*, vol. 184, pp. 2235–2250, 2013.

[125] P. Maris, J. P. Vary, P. Navratil, W. E. Ormand, H. Nam, and D. J. Dean, "Origin of the anomalous long lifetime of ^{14}C," *Phys. Rev. Lett.*, vol. 106, no. 20, p. 202502, 2011.

[126] S. Binder, A. Calci, E. Epelbaum, R. J. Furnstahl, J. Golak, K. Hebeler, H. Kamada, H. Krebs, J. Langhammer, S. Liebig, P. Maris, U.-G. Meißner, D. Minossi, A. Nogga, H. Potter, R. Roth, R. Skiniński, K. Topolnicki, J. P. Vary, and H. Witała, "Few-nucleon systems with state-of-the-art chiral nucleon-nucleon forces," *Phys. Rev. C*, vol. 93, no. 4, p. 044002, 2016.

[127] A. Shirokov, I. Shin, Y. Kim, M. Sosonkina, P. Maris, and J. Vary, "N3LO NN interaction adjusted to light nuclei in ab exitu approach," *Phys. Lett. B*, vol. 761, pp. 87–91, 2016.

[128] E. Epelbaum *et al.*, "Few- and many-nucleon systems with semilocal coordinate-space regularized chiral two- and three-body forces," *Phys. Rev. C*, vol. 99, no. 2, p. 024313, 2019.

[129] M. Caprio, P. Fasano, P. Maris, A. McCoy, and J. Vary, "Probing ab initio emergence of nuclear rotation," *Eur. Phys. J. A*, vol. 56, no. 4, p. 120, 2020.

[130] T. Groves, R. E. Grant, S. Hemmer, S. Hammond, M. Levenhagen, and D. C. Arnold, "(SAI) stalled, active and idle: Characterizing power and performance of large-scale dragonfly networks," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*. IEEE, 2016, pp. 50–59.

[131] J. J. Wilke, J. P. Kenny, S. Knight, and S. Rumley, "Compiler-assisted source-to-source skeletonization of application models for system simulation," in *International Conference on High Performance Computing*. Springer, 2018, pp. 123–143.

[132] T. Connors, T. Groves, T. Quan, and S. Hemmert, "Simulation framework for studying optical cable failures in dragonfly topologies," in *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE, 2019, pp. 859–864.

[133] B. Alverson, E. Froese, L. Kaplan, and D. Roweth, "Cray xc series network," *Cray Inc., White Paper WP-Aries01-1112*, 2012.

[134] S. Chunduri, T. Groves, P. Mendygral, B. Austin, J. Balma, K. Kandalla, K. Kumaran, G. Lockwood, S. Parker, S. Warren, N. Wichmann, and N. Wright, "Gpcnet: Designing a benchmark suite for inducing and measuring contention in hpc networks," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19. New York, NY, USA: Association for Computing Machinery, 2019. [Online]. Available: https://doi.org/10.1145/3295500.3356215

[135] B. S. C. I. Project, "Best practice guide for writing mpi + ompss interoperable programs," 2017. [Online]. Available: http://www.intertwine-project.eu/sites/default/files/images/INTERTWinE_Best_Practice_Guide_MPI%2BOmpSs_1.0.pdf