ACCELERATING SPARSE EIGENSOLVERS THROUGH ASYNCHRONY, HYBRID ALGORITHMS, AND HETEROGENEOUS ARCHITECTURES

By

Abdullah Alperen

A DISSERTATION

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

Computer Science—Doctor of Philosophy

2024

ABSTRACT

Sparse matrix computations comprise the core component of a broad base of scientific applications in fields ranging from molecular dynamics and nuclear physics to data mining and signal processing. Among sparse matrix computations, the eigenvalue problem has a significant place due to its common use in the area of high performance scientific computing. In nuclear physics simulations, for example, one of the most challenging problems is solving large-scale eigenvalue problems arising from nuclear structure calculations. Numerous iterative algorithms have been developed to solve this problem over the years.

Lanczos and locally optimal block preconditioned conjugate gradient (LOBPCG) are two of such popular iterative eigensolvers. Together, they present a good mix of the computational motifs encountered in sparse solvers. With this work, we describe our efforts to accelerate large-scale sparse eigensolvers by employing asynchronous runtime systems, the development of hybrid algorithms and the utilization of GPU resources.

We first evaluate three task-parallel programming models, OpenMP, HPX and Regent, for Lanczos and LOBPCG. We demonstrate these asynchronous frameworks' merit on two architectures, Intel Broadwell (a multicore processor) and AMD EPYC (a modern manycore processor). We achieve up to an order of magnitude improvement both in execution time and cache performance.

We then examine and compare a few iterative methods for solving large-scale eigenvalue problems arising from nuclear structure calculations. In particular, besides Lanczos and LOBPCG, we discuss the possibility of using block Lanczos method and the residual minimization method accelerated by direct inversion of iterative subspace (RMM-DIIS). We show that RMM-DIIS can be effectively combined with either block Lanczos and LOBPCG to yield a hybrid eigensolver that has several desirable properties.

We finally demonstrate the challenges posed by the emergence of accelerator-based computer architectures to achieve high performance for large-scale sparse computations. We particularly focus on the scalability of sparse matrix vector multiplication (SpMV) and

sparse matrix multi-vector multiplication (SpMM) kernels of Lanczos and LOBPCG. We scale their performance up to hundreds of GPUs by improving their computation through hand-optimized CUDA kernels and communication aspect through asynchronous point-to-point calls and optimized NVIDIA Collective Communications Library (NCCL) collectives.

Copyright by
ABDULLAH ALPEREN
2024

I dedicate this thesis to my parents, Kadir and Nihal, my brother, Ahmet, and my sisters, Ayşenur and Betül, for their endless support from afar in helping me achieve this goal.

ACKNOWLEDGEMENTS

Reaching this milestone would not have been possible without the support of many. First and foremost, I express my genuine gratitude to my PhD advisor, Hasan Metin Aktulga. His valuable feedback and guidance have been instrumental throughout my PhD journey. His constant support through research grants helped me greatly in focusing on my studies.

Besides my advisor, I want to thank my dear committee members, Brian O'Shea and Sandeep Kulkarni, for their valuable time and feedback. I must also express my sincere appreciation to Pieter Maris, Khaled Ibrahim, and Nan Ding for their valuable collaboration. I particularly extend my profound gratitude to Chao Yang for his precious research ideas and his service on my committee.

Finally, my heartfelt gratitude goes to my fellow lab mates, Kurt, Anik, Fazlay, Çağrı, Doğa, and Ömer, for the collaborative exchange of ideas and camaraderie.

This work was supported by the U.S. Department of Energy, Office of Science, Office of Nuclear Physics under Grant DE-SC0023175, and the National Science Foundation Office of Advanced Cyberinfrastructure under Grant 1845208.

TABLE OF CONTENTS

CHAPTER 1 INTRODUCTION 1
- 1.1 Motivation 1
- 1.2 Contributions of This Thesis 4

CHAPTER 2 AN EVALUATION OF TASK-PARALLEL FRAMEWORKS FOR SPARSE SOLVERS ON MULTICORE AND MANYCORE CPU ARCHITECTURES 7
- 2.1 Related Work 9
- 2.2 Implementation and Optimizations 11
- 2.3 Benchmark Applications 19
- 2.4 Performance Evaluation 20
- 2.5 Conclusion of This Work 32

CHAPTER 3 PERFORMANCE OF THE HPX FRAMEWORK FOR SPARSE EIGENSOLVERS ON DISTRIBUTED MEMORY ARCHITECTURES 35
- 3.1 Implementation 35
- 3.2 Results 37
- 3.3 Discussion 41
- 3.4 Conclusion of This Work 45

CHAPTER 4 HYBRID EIGENSOLVERS FOR NUCLEAR CONFIGURATION INTERACTION CALCULATIONS 46
- 4.1 Existing Algorithms 49
- 4.2 Hybrid Algorithms 61
- 4.3 Numerical examples 62
- 4.4 Conclusion of This Work 81

CHAPTER 5 SCALING EIGENSOLVERS TO HUNDREDS OF GPUS 83
- 5.1 MFDn's Proprietary Distributed SpMV Algorithm 86
- 5.2 Suggested Improvements and Proposed Schemes 92
- 5.3 Lanczos Experiments 95
- 5.4 LOBPCG Experiments 114
- 5.5 Conclusion of This Work 122

CHAPTER 6 CONCLUSION AND FUTURE WORK 124

BIBLIOGRAPHY 126

CHAPTER 1

INTRODUCTION

1.1 Motivation

Sparse matrix computations manifest themselves in many forms such as the solution of systems of linear equations, matrix factorizations, linear least squares problems, and eigenvalue problems [24]. As such, they comprise the core component of a broad base of scientific applications in fields ranging from molecular dynamics and nuclear physics to data mining and signal processing. In the presence of large-scale data, sparse matrix computations become quite challenging as they demand massive parallelism but cannot effectively utilize compute resources. This underutilization stems from the memory-bound nature of the computations, which is not only the result of low arithmetic intensity but also irregular data access patterns. The latter factor becomes more evident in modern computer architectures where cache performance holds a greater significance within deep memory hierarchies with the accumulated cost of going farther away from the processor.

To effectively perform dense matrix computations on parallel systems, there exist dense linear algebra libraries such as *i.e.,* Basic Linear Algebra Subprograms (BLAS) [37], Linear Algebra Package (LAPACK) [6] and Scalable Linear Algebra Package (ScaLAPACK) [10]), which are well established. These dense linear algebra libraries also have their own vendor optimizated implementations (*i.e.,* Intel Math Kernel Library – MKL, Cray Scientific Libraries – LibSci). However, unlike its dense matrix analogue, the state of the art for sparse matrix computations is lagging far behind. The main reason behind the slow progress in the standardization of sparse linear algebra and development of libraries for it is that sparse matrices come in very different forms and properties depending on the application area.

Among sparse matrix computations, the eigenvalue problem has a significant place due to its common use in the area of high performance scientific computing. In nuclear physics simulations, for example, one of the most challenging problems is solving large-scale eigenvalue problems arising from nuclear structure calculations. The conventional approach to

compute eigenvalues involves finding all the roots of the matrix's characteristic polynomial, which is not feasible for large-scale analysis where the matrix dimensions are in millions or even billions. Therefore, numerous iterative algorithms have been developed to solve this problem over the years. These methods work by repeatedly refining approximations to the eigenvectors or eigenvalues and can be terminated whenever the approximations reach a sufficient degree of accuracy.

Lanczos and locally optimal block preconditioned conjugate gradient (LOBPCG) are two of such popular iterative eigensolvers. While the Lanczos algorithm is representative of the relatively simple sparse solvers such as the Conjugate Gradient method and Page Rank, the LOBPCG algorithm is a complex one with several steps that involve tall-skinny matrix operations in addition to the sparse matrix computations. Together, they present a good mix of the computational motifs encountered in sparse solvers. This is particularly true given the fact that the core component of the Lanczos algorithm is the sparse matrix vector multiplication (SpMV) kernel whereas the main kernel in the LOBPCG algorithm is the sparse matrix multi-vector multiplication (SpMM).

Both SpMV and SpMM operations are widely used [20, 42] and well studied [48] in the literature. It is known that SpMV has notoriously low arithmetic intensity since only two floating point operations (one addition, one multiplication) are performed per nonzero element of the sparse matrix. Therefore, the Roofline Model by Williams et al. [63] suggests that the memory bandwidth ultimately bounds the performance of the SpMV kernel. Although SpMM has significantly higher arithmetic intensity than SpMV, the extended Roofline model that was later proposed suggests that cache bandwidth, rather than the memory bandwidth, can still be a critical performance-limiting factor for SpMM [2].

Challenges posed by large-scale sparse matrix computations as in SpMV and SpMM kernels of Lanczos and LOBPCG algorithms are not particularly well addressed by a bulk synchronous parallel (BSP) approach, which is a type of coarse-grained parallelism, and imposes a barrier synchronization at the end of each computational kernel. The two main

factors that limit the performance of the BSP approaches are (i) poor cache performance that can be attributed to coarse-grained tasks which do not fit into the last level cache (LLC) and (ii) high synchronization costs that may be exacerbated by load imbalances given the skewed distribution of nonzero values within the matrices. Therefore, a fundamentally new approach is needed to tackle these issues and improve the performance of sparse eigensolvers, which validate the emergence and increased use of asynchronous many-task (AMT) programming models.

Regardless of the approach taken to extract parallelism within large-scale sparse eigensolvers, Lanczos and LOBPCG algorithms have their own limitations and shortcomings. For a long time, the Lanczos algorithm was the default algorithm to use in sparse eigensolvers because it is easy to implement and because it is quite robust. In recent work [51], it is shown that the low lying eigenvalues can be computed efficiently by using the LOBPCG method [32]. The advantages of the LOBPCG algorithm over the Lanczos algorithm include (i) performing SpMM instead of SpMV, which introduces an additional level of concurrency in the computation and enables exploiting data locality better, and (ii) allowing to take advantage of a preconditioner that can be used to accelerate convergence. However, LOBPCG can become unstable near convergence. Although methods for stabilizing the algorithm has been developed and implemented [26, 19], they do not completely eliminate the problem. As such, one could potentially combine LOBPCG with another solver to yield a hybrid eigensolver that can effectively eliminate the numerical stability as well as outperform both Lanczos and LOBPCG.

From the hardware point of view, the challenges in limiting the power consumption and addressing the heat dissipation of chips became the bottleneck that limits the clock frequency, thereby preventing the improvement of single thread performance of central processing units (CPUs). Reaching the power wall has forced CPU manufacturers to look for alternative solutions, rather than increasing the clock speed of processors, for enhancing the performance. This seeking led to the introduction of multi- and many-core processors. In the past, these

multi- and many-core central processing units (CPUs) equipped on supercomputers were the default hardware to accelerate high performance computations. Also, this seeking simultaneously gave way to the integration of accelerators such as graphics processing units (GPUs) on supercomputing nodes.

GPUs have been used to address problems CPUs struggle to overcome for many years. Rendering and processing images, analyzing big data, training neural networks are some of the tasks that unlike CPUs, GPUs easily tackle as the huge number of small cores they have enable to achieve high level of data parallelism. Nonetheless, it should be noted that GPUs are not supposed to replace CPUs by any means. For most cases, CPU loads compute-intensive parts to the GPU and once the GPU finishes the computation, results are loaded back to the CPU. Having high memory bandwidth is a significant feature of GPUs at this point to allow it to still be efficient overall with the data movements involved.

Besides exploring new runtime systems and hybrid algorithms to accelerate the eigensolvers, the increased availability of high performance computing platforms equipped with general purpose GPUs has motivated us to consider modifying the implementation of the Lanczos and LOBPCG algorithms to enable them to run efficiently on accelerator based systems. In particular, we want to focus on the efficiency of the distributed SpMV and SpMM algorithms, which constitute a significant portion of the overall solver time. There exist performance issues of the GPU-parallel Lanczos and LOBPCG solver implementations that leverage MPI collectives for communications and OpenACC directives for the computations. As such, we want to overcome scalability bottlenecks by improving both computational and communication aspects of the SpMV and SpMM kernels.

1.2 Contributions of This Thesis

In this thesis, we describe our efforts to accelerate large-scale sparse eigensolvers by employing asynchronous runtime systems, the development of hybrid algorithms and the utilization of GPUs on distributed architectures. As explained below, the contributions of this thesis are divided into four parts.

In the first part, we evaluate three task-parallel programming models, OpenMP, HPX and Regent, in terms of performance and ease of implementation, and compare them against the traditional BSP model for two popular eigensolvers, Lanczos and LOBPCG. We give a general outline in regards to achieving parallelism using these runtime systems, and present a heuristic for tuning their performance to balance tasking overheads with the degree of parallelism that can be exposed. We then demonstrate their merits on two architectures, Intel Broadwell (a multicore processor) and AMD EPYC (a modern manycore processor). In Chapter 2, we present the details of our exploration and evaluation of these AMT models in the context of sparse matrix computations.

In the second part, we explore the capabilities of HPX runtime system on distributed architectures after seeing its merit on shared memory systems in order to assess whether we could use HPX as the backbone of a large-scale sparse solver and graph analytics framework. We compare its performance to hybrid MPI+OpenMP model up to 512 cores. In Chapter 3, we present the details of our evaluation of HPX's distributed memory performance and lack thereof.

In the third part, we examine and compare a few iterative methods for solving large-scale eigenvalue problems arising from nuclear structure calculations. In particular, besides Lanczos and LOBPCG, we discuss the possibility of using block Lanczos method, a Chebyshev filtering based subspace iterations and the residual minimization method accelerated by direct inversion of iterative subspace (RMM-DIIS). We show that RMM-DIIS can be effectively combined with either the block Lanczos and LOBPCG methods to yield a hybrid eigensolver that has several desirable properties. In Chapter 4, we present the details of a few practical issues that need to be addressed to make the hybrid solver efficient and robust.

In the fourth and last part, we work towards scaling the Lanczos and LOBPCG solvers to hundreds of GPUs. To that end, we particularly focus on the costly SpMV/SpMM algorithm on distributed memory architectures with multiple GPUs. We propose performance optimizations for the compute kernels, which are originally implemented using OpenACC

directives, via hand-tuned CUDA imeplementations. We also examine scaling issues of certain MPI collectives used in the distributed SpMV/SpMM algorithm. We then propose improving the communication aspect by replacing these MPI collectives with non-blocking point-to-point calls and NVIDIA Collective Communications Library (NCCL) routines. We demonstrated improved performance with these new approaches up to 1128 GPUs. We also propose a pipelining technique that can better hide communication overheads by overlapping them with computations. In Chapter 5, we present the findings of our work on the efficient computation and communication schemes to tackle the distributed SpMV and SpMM algorithm in order to improve the overall solver time for Lanczos and LOBPCG solvers on hundreds of GPUs.

CHAPTER 2

AN EVALUATION OF TASK-PARALLEL FRAMEWORKS FOR SPARSE SOLVERS ON MULTICORE AND MANYCORE CPU ARCHITECTURES

This chapter has been published in ACM ICPP 2021 on 05 October 2021 [4], available at: https://doi.org/10.1145/3472456.3472476.

Challenges posed by large-scale sparse matrix computations are not particularly well addressed by a bulk synchronous parallel (BSP) approach. This is due to the high synchronization cost and load imbalance issues as well as data movement overheads of the traditional BSP models arising in modern shared memory architectures. Therefore, a fundamentally new approach is needed to tackle these issues, which validate the emergence and increased use of asynchronous many-task (AMT) programming models. In this work, we explore the AMT programming models to evaluate their ability to address these issues in the context of sparse solvers. We namely evaluate the performance of OpenMP, HPX and Regent runtime systems that allow task-parallel programming over the BSP model for two popular eigensolvers, Lanczos and LOBPCG.

OpenMP's task parallelism has been commonly used and well studied since 2013 [43] as it allows extracting parallelism via scheduling and asynchronous execution of fine-grained tasks. This model has the potential to remedy both deficiencies of the BSP model with regard to cache performance and load balancing issues.

There are other runtime systems that enable fine-grained task parallelism such as HPX [29]. HPX is an advanced runtime system and a programming API that conforms to the C++11/14/17/20 standards while supporting lightweight task scheduling to expose new levels of parallelism. It also extends the standard to the distributed case by employing a global address space, which renders efficient utilization of inter-node parallelism when combined with runtime adaptive resource management. HPX has demonstrated promising results in many projects as diverse as astrophysics simulations, n-body problems and storm surge forecasting [16, 25].

Another runtime system that adopts the AMT model is Regent [52], a programming language and compiler designed for HPC. Regent programs appear to be sequential codes with calls to *tasks*, i.e., functions eligible for parallel execution. Regent runtime system discovers implicit dataflow parallelism in the code by internally computing the task dependency graph, eliminating the need for explicit synchronization. Moreover, through its ability to schedule and run tasks on distributed machines, Regent frees programmers from low level distributed memory programming. Regent is shown to achieve performance comparable to OpenMP and MPI+X for a variety of applications [57, 52].

Recently, using the AMT model in OpenMP has been shown to offer important advantages over its BSP model in the context of sparse solvers with the DeepSparse framework [1]. DeepSparse adopts a fully integrated task-parallel approach that targets all computational steps in a sparse solver rather than a single kernel such as SpMV or SpMM. DeepSparse automatically generates and expresses the entire computation as a task dependency graph (TDG) and relies on OpenMP for the execution of this TDG. Despite the larger number of tasks that must be generated and managed, DeepSparse achieved significant improvements in terms of cache misses with little overheads through pipelined execution of tasks.

Having seen the success of OpenMP's task parallelism on sparse solvers, and the lack of work on evaluation of other AMT models in this area, this work aims to discern how OpenMP, HPX and Regent compare as well as what they offer over BSP models. Our main contributions can be summarized as follows:

- a novel task-parallel implementation of two sparse solvers with different characteristics, i.e., Lanczos and LOBPCG algorithms, using the HPX and Regent runtime systems by highlighting key factors to obtain an optimized code,
- an extensive performance evaluation of DeepSparse, HPX and Regent on multicore and manycore CPU architectures using a variety of sparse matrices from different domains,
- empirical demonstration of the significant cache miss reduction across all cache levels

and the execution time improvement by up to 9.9× and 7.5× compared to highly optimized library implementations for Lanczos and LOBPCG, respectively.

- presentation of a practical rule of thumb for determining the ideal task granularity for each runtime system.

After reviewing the related work in Section 2.1, we discuss how we leverage the dataflow model in each framework in the context of sparse matrix computations and point out the key factors for optimized implementations in Secton 2.2. Section 2.3 describes the solvers used for benchmarking. Section 2.4 shows the impact of the optimizations applied, evaluates the frameworks in terms of execution time and cache performance, and provides a heuristic for choosing the task granularity.

2.1 Related Work

There are several other AMT frameworks as we try to review below. Unfortunately, we cannot examine all of them in this work, but to the best of our knowledge, cross-examination of end-to-end sparse solver performances of some important AMT frameworks constitutes a unique aspect of our work.

PaRSEC is a framework and a runtime system that aims to manage tasks through architecture aware scheduling [11]. D-PLASMA library [12] is developed using PaRSEC, and shows that PaRSEC can improve the performance of *dense linear algebra algorithms* by expressing them as a directed acyclic graph (DAG) of tasks. However, PaRSEC provides a conservative data-flow model in both shared and distributed memory as the runtime system works on the data distribution and task graph information specified by the user [56]. It also offers limited work stealing on distributed memory as inter-node scheduling relies on remote completion notifications.

StarPU is a runtime system with a unified execution model at high level [7]. Its main goal is to facilitate the generation and execution of parallel tasks on heterogeneous architectures using multiple scheduling algorithms. Nevertheless, it requires the explicit data distribution

and task creation by the developer and depends on MPI communications on distributed memory [56].

Legion runtime system extracts parallelism by dynamically identifying nested parallelism and independent tasks on account of logical (definition of objects) and physical regions (actual copies of objects) [9]. Regent compiler is essentially built on top of Legion, making it simpler to program without sacrificing performance [52].

Charm++ is a C++-based, message-driven, and portable parallel programming framework and language [30]. It can expose both task and data level parallelism through the execution of parallel processes called “chares”. Charm++ is similar to HPX in that they both (i) mitigate load imbalances in a distributed system by migrating part of the data between nodes, (ii) prefer to execute the code close to where the data resides, (iii) adopt message&data-driven approach, and (iv) employ a global address space (GAS) environment.

There are several other task-based parallel programming models such as Intel Threading Building Blocks (TBB) [34], Qthreads [62] and Intel Cilk Plus [46]. What makes HPX and Regent more appealing and convenient than these models listed above is that both can employ the same task parallel approach on distributed memory systems with automatic data migration, which improves the programmability on exascale architectures.

Thoman *et. al.* [56] provide a task-focused taxonomy for HPC technologies, including HPX, Charm++, Legion, OpenMP, StarPU, Intel Cilk Plus and TBB. Kulkarni and Lumsdaine [35] theoretically compare AMT runtimes along programming model, execution model, and implementation characteristics bases. Stpiczyński [55] evaluates the performance of OpenMP, TBB and Cilk Plus and advises how to improve performance using the Belman-Ford algorithm as an example. Wang [59] provides a guideline to help programmers select an appropriate task model between Cilk, OpenMP and HPX by drawing conclusions from six benchmarks: *Fibonacci, Knight, Pi, Sort, N-Queens, and Unbalanced-Tree-Search.* To our knowledge, our work is the first to compare the empirical performance of three AMT models; OpenMP, HPX and Regent to an optimized BSP approach within the context of

sparse solvers on both manycore and multicore architectures.

2.2 Implementation and Optimizations

In many large-scale scientific applications, sparse matrix computations are the most expensive kernels. As such, in all three frameworks (DeepSparse, HPX and Regent), we define tasks based on the decomposition of the input sparse matrices. Compared to a 1D (block row) partitioning, a 2D (sparse block) partitioning is known to expose a higher degree of parallelism while potentially reducing data movement [50]. Therefore, we adapt a 2D partitioning scheme where tasks are defined based on the Compressed Sparse Block (CSB) [13] representation of the sparse matrix. All three task-parallel versions start by partitioning the sparse matrix into CSB blocks, which also dictates the decomposition of all other data structures involved, such as input/output vectors and/or vector blocks.

Listing 2.1 An example pseudocode.

```
1 SpMM(A, X, Y, m, n); // A*X = Y
2 cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, m, n, n, 1.0, Y,
      n, Z, n, 0, Q, n); // Y*Z = Q
3 cblas_dgemm(CblasRowMajor, CblasTrans, CblasNoTrans, n, n, m, 1.0, Y, n
      , Q, n, 0, P, n); // Y'*Q = P
```

Consider the simple code snippet in Listing 2.1, which we will utilize to illustrate the salient technical details of the Lanczos and LOBPCG algorithms and our optimization ideas. This code snippet includes three of the most common sparse solver kernels. Suppose that the input matrix A is of size m×m, block size is denoted by b, and the vector width by $n \geq 1$:

- SpMM kernel is partitioned into tasks where each operates on a b×b block of the matrix A, and b×n block of X and Y as shown in Figure 2.1. Tasks are created only for non-empty blocks.

- In the second kernel, which is a linear combination operation and will be referred to as the XY kernel, each task operates on a b×n block of Y, the entire Z matrix (a single n×n block) and b×n block of Q.

- The third one is the inner product kernel, which will be referred to as XTY kernel, spawns tasks as shown in Figure 2.2, computing partial results from the multiplication of n×b block of Y^T and b×n block of Q. A final task reduces the partial results.

Notice that there are two different ways to implement the task-parallel SpMM kernel: (i) to launch all SpMM tasks asynchronously and keep a partial output vector on each thread/core, or (ii) to setup dependencies between tasks to ensure no two threads/cores access the same portion of the output vector. With the latter option, the maximum degree of concurrency in SpMM is still equal to the number of blocks in the output vector. Therefore, as long as the number of blocks in the output vector is greater than the number of threads, it avoids the memory cost and processing overhead of the reduction required for the first option. As shown in Section 2.4, the degree of parallelism yielded by the optimal block sizes exceeds the thread count for DeepSparse and HPX. Experimental results on Regent pointed out that the dependency based solution achieves better performance than the buffer based solution even when there is not enough tasks to keep all threads occupied. As a result, we adopt the latter approach in all three frameworks.

With this crucial detail in mind, for $b = m/3$, the computational DAG for Listing 2.1 is shown in Figure 2.3. Regardless of the underlying representation of the DAG by the runtime system, the correctness of computation depends on a valid execution order with respect to the DAG topology, whereas the performance relies on exploiting the maximum parallelism available while determining a schedule that reduces data movement. Next, we discuss how each framework accomplishes these conflicting goals in detail.

2.2.1 DeepSparse

DeepSparse [1] consists of two major components:

2.2.1.1 Primitive Conversion Unit (PCU)

PCU essentially provides a high level front-end scientific application development. Task Identifier (TI), the first subcomponent of PCU, parses GraphBLAS [31] and BLAS/LA-

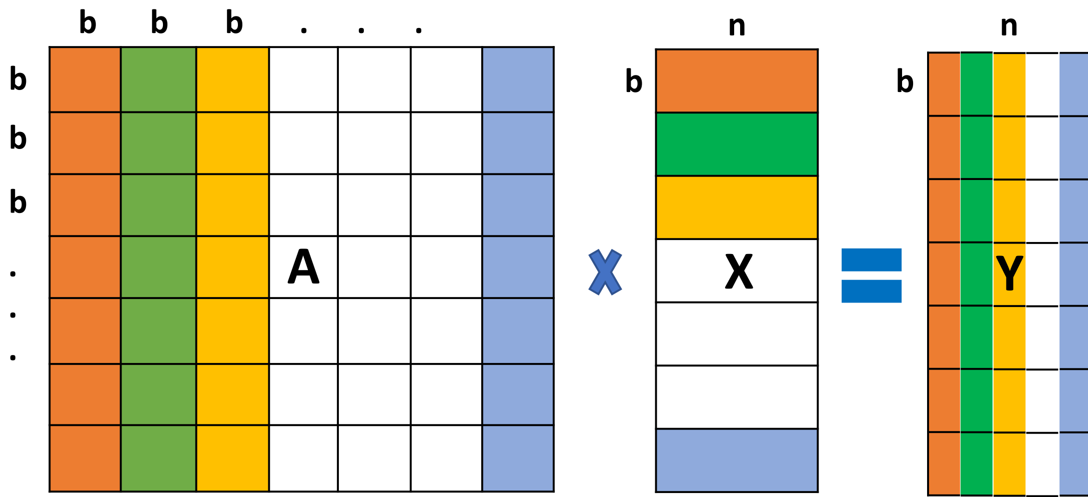


Figure 2.1 Task partitioning of the SpMM kernel.

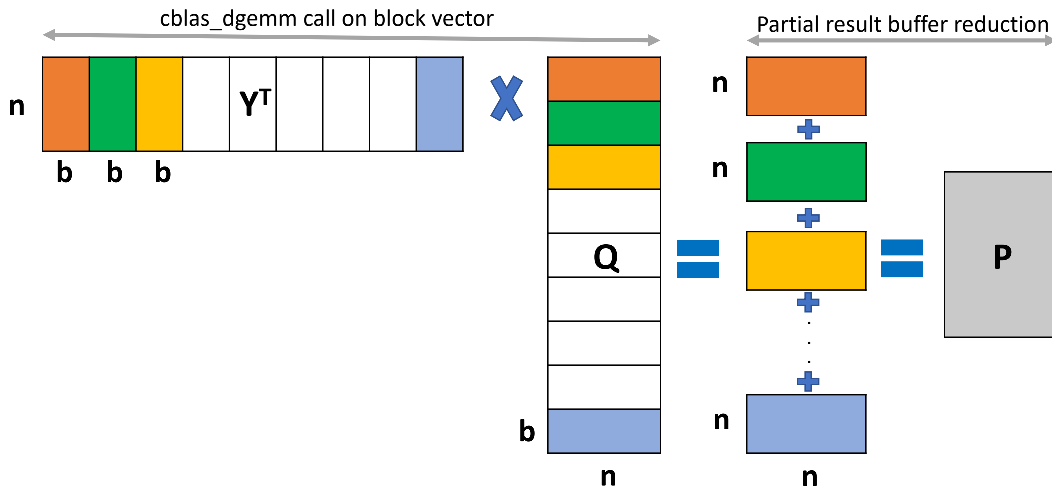


Figure 2.2 Task partitioning of the inner product kernel.

Figure 2.3 Task graph for the pseudocode in Listing 2.1.

PACK [6, 37] function calls, which are similar to those in Listing 2.1, expressed through DeepSparse API. The output of TI is a dependency graph at the function call level whereas tasks must be created at a much finer granularity to expose parallelism and to allow control over data movement. Task Dependency Graph Generator (TDGG), the second subcomponent, accomplishes that by going over the input/output data information generated by TI for each function call and decomposing corresponding data structures. TDGG then generates the dependencies between individual fine-granularity tasks by examining the function call dependencies determined by TI, while taking into consideration the non-zero pattern of the sparse matrix.

2.2.1.2 Task Executor

DeepSparse provides the OpenMP task-based implementation of all computational kernels it supports. As such, Task Executor picks each node from the output of TDGG one by one and extracts the corresponding task information. Based on the kernel id, partition id of the input/output data structures and other required parameters, Task Executor calls the corresponding function found in the DeepSparse library, effectively spawning an OpenMP task. In DeepSparse, the master thread spawns all OpenMP tasks in a depth-first topological order, and relies on OpenMP's default task scheduling algorithms for execution of these tasks.

DeepSparse will explicitly generate the task dependency graph for each algorithm and input sparse matrix combination but the overhead of this graph generation is negligible for two reasons. First, each vertex in the graph corresponds to a task operating on a large set of data in the original problem. Secondly, sparse solvers are typically iterative, and the same task dependency graph is used for several iterations. Therefore, such overhead would be insignificant in comparison to the actual problem size. This is also why we will not account for this overhead when making a performance comparison.

2.2.2 HPX

HPX attains asynchronous parallelism through asynchronous function execution (*async*) and *future* instances: Asynchronous execution of a function will result in scheduling of it as a new HPX thread and return a new *future* instance as HPX threads in the queue are dynamically managed by the runtime system [29]. A *dataflow* object, on the other hand, triggers a predefined function when a set of futures become ready. Combining a *dataflow* object with asynchronous execution provides a powerful mechanism for maintaining data dependencies and constructing an execution tree. We show HPX's dataflow model in Listing 2.2, which implements the pseudocode in Listing 2.1.

Each *future* defined (line 1-4) indicates whether a task of *void* is executed, and thus the futures are of *void* type as well. We have four functions that are not given in the code snippet: (i) `SpMM` for a single block of the matrix, (ii) `f_dgemm` and (iii) `f_dgemm_t` that are wrapper functions for `cblass_dgemm` calls to execute XY and XTY tasks, and (iv) `reduce_buffer` to accumulate partial results of P. We define a proxy function for each of these four functions (line 9-12) whose sole purpose is to unwrap the futures when ready before passing to the actual function. That enables programmers to write functions with the same ease as the equivalent sequential code.

We launch an asynchronous SpMM task (line 17) that operates on sparse matrix block $A_{i,j}$ and block X_j to update block Y_i. The *dataflow* returns a *future* to the result of the SpMM task, which is assigned to $Y_ftr[i]$ (line 17). Since the use of a buffer for the output Y is avoided through a dependency based approach, this *future* depends on itself. To compute block Q_i within the XY kernel, the computation of Y_i should be finished, which is indicated by the readiness of $Y_ftr[i]$ (line 20). Likewise, in the XTY kernel, the task responsible for ith buffer of P will be triggered when both $Y_ftr[i]$ and $Q_ftr[i]$ are ready (line 23).

Note that checking $Y_ftr[i]$ is redundant there as $Q_ftr[i]$ already depends on $Y_ftr[i]$. HPX allows a vector of futures being provided as a parameter in *dataflow* to set the dependencies; we use this feature for P: `reduce_buffer` will be invoked once every *future* in

vector *P_prtl_ftr* is ready (line 24). Last but not least, we skip the empty matrix blocks (line 16) since they do not contribute to the output, in order to lighten the burden on the runtime system and improve the performance.

Listing 2.2 HPX code for the pseudocode in Listing 2.1.

```
1  std::vector<hpx::shared_future<void>> Y(np);
2  std::vector<hpx::shared_future<void>> Q(np);
3  std::vector<hpx::shared_future<void>> P_prtl_ftr(np);
4  hpx::shared_future<void> P_rdcd_ftr;
5  // np (number of partitions) = ceil(m/blocksize)
6  for(int i = 0; i != np; ++i)
7      Y_ftr[i] = hpx::make_ready_future();
8  // to unwrap futures passed to functions
9  auto OpSpMM = hpx::util::unwrapping(&SpMM);
10 auto OpDGEMV = hpx::util::unwrapping(&f_dgemm);
11 auto OpDGEMV_T = hpx::util::unwrapping(&f_dgemm_t);
12 auto OpRed = hpx::util::unwrapping(&reduce_buf);
13 // Y = A * X
14 for(i = 0; i != np; ++i)
15     for(int j = 0; j != np; ++j)
16         if(A[i * np + j].nnz > 0)
17             Y_ftr[i] = hpx::dataflow(hpx::launch::async, OpSpMM, Y_ftr[
                   i], A, X, Y, i, j);
18 // Q = Y * Z
19 for(i = 0; i != np; ++i)
20     Q_ftr[i] = hpx::dataflow(hpx::launch::async, OpDGEMV, Y_ftr[i], Y,
           Z, Q, i);
21 // P = Y' * Q
22 for(i = 0; i != np; ++i)
23     P_prtl_ftr[i] = dataflow(hpx::launch::async, OpDGEMV_T, Y_ftr[i],
           Q_ftr[i], Y, Q, Pbuf, i);
24 P_rdcd_ftr = dataflow(hpx::launch::async, OpRed, P_prtl_ftr, Pbuf, P);
```

2.2.3 Regent

Regent is a language, runtime system, and a compiler that exerts implicit dataflow parallelism through two key abstractions: *tasks* and *logical regions* (or simply *regions*) [52]. *Tasks* are functions that are marked as eligible for parallel execution by the programmer and *regions* are collections of structured objects that can be recursively partitioned to render parallel execution possible. *Tasks* in Regent are forced to describe how they interact

with each *region* they take as argument by declaring *privileges*: *read, write, read/write* or *reduce*, which in return allows Regent to discover parallelism in seemingly sequential code. To illustrate this, in Listing 2.3, we provide the Regent code of the pseudocode presented in Listing 2.1.

A *region* (line 16-17) is the cross-product between an *index space* (lines 14-15) and a field space (lines 1-3, like *structs* in C). A *region* can be disjointly partitioned into *subregions* with a simple use of *partition* function (line 20). An *index space* (line 19) must be provided to name the respective *subregions*. CSB format requires each block (*subregion* in Regent) to dynamically allocate memory based on their number of non-zero entries, but Regent does not allow such allocation. As a workaround, we create a *region* that contains all entries (line 14&16) in advance where the entries falling into the same block are kept contiguous to better utilize the cache.

SpMM task implementation is similar to that of HPX, but here, rather than passing the pointer to the entire X and Y data and making sure we access and update the appropriate portion, we directly pass the corresponding *subregion* (i.e., X_j and Y_i in line 26). By analyzing the *privileges* defined on each passed *region/subregion* (line 8), the runtime system extracts parallelism for SpMM tasks as shown in Figure 2.3. Moreover, the index launch represents a loop of tasks that are non-interfering and is a compiler-level optimization. This concept helps the Regent to launch those tasks without any dependency checks. It is not required, but the programmer can use it to ensure the implementation is sound (line 31&36), for `f_dgemm` and `f_dgemm_t` tasks (line 33&38), for instance. Although we do not share it due to space constraints, `f_dgemm` declares *read privilege* on Y_i and Z and *write privilege* on Q_i. Slightly different from HPX, `f_dgemm_t` declares *reduce privilege* on P, which is convenient in contrast to using *reduce* on Y_i for SpMM since P is a much smaller matrix with a lower overhead.

Listing 2.3 Regent code for the pseudocode in Listing 2.1.

```
1  fspace csb_entry{
2      {rloc, cloc}: uint16, val: double,
3  }
4  task SpMM(rA: region(ispace(int1d), csb_entry),
5              rX: region(ispace(int1d), double),
6              rY: region(ispace(int1d), double),
7              s: int, e: int)
8  where reads(rA, rX), reads writes(rY) do
9      -- ... (SpMM implementation)
10 end
11 -- ... (other tasks)
12 task main()
13     -- ... np (num partitions) = ceil(m/blksize)
14     var sparse_matrix_is = ispace(int1d, nnz)
15     var vector_block_is = ispace(int1d, m * n)
16     var Alr = region(sparse_matrix_is, csb_entry)
17     var Xlr = region(vector_block_is, double)
18  init)
19     var part = ispace(int1d, np)
20     var Xlp = partition(equal, Xlr, part)
21     -- ... (Y and Q partitionings, etc.)
22     -- Y = A * X
23     for i = 0, np do
24         for j = 0, np do
25             if blkptrs[i*np+j] < blkptrs[i*np+j+1] then
26                 SpMM(Alr, Xlp[j], Ylp[i], blkptrs[i*np+j], blkptrs[i*np
                       +j+1])
27             end
28         end
29     end
30     -- Q = Y * Z
31     __demand(__index_launch)
32     for i = 0, np do
33         f_dgemm(Ylp[i], Zlr, Qlp[i], m, n, blksize, i)
34     end
35     -- P = Y' * Q
36     __demand(__index_launch)
37     for i = 0, np do
38         f_dgemm_t(Ylp[i], Qlp[i], Plr, m, n, blksize,i)
39     end
```

2.3 Benchmark Applications

We evaluate the performance of the task parallel frameworks on two popular eigensolvers with different characteristics: Lanczos [36], which is SpMV based, and Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) [33], which is SpMM based.

Lanczos computes the k algebraically largest (or smallest) eigenvalues of a symmetric matrix by building the Krylov subspace Q, a block of orthogonal vectors. As $k \ll m$, it is a relatively simple algorithm (see Alg. 2.1) where SpMV is the main kernel. We give

Algorithm 2.1 Lanczos Algorithm.

1: $b \leftarrow initial\ vector$
2: $Q_0 \leftarrow b/||b||_2$
3: **for** $i = 1$ to k **do**
4: $\quad z \leftarrow AQ_{i-1}$
5: $\quad \alpha_i \leftarrow {Q_{i-1}}^T z$
6: $\quad q \leftarrow [Q_0, \ldots, Q_{i-1}]$
7: $\quad z \leftarrow z - qq^T z$
8: $\quad \beta_i \leftarrow ||z||$
9: $\quad Q_i \leftarrow z/\beta_i$
10: **end for**

the pseudocode for the LOBPCG solver in Alg. 2.2. It involves kernels with much higher arithmetic intensities (such as SpMM and several level-3 BLAS calls) compared to Lanczos. The total memory needed for block vectors Ψ, R, Q and others can easily exceed the space matrix $\widehat{H}$ takes up. [1] shows a sample task graph for LOBPCG for a toy problem, which demonstrates the difficulty of creating a schedule to attain an efficient execution.

DeepSparse uses the DAG constructed for a single iteration with barriers in between because the number of iterations until convergence is unknown. HPX and Regent form the DAG internally on-the-fly, so they might proceed between iterations without a barrier, but this is hard to achieve in practice due to the convergence check at each iteration. Critical path lengths in Lanczos and LOBPCG are 5 and 29, respectively. Number of tasks depends on block and matrix sizes, and ranges from 56 to 6,570,446 per iteration.

Algorithm 2.2 LOBPCG Algorithm solving $\hat{H}\Psi = M\Psi$.

Input: $\hat{H}$, matrix of dimensions $m \times m$
Input: Ψ_0, a block of vectors of dimensions of $m \times n$
Output: Ψ and M such that $\|\hat{H}\Psi - \Psi M\|_F < \epsilon$, and $\Psi^T \Psi = I_n$
1: Orthonormalize the columns of Ψ_0
2: $Q_0 \leftarrow 0$
3: **for** i = 0, 1, . . . , until convergence **do**
4: $M_i \leftarrow \Psi_i^T \hat{H} \Psi_i$
5: $R_i \leftarrow \hat{H}\Psi_i - \Psi_i M_i$
6: Apply the Rayleigh–Ritz procedure on span$\{\Psi_i, R_i, Q_i\}$
7: $\Psi_{i+1} \leftarrow \underset{V \in \text{span}\{\Psi_i . R_i, Q_i\},\ V^T V = I_n}{\text{argmin}} \text{trace}(V^T \hat{H} V)$
8: $Q_{i+1} \leftarrow \Psi_{i+1} - \Psi_i$
9: **end for**
10: $\Psi = \Psi_{i+1}$

2.4 Performance Evaluation

Performance evaluations were carried out on two systems, an Intel Broadwell-based multicore cluster and an AMD EPYC-based manycore cluster. Each Broadwell cluster node has two 14-core Intel Xeon E5-2680v4 Broadwell 2.4 GHz processors with a 64 KB L1 cache (32 KB instruction, 32 KB data) and a 256 KB L2 cache per core, in addition to a 35 MB shared L3 cache. Each EPYC cluster node has two 64-core AMD EPYC 7H12 2.6 GHz processors. Each EPYC core has a 64 KB L1 cache (32 KB instruction, 32 KB data), a 512 KB L2 cache and 16 MB L3 cache is shared between every four cores.

We compare the performance of DeepSparse, HPX and Regent implementations with two library-based BSP versions: (i) **libcsr** is the implementation of the benchmark solvers using thread-parallel Intel MKL Library calls (including SpMV/SpMM) with CSR storage of the sparse matrix, (ii) **libcsb** also uses Intel MKL calls, but with the matrix being stored in the CSB format. We use MKL calls within each task of the AMT models whenever possible for a fair comparison. Finally, we do not claim to have the best possible implementation for all solver versions, although we did our best to optimize them.

We utilized an entire node for our runs on both clusters, i.e., 28 cores on Broadwell and 128 cores on EPYC. For DeepSparse, libcsr and libcsb, we bind OpenMP threads to cores.

For HPX, the number of OS threads spawned through *–hpx:threads* argument is the same as the number of cores. For Regent, the number of cores to be used for executing the application tasks is specified using *-ll:cpu*, and *-ll:util* determines the number of cores allocated to the runtime system. Empirically we found that *-ll:cpu 24 -ll:util 4* on Broadwell and *-ll:cpu 110 -ll:util 18* on EPYC yield (near-)optimal results on both benchmark applications.

We selected 14 matrices with varying sizes, sparsity patterns, and domains from the SuiteSparse Matrix Collection in addition to the Nm7 matrix, which is from a nuclear shell model code (see Table 2.1) [17]. Since both solvers require the input matrix to be symmetric, the matrices that are not symmetric (shown in bold in Table 2.1) are made so by copying the transpose of the lower triangular part over the upper triangular part: $A_{new} = L + L^T - D$. Matrices shown in italics were originally binary matrix, and hence were filled with random values without breaking the symmetry.

Matrix	**#Rows**	**#Non-zeros**
inline1	503,712	36,816,170
dielFilterV3real	1,102,824	89,306,020
Flan_1565	1,564,794	117,406,044
HV15R	2,017,169	281,419,743
Bump_2911	2,911,419	127,729,899
Queen4147	4,147,110	329,499,284
Nm7	4,985,422	647,663,919
nlpkkt160	8,345,600	229,518,112
nlpkkt200	16,240,000	448,225,632
nlpkkt240	27,993,600	774,472,352
it-2004	41,291,594	1,120,355,761
twitter7	41,652,230	868,012,304
sk-2005	50,636,154	1,909,906,755
webbase-2001	118,142,155	1,013,570,040
mawi_201512020130	128,568,730	270,234,840

Table 2.1 Matrices used in our evaluation.

All presented performance data come solely from the solver iteration parts, excluding any I/O, initialization and setup parts. Performance data were averaged over multiple iterations (20 for Lanczos, 10 for LOBPCG). For the last two matrices, number of iterations was 10 for Lanczos and 5 for LOBPCG due to their size. Our comparison criteria are L1, L2, LLC

(L3) misses (unavailable on EPYC due to root access requirement) and execution times for both solvers and architectures. Cache misses were normalized with respect to that of libcsr, and speedups were calculated over libcsr. Cache miss data was obtained using "perf stat" command.

We note that performance data from where a single socket is used on both architectures (14 cores on Broadwell and 64 cores on EPYC) are similar to the case presented here, where both sockets are used. The only difference is, on EPYC, the task parallel frameworks seem to be affected less by the NUMA-related performance issues as their speedup numbers improve going from a single socket to the entire node. The single socket results are not presented due to space constraints.

The impact of the degree of parallelism is measured by conducting tests for several different block sizes. However, for a given block size, all AMT models are essentially presented the same DAG, i.e., the available degree of parallelism are identical across all runtimes. Since all runtimes are executing the same DAG, we believe their performance differences are due to the different scheduling algorithms employed as this directly impacts thread idling and cache utilization.

Scheduling policies of the runtimes studied here are either opaque or are not well documented, making a detailed comparison of the impact of the different scheduling policies difficult. For instance, OpenMP's task scheduling is left to the implementation and not well documented; task priorities are only ignorable hints. For HPX, NUMA-aware scheduling made a big difference in cache utilization and performance, but their scheduling algorithms are not well documented either. Regent gives task mapping options, which is, however, mainly recommended for heterogeneous computing.

In Section 2.4.2 and 2.4.3, we share the results from the experiments where the optimal block size is employed, which depends on the solver, architecture, runtime system and matrix type. Then in Section 2.4.4, we present a practical rule of thumb for determining the ideal task granularity by choosing the ideal block size for each runtime system.

2.4.1 The Effect of Optimizations

We tried to optimize the code at the same level for each task parallel system using the techniques discussed below. Although each framework benefits from all optimizations, due to space constraints we only share the results of every optimization for a certain framework, solver and architecture combination where the impact is the most evident. In all following optimization plots, compared implementations incorporate all other optimization techniques so that the only variable is the selected optimization at hand.

2.4.1.1 First-Touch Placement Policy

This policy refers to allocation of a data page in the memory closest to the thread accessing it first. When a single thread initializes all data structures, the data ends up residing in the memory of a single NUMA node, which increases access times, consequently hurting the performance. Leveraging this policy would simply require the initialization of vector blocks and the sparse matrix in parallel in the case of sparse solvers. The 128 cores on an EPYC node are internally organized into 8 NUMA subregions, 4 per socket. As shown in Figure 2.4 for DeepSparse, this optimization is vital for good performance (up to 2.5 fold) for the small and mid-sized matrices on the EPYC system. We also utilize it in the BSP versions (libcsr and libcsb) for a fair comparison.

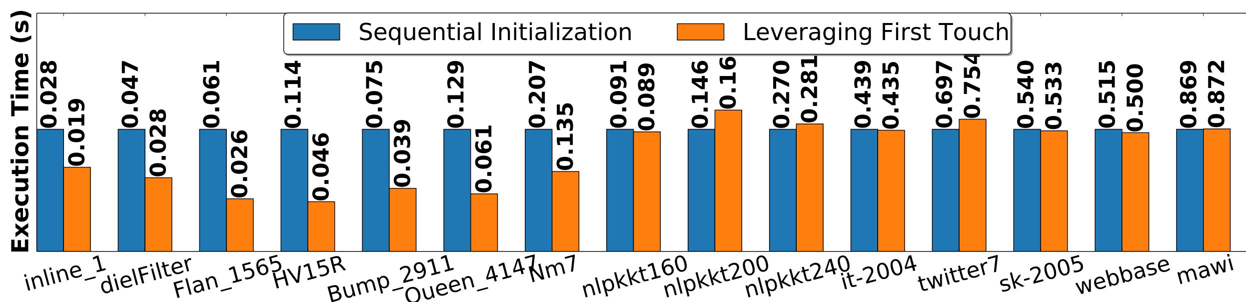


Figure 2.4 Execution time of DeepSparse for Lanczos on EPYC wrt first-touch policy.

2.4.1.2 Skipping Empty Tasks

Depending on the sparsity pattern of a matrix and chosen CSB block size, there are empty blocks which do not contribute to the result of SpMV/SpMM operation. Spawning

tasks for those blocks creates scheduling overheads for the runtime system. Figure 2.5 shows that skipping such tasks may speed up the execution time by 30% on average, albeit not as effective on some matrices. We attribute the lack of improvement in those cases to the fact that both implementations use the optimal block size for each matrix, which in general does not yield too many empty blocks.

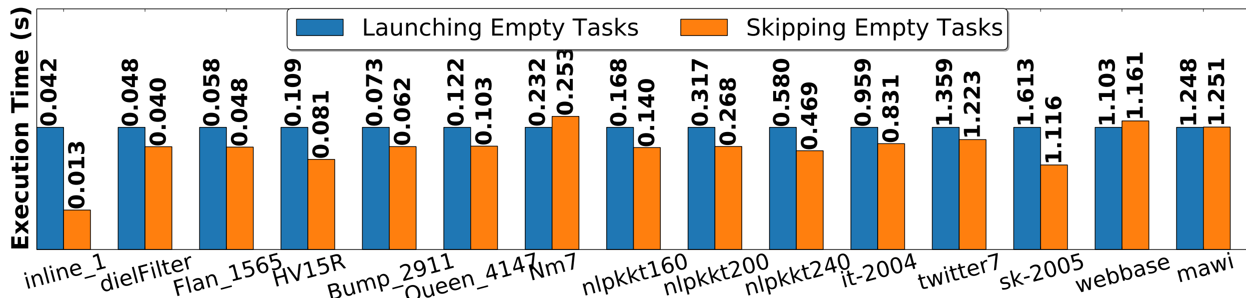


Figure 2.5 Execution time of HPX for Lanczos on Broadwell wrt skipping empty tasks.

2.4.1.3 Eliminating Reduction for SpMV/SpMM Output

As discussed in Section 2.2, the dependency based approach is used on all frameworks to eliminate the reduction overheads. In Figure 2.6, we show the empirical advantage of this decision on Regent for the LOBPCG algorithm. We observe that the reduce-based approach yields an extremely poor performance on large matrices, and we believe this is due to large buffers that need to be allocated by each core. Furthermore, Regent runtime system manages the reduce operation internally (recall that one of the *region* privileges was *reduce*). Given the problematic scaling behavior of Regent with regard to the number of tasks (discussed in Section 2.4.4), poor performance of the reduce based approach on Regent is not a surprise.

2.4.1.4 Other Attempts

We also attempted several framework-specific optimizations. For instance, HPX allows passing scheduling hints to their scheduler in an effort to create executors that target a specific NUMA domain or to pin HPX threads to a particular core. We employed scheduling hints to achieve a locality-aware scheduling for both solvers. This technique improved HPX's both Lanczos and LOBPCG performance significantly on EPYC, where there exist

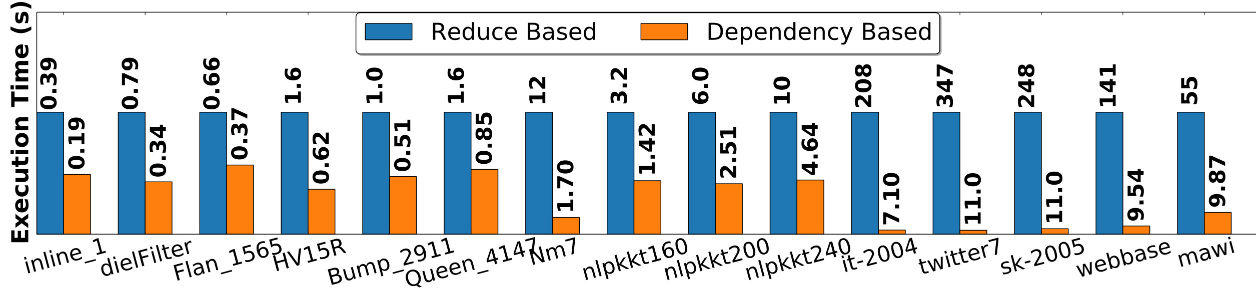


Figure 2.6 Execution time of Regent for LOBPCG on Broadwell wrt two SpMV/SpMM computation approaches.

8 NUMA domains (16 cores each). Regent provides a technique called *dynamic tracing* [38] to reduce the task management overhead for iterative solvers. This technique relies on capturing the task graph in the first iteration and replaying it for subsequent iterations through memoization to avoid the dependence analysis. However, this last attempt did not yield any significant performance improvement.

2.4.2 Lanczos Evaluation

Lanczos algorithm is a relatively simple algorithm in the sense that it has much fewer types and number of tasks than LOBPCG because it essentially consists of one SpMV and one inner product kernel at each iteration. As such, scheduling decisions are simpler and there are fewer data reuse opportunities. Consequently, we observe that the task parallel systems often lead to little to no improvement in terms of cache misses. This can be seen in Figure 2.7 where the cache misses comparison on EPYC for different Lanczos versions is shown. No framework achieves consistent reduction in cache misses on L1 level. Moreover, the improvements on L2 level can be attributed to the matrices being stored in the CSB format since libcsb, the other BSP version, yields similar improvements.

Most importantly, all three task-parallel versions give decent speedups on both architectures as shown in Figure 2.8. On Broadwell (the top subplot), DeepSparse, HPX and Regent achieve up to 2.3×, 4.3× and 2.0× improvement although on average, the speedup achieved is somewhat modest (1.5×, 2.2× and 1.1×, respectively). Task parallel versions perform better when we go from a multicore (Broadwell) to a manycore (EPYC) architec-

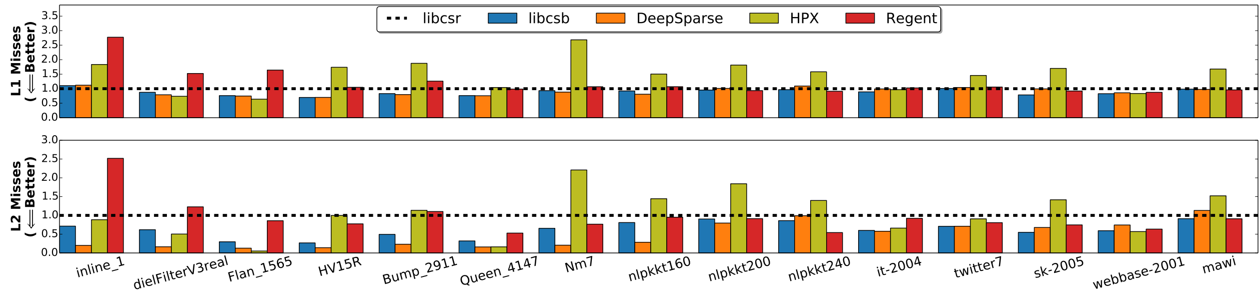


Figure 2.7 L1 and L2 misses of different Lanczos versions on EPYC normalized wrt libcsr.

ture. DeepSparse achieves as high as 6.5× speedup, HPX up to 9.9× speedup and Regent up to 2.7× speedup. On average, DeepSparse, HPX and Regent achieve 3.3×, 4.9× and 1.6× speedup, the majority of which comes from the large matrices.

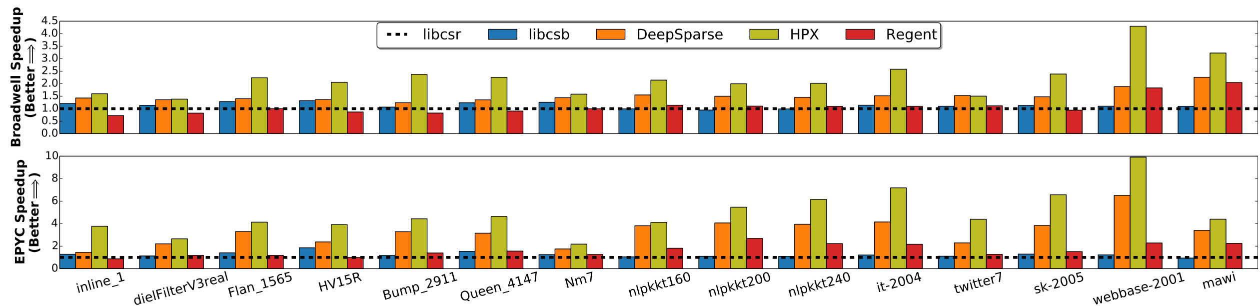
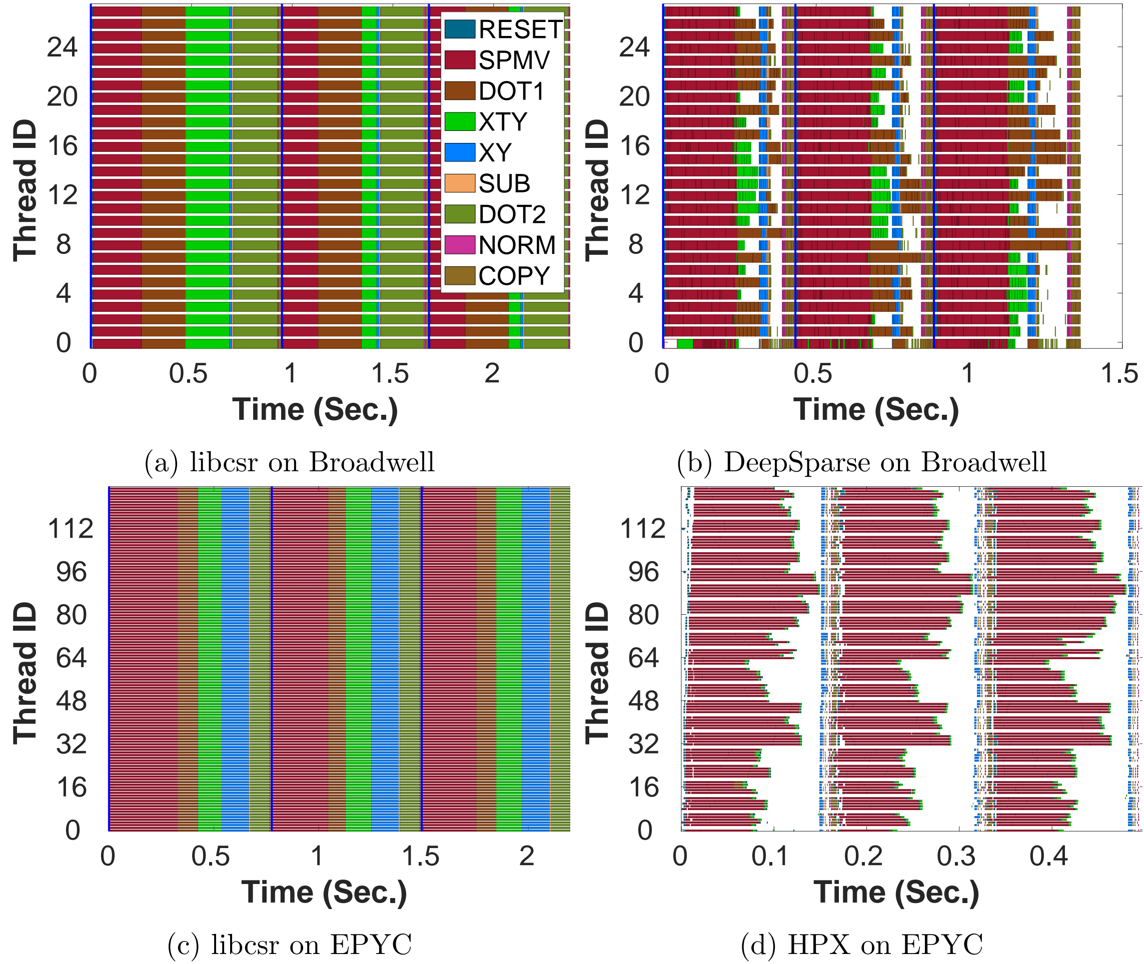


Figure 2.8 Speedup of different Lanczos versions on Broadwell (top) and EPYC (bottom) over libcsr.

We attribute the speedups observed across the board to the increased parallelism with tasking and reduced synchronization overheads. In fact, a further investigation of the execution flow graph of tasks in Figure 2.9 shows that the manycore architecture provides a greater level of parallelism for the task parallel systems to fill the gap resulting from load imbalances of SpMV with the succeeding tasks. Therefore, each iteration is completed not long after the execution of the last SpMV task on EPYC, providing the AMT approaches with a greater success.

2.4.3 LOBPCG Evaluation

LOBPCG is a complex algorithm with several different kernel types; its task graph may result in millions of tasks depending on the block size. In LOBPCG, vector blocks have only 8-16 columns, hence there is no tiling in the column dimension. Block sizes refer to



(a) libcsr on Broadwell

(b) DeepSparse on Broadwell

(c) libcsr on EPYC

(d) HPX on EPYC

Figure 2.9 Execution flow graph of nlpkkt240 from first three iterations of Lanczos for different versions and architectures.

the number of rows in each chunk, they ranged from 1K to 16M. Since LOBPCG requires several vector operations consecutively, there are plenty of data reuse opportunities for vector chunks.

In Figure 2.10, we show the cache misses and speedup comparison on Broadwell for all five LOBPCG versions. The libcsr and libcsb versions achieve similar number of cache misses, while the task-parallel versions demonstrate an outstanding cache performance:

- DeepSparse yields a consistent improvement throughout all cache levels: It achieves 3.0× - 10.4× fewer L1 misses, 3.8× - 12.0× fewer L2 misses and 1.4× - 4.7× fewer L3 cache misses than libcsr.

- HPX's cache performance is on par with DeepSparse: It achieves 2.8× - 13.7× fewer L1 misses, 3.7× - 13.1× fewer L2 misses and 1.4× - 5.2× fewer L3 cache misses than libcsr.

- Regent has competitive cache utilization too, as it produces 4.3× - 9.6× fewer L1 misses, 4.0× - 12.3× fewer L2 misses and 1.6× - 6.2× fewer L3 cache misses compared to libcsr.

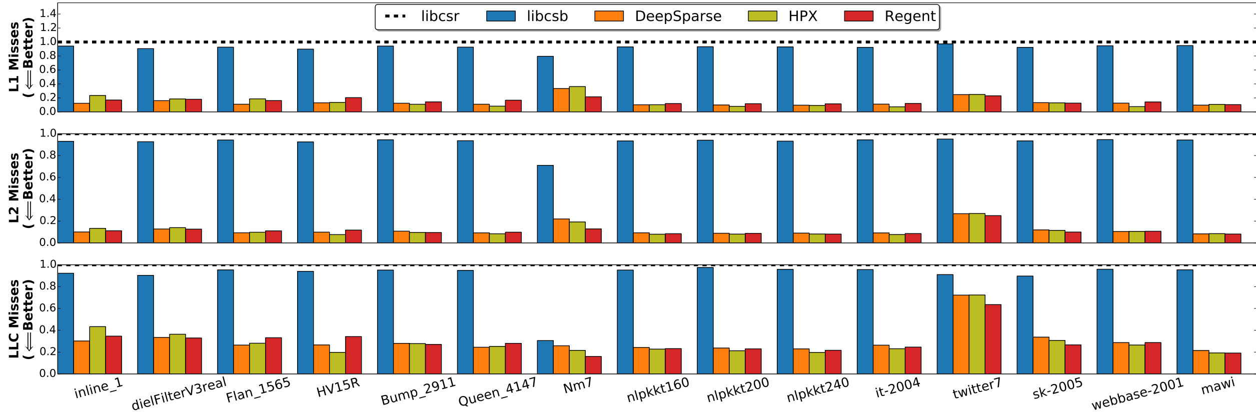


Figure 2.10 L1, L2 and LLC (L3) misses of different LOBPCG versions on Broadwell normalized wrt libcsr.

As the top subplot of Figure 2.11 shows that on Broadwell, even with the implicit task graph creation and execution overheads of the runtime systems, this significant reduction in cache misses leads to 1.8× - 3.0× speedup for DeepSparse, 1.5× - 4.4× speedup for HPX and 0.8× - 1.9× speedup for Regent (slowdown occurring on a few smaller matrices) over the execution times of libcsr. Given the highly complex underlying task dependency graph 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 but use the same data structures.

AMT models continue their superior performance in terms of execution time on EPYC as shown in the bottom subplot of Figure 2.11. As a matter of fact, DeepSparse and HPX improve their performance further compared to Broadwell: DeepSparse achieves 1.2× - 5.5×

speedups and HPX achieves 1.7× - 7.5× speedups over libcsr. However, Regent demonstrate a similar performance on this architecture achieving 0.8× - 2.3× speedup where the performance degradation is again being observed on the smaller matrices.

AMT models achieve up to 99% L1 hits for LOBPCG, compared to the 85-90% hit ratio of loop-parallel versions. Considering AMT models' outstanding cache miss performance as well, we conclude that cache utilization is an important factor with LOBPCG due to data reuse opportunities. The improved performance observed in HPX by switching to NUMA-aware scheduling, which is around 50%, also supports this view.

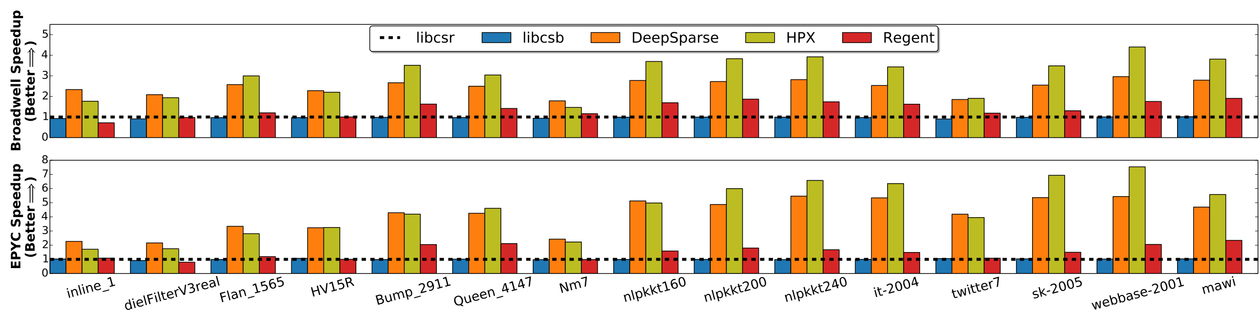
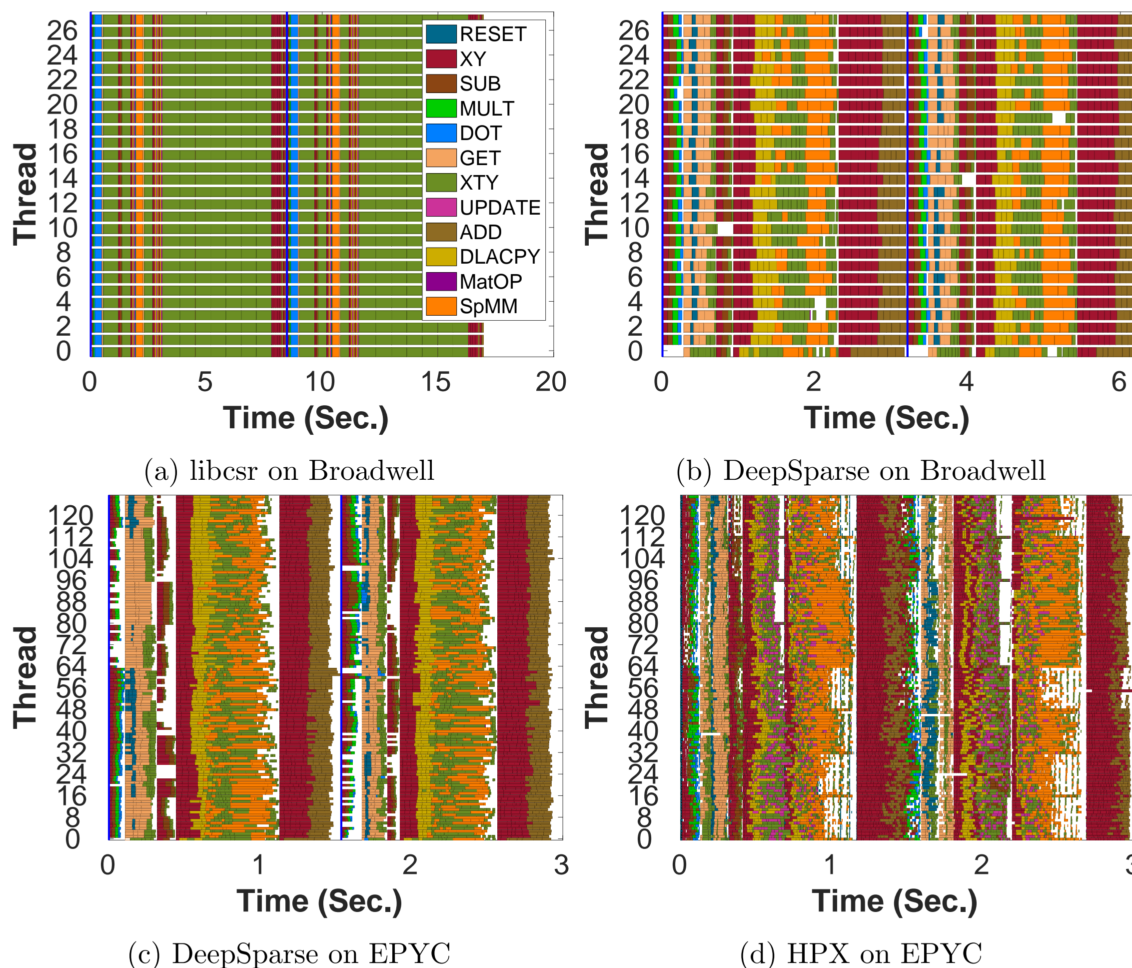


Figure 2.11 Speedup of different LOBPCG versions on Broadwell (top) and EPYC (bottom) over libcsr.

The pipelined execution of tasks in DeepSparse and HPX in comparison to libcsr can be observed in Figure 2.12. The performance on XTY kernel accounts for the main difference in timing (see Figure 2.12a & Figure 2.12b). Data parallel execution of this kernel in the BSP model considerably hurts the performance, which seems to be avoided in task parallel execution to a great extent through the re-use of involved matrix blocks in kernels such as XY or SpMM after the execution of XTY tasks. There also exist characteristic differences within the task parallel models as seen in Figure 2.12c & Figure 2.12d: Both DeepSparse and HPX give the best result on the nlpkkt240 matrix with 256K CSB block size whereas their execution flow graphs do not show much resemblance to each other. HPX in general seems to place less value on prioritization of the tasks that are launched earlier, which consequently produces a more shuffled graph where the overlap in each kernel's start and finish time increases. Regardless of that difference, their execution times are similar in this example

($\approx 3.0sec$).



(a) libcsr on Broadwell

(b) DeepSparse on Broadwell

(c) DeepSparse on EPYC

(d) HPX on EPYC

Figure 2.12 Execution flow graph of nlpkkt240 from two iterations of LOBPCG for different versions and architectures.

2.4.4 Block Size Selection

The CSB block size has a significant effect on the performance of task parallel models as the factors such as task granularity, degree of parallelism, and scheduling overhead of the runtime systems are directly shaped by the block size for a given matrix and solver type. This is because we use this block size as a uniform partitioning factor whether it is for a 2D (e.g., SpMV and SpMM) or 1D (e.g., all vector operations) kernel. Therefore, by the "block count", we simply mean the number of tiles/blocks in each dimension, which is determined by the row count of matrices/vectors and the CSB block size. Choosing a small block size

creates a large number of small tasks, which is preferable on a parallel architecture, but the large number of tasks may lead to significant scheduling overheads. Increasing the block size reduces such overheads, but this may then lead to increased thread idle times and load imbalances. Therefore, finding the sweet spot between these two extremes is important.

We found the optimal block size, which differs for each matrix, solver architecture, and runtime system combination by trying a variety of numbers from 2^{10} to 2^{24}. This brute-force search may not always be practical. Thus, analyzing the data further, we have come to notice that the optimal block size would always yield a block count between 8 and 511 regardless of the case. As such, picking the optimal one boils down to the comparison of the six block sizes, the ones that result in 8 to 15, 16 to 31, 32 to 63, 64 to 127, 128 to 255 or 256 to 511 block counts.

We show the performance profiles in Figure 2.13 to compare these six block counts for each runtime system and architecture. Note that our findings on LOBPCG solver match the ones from Lanczos solver. So, to avoid redundancy, we only share LOBPCG results here. In the performance profiles, we plot the percentages of the instances in which a block count yields an execution time on a matrix that is no longer than τ times the best execution time found by any block count for that matrix. Therefore, the higher a profile at a given τ, the better a heuristic is. We observe the following:

- For DeepSparse, 32-63 block count is the best option and it is always within 1.15× the best option on Broadwell. On EPYC, 64-127 block count have the top spot and 32-63 block count provides a comparable performance.

- For HPX, 64-127 block count gives the optimal performance in general on both Broadwell and EPYC. However, 32-63 and 128-255 block count configurations perform similarly well regardless of the architecture.

- Regent prefers more coarse-grained tasks as 16-31 block count performs the best on both architectures. Considering that bottom three spots belong to highest three options

suggests that Regent has scaling issues with regard to creation or scheduling of large number of tasks. In fact, going beyond 64 block count can cause 5× - 10× slowdowns although we cannot see it here as τ is shown for between 1.0 and 2.0 for all systems.

As a practical rule of thumb, we can say that 32-63 block count on Broadwell and 64-127 block count on EPYC for DeepSparse and HPX are good choices. Even though we have slightly less than a task per thread per kernel with the 64-127 block count, there are many kernels in LOBPCG to be executed in parallel. In fact, the execution flow graphs verify that this kernel-level parallelism might be just enough to keep all threads occupied by exposing more than a task per thread.

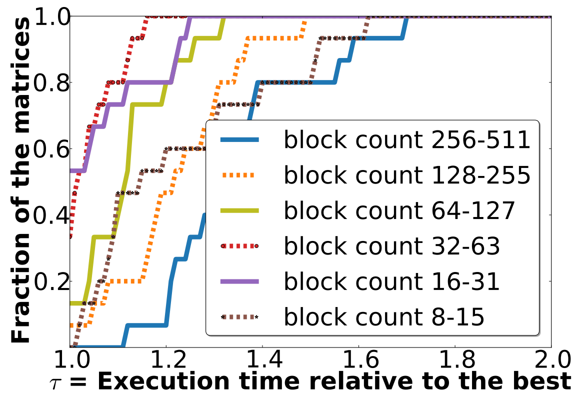
We know from the speedup plots that DeepSparse and HPX are the best two versions by far. Taking into account these block counts, they achieve high performance by over-decomposing the work to yield more than one task per thread for load balancing purposes while limiting that task to thread ratio to avoid scheduling overhead.

Tuning the block size is very important for best results. However, this is a complicated choice that depends on the specific problem, architecture and compiler. We note that even for the easier-to-characterize dense linear algebra kernels, auto-tuning is necessary (e.g., ATLAS library [61]). Hence, for sparse solvers this is a very difficult problem. Nevertheless, we tried to illustrate the trade-offs and give some insights which we hope could be helpful to others.

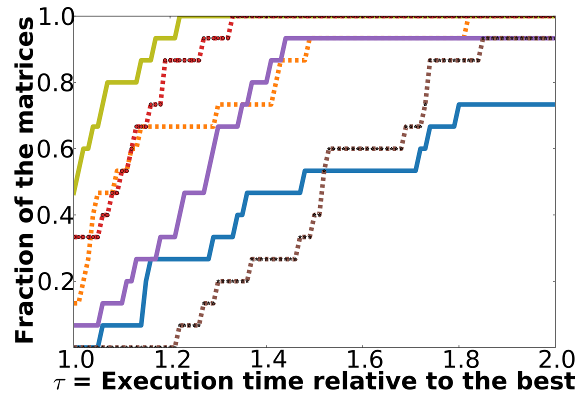
2.5 Conclusion of This Work

Several AMT frameworks emerged as we move towards exascale, and there is a lack of comparative studies, by third party users in particular. We believe a fair evaluation on various application domains would benefit readers in helping them make a well-informed decision in preparing for exascale. This is precisely the motivation for our evaluation of three runtime systems in the context of sparse solvers. To our knowledge, this is the first such comparative work.

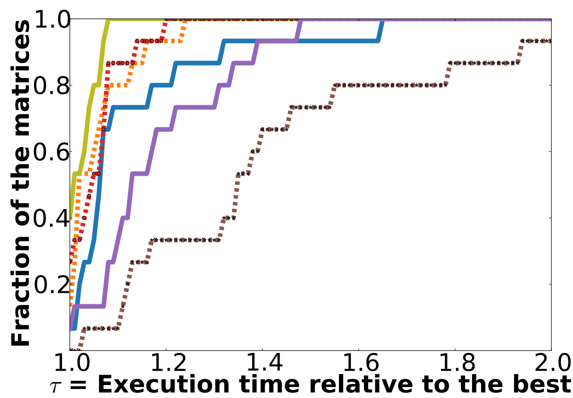
We introduced optimized implementations of LOBPCG and Lanczos eigensolvers using the task-parallel paradigm using three runtime systems: OpenMP (through the DeepSparse



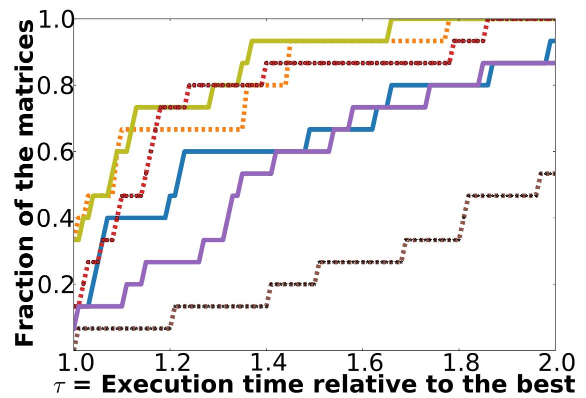
(a) Broadwell DeepSparse



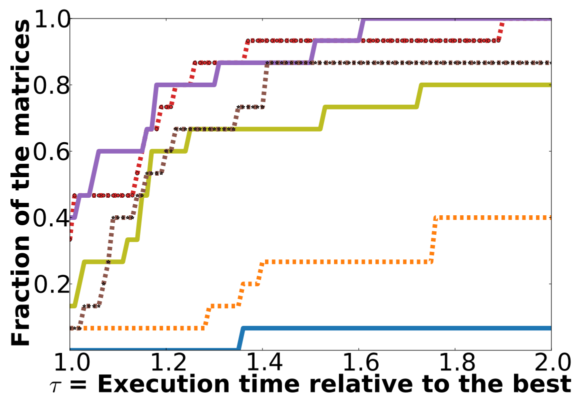
(b) EPYC DeepSparse



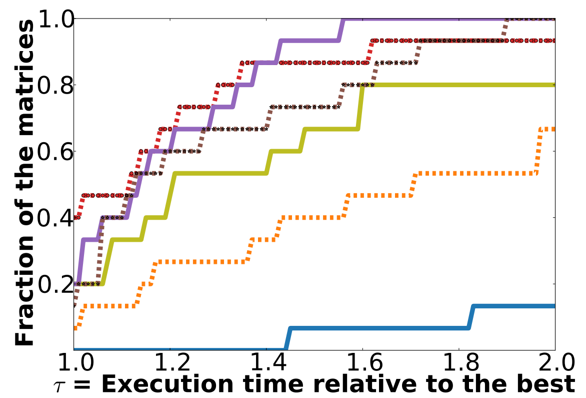
(c) Broadwell HPX



(d) EPYC HPX



(e) Broadwell Regent



(f) EPYC Regent

Figure 2.13 The relative performance of block counts for DeepSparse, HPX and Regent on LOBPCG solver.

framework), HPX and Regent. We show that these task-parallel systems achieve significantly fewer cache misses across different cache layers for LOBPCG, a fairly complex solver. They also provide promising improvements in the execution time over a traditional optimized

BSP implementation for both solvers on two different architectures: Broadwell, an Intel-based multicore system, and EPYC, an AMD-based manycore system. Moreover, they allow achieving such performance improvements without sacrificing the ease of high-level programming.

We conclude that OpenMP tasking and HPX are setting themselves apart from others. OpenMP's great performance is commendable, but so is HPX's because it generates the DAG itself as it goes along and is extensible to distributed memory architectures. Future work will be in the direction of testing HPX in a distributed memory environment using large-scale sparse solvers and graph analytics kernels, and comparing these to hybrid MPI+OpenMP solutions.

CHAPTER 3

PERFORMANCE OF THE HPX FRAMEWORK FOR SPARSE EIGENSOLVERS ON DISTRIBUTED MEMORY ARCHITECTURES

After seeing the success of HPX with our work described in Chapter 2 and given that HPX extends its asynchronous dataflow model to distributed memory environments, our next goal became testing HPX in a distributed memory environment using large-scale sparse solvers and graph analytics kernels, and comparing these to hybrid MPI+OpenMP solutions.

To this end, we have written the distributed Lanczos algorithm using both HPX and MPI+OpenMP programming models. Both implementations are simple in a sense that the matrix is 1D row partitioned, which results in all the communications being global and straightforward. However, the HPX code runs significantly slower than the plain MPI+OpenMP code on average up to 512 AMD EPYC cores.

Moreover, a careful investigation of HPX's performance with respect to the matrix block (tile) size, which dictates the partitioning of the input and output column vectors and a tall-and-skinny matrix used for this algorithm, shows that HPX prefers considerably large block sizes where localities (a locality in HPX's terminology is same as a rank in MPI) and threads clearly suffer from starvation. This behavior contradicts our shared memory results for the same algorithm where HPX achieves high performance by over-decomposing the work to yield more than one task per thread for load balancing purposes.

3.1 Implementation

The pseudocode of the distributed Lanczos algorithm can be seen in Listing 3.1.

Lanczos computes the k algebraically largest (or smallest) eigenvalues of a symmetric matrix by building the Krylov subspace Q, a block of orthogonal vectors. As $k \ll n$, it is a relatively simple algorithm where SpMV is the main kernel. In the distributed implementation of Lanczos, there are four global communication calls: two *all_reduce* calls on the local dot product results (line 18 and 24), one *all_reduce* call on a small vector that is the local output of a linear combination kernel (21), and a rather expensive *all_gather* call (line 15.)

at the end of which each locality gets the global input vector for the SpMV operation. Note that this *all_gather* is needed as the matrix is 1D row partitioned and in order to compute the local output vector of the SpMV, each locality needs the full input vector. As such, all the column vectors $q, z, QQpZ$ and the Krylov subspace Q are naturally 1D partitioned.

Listing 3.1 The pseudocode of the distributed Lanczos algorithm.

```
1  // n = number of rows/columns (of a square matrix)
2  // p = number of MPI ranks or HPX localities
3  // n_local = number of matrix rows a locality owns (roughly n/p)
4  // k = number of largest eigenvalues to be computed
5  // A = n_local × n local sparse matrix
6  // q, z, QQpZ = n_local × 1 column vector
7  // q_global = n × 1 column vector
8  // Q = n_local × k tall-and-skinny matrix
9  // QpZ = k × 1 small vector
10
11 q = a random unit vector
12 Q = q;
13 for(it = 0; it != maxIterations; ++it)
14 {
15     q_global = all_gather(q);
16     z = A*q_global; //SpmV
17     alpha = q^T z;
18     alpha_global = all_reduce(alpha);
19     QpZ = Q^T z; // inner product kernel
20     QpZ_global = all_reduce(QpZ);
21     QQpZ = Q*QpZ_global; // linear combination kernel
22     z = z - QQpZ;
23     beta = ||z||^2
24     beta_global = sqrt(all_reduce(beta));
25     q = z/beta_global;
26     Q = [Q q];
27 }
```

Both MPI+OpenMP (or simply MPI) and HPX implement the pseudocode provided in Listing 3.1. The biggest difference is, however, we adapt a 2D partitioning scheme for HPX where tasks are defined based on the Compressed Sparse Block (CSB) representation of the sparse matrix whereas we simply adapt a fork-join model for the MPI code. HPX code starts by partitioning the sparse matrix into CSB blocks, which are then 1D row block partitioned

among localities. To give an example based on real experiments, when the block size is chosen to be 2^{20} for the matrix *webbase_2001*, which has 118, 142, 155 rows and columns, there will be 113×113 CSB matrix blocks as $ceil(118{,}142{,}155/2^{20}) = 113$. When there are four localities, for instance, first locality will own the first 29×113 CSB blocks whereas the rest will own 28×113 CSB blocks each. Since the column vectors are 1D block partitioned as well, the first locality will get 29×1 blocks of q, z and $QQzP$ where the block size is again 2^{20}.

There are other significant differences between the HPX and MPI code. For example, HPX can overlap the *all_gather* operation in line 16 with the SpMV tasks in line 17 that only need the local blocks of the input vector as the *all_gather* operation is non-blocking. Furthermore, as the SpMV kernel is the most expensive kernel, we try to keep the threads in each locality as occupied as possible by using buffers for the output vector of the SpMV operation. Whenever a buffer is available, a thread will be assigned to an SpMV task to fill in the buffer, assuming that not all tasks are complete. And whenever the block of the output vector which corresponds to that SpMV task is available, meaning there is no task updating it, a thread will add the partial result in that buffer to that block of the output vector. Then, a task will be created to empty that buffer. The full implementation can be seen here.

3.2 Results

We used 5 matrices to assess the performance of the distributed Lanczos algorithm as shown in Table 3.1. All matrices besides $Nm7$ is from the suitesparse matrix collection. Those that are not originally symmetric are made so by copying the transpose of the lower triangular part over the upper triangular part. Also, we assigned random weights to the originally binary matrices.

We conduct experiments on AMD EPYC clusters at HPCC up to 4 nodes. Each EPYC cluster node has two 64-core AMD EPYC 7H12 2.6 GHz processors. Each EPYC core has a 64 KB L1 cache (32 KB instruction, 32 KB data), a 512 KB L2 cache and 16 MB L3 cache is

Matrix	#Rows	#Non-zeros
inline1	503,712	36,816,170
dielFilterV3real	1,102,824	89,306,020
Nm7	4,985,422	647,663,919
twitter7	41,652,230	868,012,304
webbase-2001	118,142,155	1,013,570,040

Table 3.1 Matrices used in our evaluation.

shared between every four cores. We utilize the entire node for our runs, i.e., all 128 cores on EPYC. For HPX, the number of OS threads spawned per locality is the same as the number of cores per locality. The same goes for MPI regarding the number of OpenMP threads per MPI process.

We ran each matrix for HPX with varying CSB block sizes defining the task granularity. In the first plots (Figure 3.1, 3.2 & 3.3) where we compare the performance of HPX and MPI, **we report the execution times that are obtained using the optimal block size in the case of HPX**. On the other hand, MPI uses the CSR matrix format without any blocking and as such, the block size is not a parameter there. For both HPX and MPI, we report the average iteration time over 20 iterations on the first four matrices and over 10 iterations on the last matrix, *webbase-2001*.

In Figure 3.1, 3.2 & 3.3, each row corresponds to a set of results for the same matrix whereas each column corresponds to a set of results for the same processor affinity. In Figure 3.2, for example, as we are using two EPYC nodes and there are 256 cores total, each MPI process or HPX locality will be assigned 16 cores in the case of 16 ranks. The middle column in Figure 3.2 shows the experiment results with such processor affinity.

3.2.1 1-Node Results

We see in Figure 3.1 that HPX is slower than MPI for every matrix and thread configuration combination. Although the slowdown is closer to 5× on *inline_1*, the smallest matrix, it is never higher than 2× on the largest two matrices: *twitter7* and *webbase-2001*. In fact, HPX delivers a competitive performance for certain combinations such as on *twitter7* with 4 ranks.

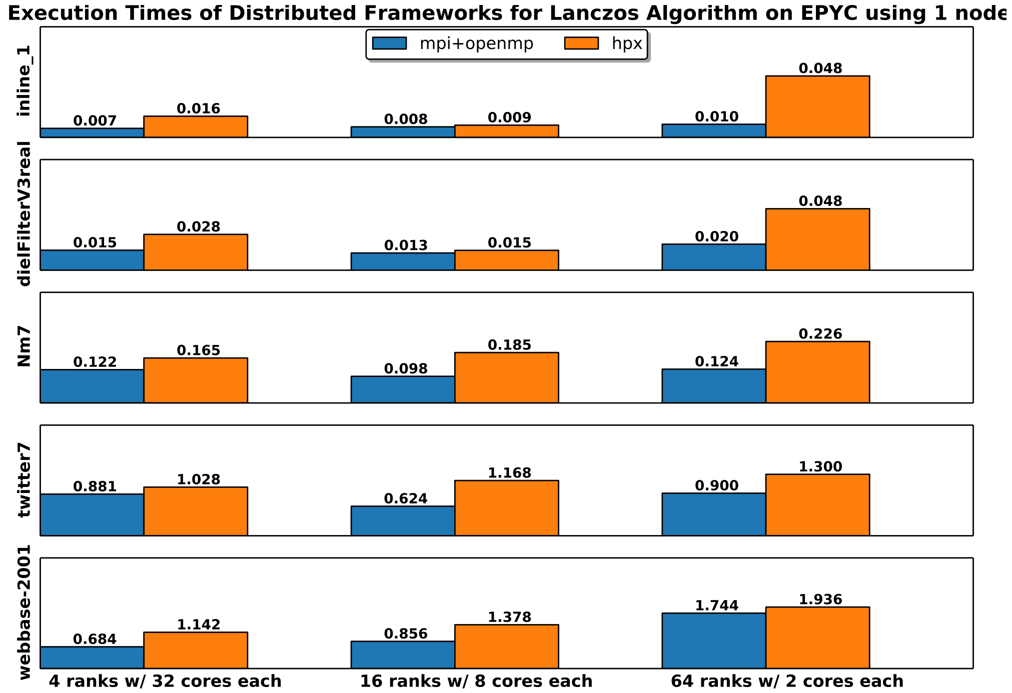


Figure 3.1 Execution times of the distributed Lanczos algorithm on a single EPYC node.

3.2.2 2-Node Results

In Figure 3.2, we see a similar trend to that of the single node experiments. That is, HPX is slower than MPI in 14 out of 15 combination and the difference is more apparent on the small matrices. Among the three rank-core configurations, 64 ranks with 4 cores each is the bottom performing one. Also, neither HPX nor MPI scales well when we go from one node to two nodes in general. For the largest matrix, *webbase-2001*, with 4 ranks, for example, MPI and HPX take 0.68 and 1.14 sec on single node, and 0.66 and 1.28 sec on two nodes, respectively.

3.2.3 4-Node Results

Figure 3.3 shows that for the large matrices, 16 ranks with 32 cores each configuration (middle column) is where HPX performs similar to MPI. However, in the first column for the finished experiments we see that HPX is 24× and 7× slower than the MPI. Also on *twitter7*, HPX is, for the first time, more than 2× slower (the right column). Nonetheless, we observe some speedup when going from two nodes to four nodes on *webbase-2001* for both codes although it is not nearly ideal: considering the 64 rank configuration, MPI and HPX take

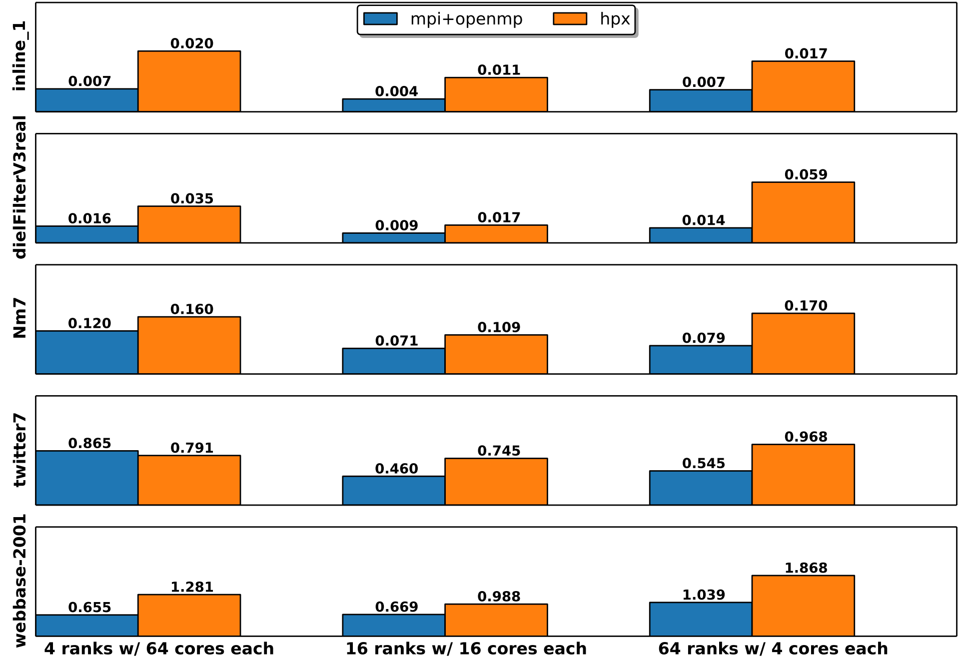


Figure 3.2 Execution times of the distributed Lanczos algorithm on two EPYC nodes.

1.04 and 1.87 sec on two nodes, respectively, and 0.80 and 1.44 sec on four nodes.

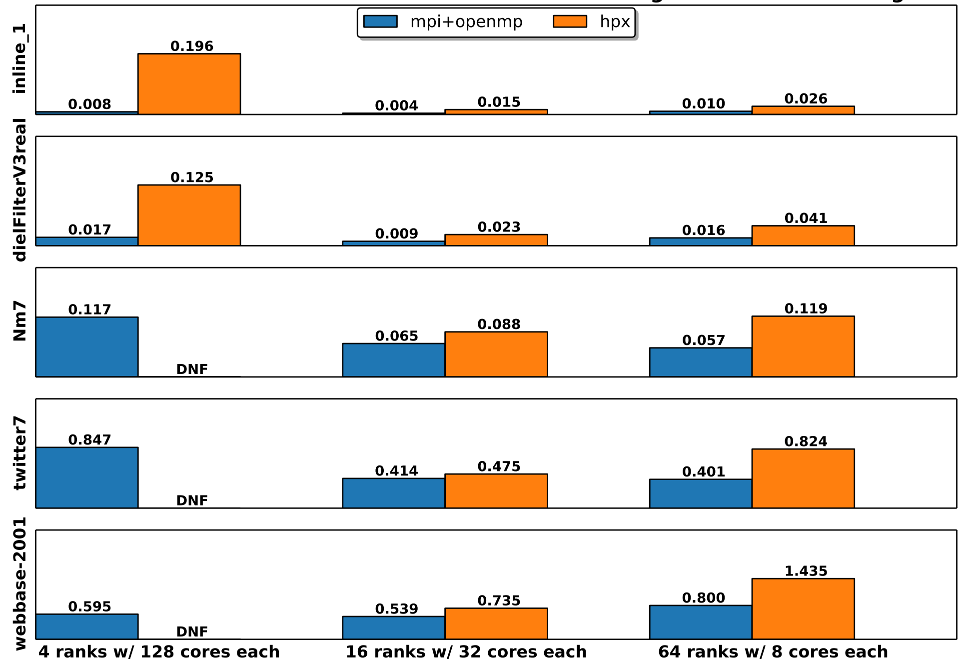


Figure 3.3 Execution times of the distributed Lanczos algorithm on four EPYC nodes.

3.3 Discussion

While HPX achieves excellent parallelism and performance on shared memory systems in comparison to OpenMP's both task-parallel and loop-parallel models, it fails to do so on distributed memory systems in comparison to the plain MPI+OpenMP model. As such, we performed some analysis to figure out why it is the case.

We limit our analysis to *webbase-2001*, the largest matrix, as it should provide plenty of work to keep the threads occupied in an ideal scenario. However, we must state that we observe the trends we see on this matrix on the others as well.

Table 3.2 shows the average execution time of a single Lanczos iteration for the shared and distributed implementation of HPX with varying block sizes. Both versions use an entire EPYC node.

Looking at the raw data and comparing the distributed HPX results to the shared HPX results, we observe that HPX's distributed implementation does a poor job, performing around 25× worse than the shared version (13.7 sec vs 0.57 sec) at the finest task granularity. What is more, the distributed version performs the best when there is not nearly enough degree of parallelism. Specifically when the block size is 2^{23}, the CSB block count is 15×15, which is 1D row partitioned among 16 localities. As such, the first 15 localities get 1×15 matrix blocks while the last one gets virtually nothing. Moreover, while the last locality gets no block of any column vectors, the rest gets one such block each. This means that there are 15 tasks spawned for the SpMV kernel and a single task spawned for the other kernels (such as the dot product or linear combination kernel). Despite the fact that there are 8 threads per locality waiting for a task, it is surprising to realize that this task granularity yields the top performance.

The results make much more sense for the shared version: there are 226×226 SpMV tasks and 226 tasks per other kernel spawned, which are then executed by 128 threads, with the block size of 2^{19}. And this task granularity performs better than the others. Such disparity between the shared and distributed versions makes us wonder whether this is simply an

implementation issue or it is a problem in the runtime system's core.

Matrix	Block Size	Block Count	Shared HPX	Distributed HPX
webbase	2^{19}	226×226	0.57	13.73
	2^{20}	113×113	0.77	6.82
	2^{21}	57×57	1.38	3.71
	2^{22}	29×29	0.74	3.55
	2^{23}	15×15	0.99	1.38
	2^{24}	8×8	1.29	1.47

Table 3.2 Execution times of shared and distributed HPX codes per Lanczos iteration on a single node (128 threads on shared and 16 localities with 8 threads each on distributed run).

We were also wondering if these problematic trends we observed with HPX could be attributed to the lack of new software packages and libraries available to us on HPCC at MSU. As such, we repeated some of the experiments at NERSC using Haswell nodes but as seen in Table 3.3, we observed the same outcome there too: first, HPX is considerably slower than the MPI (it is not given in the table but it takes 0.73 sec for MPI on 4 nodes of Haswell per Lanczos iteration whereas HPX cannot get faster than 1.69 sec). Second, HPX performs the best when there is not nearly enough number of tasks to utilize all the available localities and threads. HPX somehow prefers coarse-grained tasks on distributed runs, which is the opposite of what we see on shared memory runs.

Matrix	Block Size	Block Count	Haswell	EPYC
webbase	2^{19}	226×226	15.14	5.42
	2^{20}	113×113	7.70	2.94
	2^{21}	57×57	4.09	1.44
	2^{22}	29×29	2.15	0.96
	2^{23}	15×15	2.27	0.74
	2^{24}	8×8	1.69	0.89

Table 3.3 Execution times of distributed HPX codes per Lanczos iteration on four nodes of Haswell and EPYC (16 localities with 8 threads each on Haswell and 16 localities with 32 threads each on EPYC).

We also tried a couple of possible optimization techniques such as calling *dgemv* function of *sequential MKL* from the inner product and linear combination kernels, disabling work stealing and/or forcing *[]* operator, which is a member function in data-holding classes, to

be inlined by the compiler. Nonetheless, the original code ran faster without any of these possible optimization efforts.

In addition, we thought the lack of improvements we see on the distributed code might be due to the overhead of using an object oriented approach with the client and server classes. In the shared memory counterpart of Lanczos, we had simply used flat arrays to hold vector and matrix data while relying on future to void variables to determine the availability of vector chunks. As such, we wondered if we could improve the performance for the distributed runs by simply adopting the same approach, except for the global communication part of course. This code can be seen here. However, this version without any object creation or destruction overheads worked slower than the first version with the client-server approach.

Finally, we wanted to estimate the time the global communications take for both HPX and MPI+OpenMP. To do that, we made tasks return immediately without any computation for all kernels. This way, the entire time would be spent on either global communications or task creation/execution/deletion overheads. The execution times of normal runs and those with **empty** tasks can be seen in Table 3.4. We know from the shared memory runs that HPX does not yield any significant runtime system overheads. Therefore, we believe that most time for the empty runs is spent on the communication part. Although both MPI+OpenMP and HPX communicates the same data, HPX spends much more time on it: MPI+OpenMP takes 0.65 sec for communication whereas it varies between 0.73 and 10.67 sec for HPX. It is important to note there that 0.73 sec occurs when only half of the localities participate in communication as there are only 8 row blocks where only the localities with an even locality ID do all the communication and computation.

To confirm our suspicions regarding HPX's network communication issues, we measured the bandwidth and latency achieved within HPX and compared it to pure MPI numbers. Note that HPX itself enables the use of MPI as well as TCP for the networking operations in the HPX runtime but since TCP was considerably slower than MPI, we opted for MPI within HPX as the communication layer. Therefore, whenever a communication call is used

Matrix	BS	BC	MPI	MPI empty	HPX	HPX empty
webbase	2^{19}	226×226	0.97	0.65	13.31	10.67
	2^{20}	113×113	0.98	0.65	6.62	6.81
	2^{21}	57×57	0.97	0.65	3.50	2.77
	2^{22}	29×29	0.98	0.66	2.14	1.41
	2^{23}	15×15	0.98	0.65	1.58	1.03
	2^{24}	8×8	0.98	0.64	1.33	0.73

Table 3.4 Execution times of MPI+OpenMP and HPX codes with normal and *empty* tasks per Lanczos iteration on a single node of EPYC (16 localities with 8 threads each on EPYC) highlighting the global communication overhead.

with HPX, it is simply a wrapped MPI call inside HPX's runtime system.

To assess the network bandwidth, we simply create a vector of double of varying sizes between 8KB and 512MB and synchronously send it from the $rank/locality0$ to $rank/locality1$. For the latency, we only send a single integer in the same way. We report the average over 100 tries. The benchmark numbers are reported in Table 3.5 and 3.6.

Number	pure MPI	HPX using MPI
min Bandwidth	202.4	39.2
max Bandwidth	18857.6	2170.5
avg Bandwidth	13177.1	1148.6
min Latency	0.08	81.80
max Latency	0.12	122.29
avg Latency	0.09	91.63

Table 3.5 Network Benchmark on AMD EPYC for Intra-Node Communication. The bandwidth reported is in MB/s and the latency is in µ (microsecond).

Number	pure MPI	HPX using MPI
min Bandwidth	241.5	10.87
max Bandwidth	12075.1	704.8
avg Bandwidth	10468.2	332.8
min Latency	0.30	254.71
max Latency	1.33	406.91
avg Latency	0.37	312.35

Table 3.6 Network Benchmark on AMD EPYC for Inter-Node Communication. The bandwidth reported is in MB/s and the latency is in µ (microsecond).

3.4 Conclusion of This Work

It seems that the distributed HPX yields poor and questionable numbers regarding the 1-node, 2-node and 4-node experiments. That is, it is not possible to achieve a speedup over the plain MPI+OpenMP implementation regardless of the blocking factor. Also, it is interesting to see HPX performing best when the tasks are too coarse grained. Seeing how slow HPX is when run with empty tasks suggests that the communication becomes the bottleneck as the tasks are finer grained. Moreover, the bandwidth and latency numbers observed suggest that HPX introduces a significant overhead when a communication call is used despite using MPI as the networking layer. After seeing the performance of HPX in distributed case for Lanczos solver, we decided not to pursue our initial intentions, which was to develop our own large-scale sparse solver and graph analytics framework using HPX as the backbone.

CHAPTER 4

HYBRID EIGENSOLVERS FOR NUCLEAR CONFIGURATION INTERACTION CALCULATIONS

This chapter has been published in Computer Physics Communications on November 2023 [5], available at: https://doi.org/10.1016/j.cpc.2023.108888.

So far in Chapter 2 and 3, we were targeting a broad base of matrices with varying sizes, sparsity patterns, and domains from the SuiteSparse Matrix Collection. The goal was to explore the acceleration of sparse matrix computations through asynchronous dataflow models on shared and distributed memory architectures. This kind of general exploration allowed us to experiment with different runtime systems and different matrices. Although we showed these runtime systems' merit on the Lanczos and LOBPCG algorithms, we only approached these sparse eigensolvers as a collection of linear algebra kernels where we ran each solver for a fixed number of iterations using those matrices.

In this work, we take an interest in the convergence behavior of these algorithms in the context of nuclear physics where the eigenvalue problem has certain significance. We namely explore whether we could improve the convergence of the iterative eigensolvers through hybrid algorithms. While doing that, we also attempt to tackle the limitations of the eigensolvers with respect to the large-scale sparse systems that arise in nuclear physics. Therefore, while observing the iteration counts and total number of SpMVs and/or SpMMs performed as we employ both pure and hybrid algorithms, we confine ourselves to the test problems from the study of nuclear physics. Moreover, although we use MATLAB and Fortran for the experiments of this work, iteration and SpMV counts of the algorithms are agnostic to programming languages and runtime systems.

The computational study of the structure of atomic nuclei involves solving the many-body Schrödinger equation for a nucleus consisting of Z protons and N neutrons, with $A = Z + N$ the total number of nucleons,

$$\hat{H}\ \Psi_i(\vec{r}_1, \ldots, \vec{r}_A) \quad = \quad E_i\, \Psi_i(\vec{r}_1, \ldots, \vec{r}_A)\,, \tag{4.1}$$

where $\hat{H}$ is the nuclear Hamiltonian, E_i are the discrete energy levels of the low-lying spectrum of the nucleus, and Ψ_i the corresponding A-body wavefunctions. A commonly used approach to address this problem is the no-core Configuration Interaction (CI) method (or No-Core Shell Model), in which the many-body Schrödinger equation, Eq. (4.1), becomes an eigenvalue problem

$$H\,x \;=\; \lambda\,x\,, \tag{4.2}$$

where H is an $n \times n$ square matrix that approximates the many-body Hamiltonian $\hat{H}$, λ is an eigenvalue of H, and x is the corresponding eigenvector [8, 53, 58]. The size n of the symmetric matrix H grows rapidly with the number of nucleons A and with the desired numerical accuracy, and can easily be several billion or more; however, this matrix is extremely sparse, at least for nuclei with $A \geq 6$. Furthermore, we are typically interested in only a few (5–10) eigenvalues at the low end of the spectrum of H. An iterative method that can make use of an efficient Hamiltonian-vector multiplication procedure is therefore often the preferred method to solve Eq. (4.2) for the lowest eigenpairs.

For a long time, the Lanczos algorithm [36] with full orthogonalization was the default algorithm to use because it is easy to implement and because it is quite robust even though it requires storing hundreds of Lanczos basis vectors. Indeed, there are several software packages in which the Lanczos algorithm is implemented for nuclear structure calculations. Here we focus on the software MFDn (Many-Fermion Dynamics for nuclear structure), which is a hybrid MPI/OpenMPI code that is being used at several High-Performance Computing centers; it has recently also been ported to GPUs using OpenACC.

In recent work [51], we have shown that the low-lying eigenvalues can be computed efficiently by using the Locally Optimal Block Preconditioned Conjugate Gradient (LOBPCG) algorithm [32]. The advantages of the LOBPCG algorithm, which we will describe with some detail in the next section, over the Lanczos algorithm include

- The algorithm is a block method that allows us to multiply H with several vectors

simultaneously. That is, instead of an SpMV, one performs an Sparse Matrix-Matrix multiplication (SpMM) of a sparse square $n \times n$ matrix on a tall skinny $n \times n_b$ matrix at every iteration, which introduces an additional level of concurrency in the computation and enables us to exploit data locality better. In order to converge 5 to 10 eigenpairs we typically use blocks of 8 to 16 vectors – this can also be tuned to the hardware of the HPC platform.

- The algorithm allows us to make effective use of pre-existing approximations to several eigenvectors.

- The algorithm allows us to take advantage of a preconditioner that can be used to accelerate convergence.

- Other dense linear algebra operations can be implemented as level 3 BLAS.

Even though Lanczos is efficient in terms of the number of sparse matrix vector multiplications (SpMV) it uses, we have shown that the LOBPCG method often takes less wallclock time to run because performing a single SpMM is more efficient than performing several SpMVs sequentially, which is required in the Lanczos algorithm.

However, there are occasionally some issues with LOBPCG:

- The method can become unstable near convergence. Although methods for stabilizing the algorithm has been developed and implemented [26, 19], they do not completely eliminate the problem.

- Even though the algorithm in principle only requires storing three blocks of vectors, in practice, many more blocks of vectors are needed to avoid performing additional SpMMs in the Rayleigh-Ritz procedure. This is a problem for machines on which high bandwidth memory is in short supply (such as GPUs).

In this work, we examine several alternative algorithms for solving large-scale eigenvalue problems in the context of nuclear configuration interaction calculations. In particular, we

will examine the block Lanczos algorithm [21] and the Chebyshev filtered subspace iteration. Both are block algorithms that can benefit from an efficient implementation of the SpMM operation and can take advantage of good initial guesses to several eigenvectors, if they are available. Neither one of these algorithms can incorporate a preconditioner, which is a main drawback. However, as we will show in Section 4.3, in the early iterations of these algorithms, good approximations to the desired eigenpairs emerge quickly, even though the total number of SpMVs required to obtain accurate approximations can be higher compared to the Lanczos and LOBPCG algorithms. This observation suggests that these algorithms can be combined with algorithms that are effective in refining existing eigenvector approximations. One such refinement algorithm is the residual minimization method (RMM) with direct inversion of iterative subspace (DIIS) correction. This algorithm has an additional feature that it can reach convergence to a specific eigenpair without performing orthogonalization against approximations to other eigenpairs as long as a sufficiently accurate initial guess is available. Therefore, this algorithm can also be used to compute (or refine) different eigenpairs independently. This feature introduces an additional level of concurrency in the eigenvalue computation that enhances the parallel scalability.

The chapter is organized as follows. In the next section, we give an overview of the Lanczos, block Lanczos, LOBPCG as well as the Chebyshev filtered subspace iteration (ChebFSI) algorithms. We also describe the RMM-DIIS algorithm and discuss how it can be combined with (block) Lanczos, LOBPCG and ChebFSI to form a hybrid algorithm to efficiently compute the desired eigenpairs. In Section 4.3, we give several numerical examples to demonstrate the effectiveness of the hybrid algorithm and compare it with the standard algorithms. We discuss both the convergence of the algorithm and some implementation issues.

4.1 Existing Algorithms

We review several algorithms for computing a few algebraically smallest eigenvalues and the corresponding eigenvectors. We denote the eigenvalues of the $n \times n$ nuclear CI Hamiltonian H arranged in an increasing order by $\lambda_1 \leq \lambda_2 \leq \cdots \leq \lambda_n$. Their corresponding

eigenvectors are denoted by x_1, x_2, ..., x_n. We are interested in the first $n_{ev} \ll n$ eigenvalues and eigenvectors. If we define $X = [x_1, x_2, \ldots, x_{n_{ev}}]$ and $\Lambda = \text{diag}\{\lambda_1, \lambda_2, \ldots, \lambda_{n_{ev}}\}$, respectively, we have $HX = X\Lambda$.

4.1.1 Lanczos Algorithm

The Lanczos algorithm is a classical algorithm for solving large scale eigenvalue problems. The algorithm generates an orthonormal basis of a k-dimensional Krylov subspace

$$\mathcal{K}(H; v_1) = \{v_1, Hv_1, ..., H^{k-1}v_1\}, \tag{4.3}$$

where v_1 is an appropriately chosen and normalized starting guess. Such a basis is produced by a Gram-Schmidt process in which the key step of obtaining the $j+1$st basis vector is

$$w_j = (I - V_j V_j^T)Hv_j, \quad v_{j+1} = w_j / \|w_j\|, \tag{4.4}$$

where V_j is a matrix that contains all previous orthonormal basis vectors, i.e.,

$$V_j = (v_1, v_2, ..., v_j) \,.$$

The projection of H into the k-dimensional subspace spanned by columns of V_k is a tridiagonal matrix T_k that satisfies

$$HV_k = V_k T_k + w_k e_k^T, \tag{4.5}$$

where e_k is the last column of a $k \times k$ identity matrix. Approximate eigenpairs of H are obtained by solving the $k \times k$ eigenvalue problem

$$T_k q = \theta q. \tag{4.6}$$

It follows from (4.5), (4.6) and the fact that $V_k^T V_k = I_k$, $V_k^T w_k = 0$ that the relative residual norm associated with an approximate eigenpair $(\theta, V_k q)$ can be estimated by

$$\frac{\|H(V_k q) - \theta(V_k q)\|}{|\theta|} = \frac{\|w_k\| \cdot |e_k^T q|}{|\theta|}. \tag{4.7}$$

4.1.2 Block Lanczos

One of the drawbacks of the standard Lanczos algorithm is that it is not easy for the algorithm to take advantage of good initial guesses to several desired eigenvectors. Although we can take a simple linear combination or average of these initial guesses as the initial vector v_1, the Lanczos algorithm tends to converge to one of the eigenvectors much faster than others.

An algorithm that can take advantage of multiple starting guesses to different eigenvectors is the block Lanczos algorithm. The Block Lanczos algorithm generates an orthonormal basis of a block Krylov subspace

$$\mathcal{K}(H; V_1) = \{V_1, HV_1, ..., H^{k-1}V_1\}, \tag{4.8}$$

where V_1 is a matrix that contains n_b orthonormal basis vectors that are often good initial guesses to the desired eigenvectors. Note that, in practice, n_b can be chosen to be slightly larger than the number of desired eigenvalues n_{ev}.

The Gram-Schmidt process used to generate an orthonormal basis in Lanczos is replaced by a block Gram-Schmidt step that is characterized by

$$W_j = (I - \mathbf{V}_j \mathbf{V}_j^T) HV_j, \tag{4.9}$$

where the matrix $\mathbf{V}_j$ contains j block orthonormal basis, i.e.,

$$\mathbf{V}_j = (V_1, V_2, ..., V_j). \tag{4.10}$$

The normalization step in (4.4) is simply replaced by a QR factorization step, i.e.

$$W_j = V_{j+1} R_{j+1},$$

where $V_{j+1}^T V_{j+1} = I_{n_b}$, and R_{j+1} is an $n_b \times n_b$ upper triangular matrix.

The projection of H into the subspace spanned by columns of $\mathbf{V}_k$ is a block tridiagonal matrix $\mathbf{T}_k$ that satisfies

$$H\mathbf{V}_k = \mathbf{V}_k \mathbf{T}_k + W_k E_k^T, \tag{4.11}$$

where E_k is the last n_b columns of an $n_b \cdot k \times n_b \cdot k$ identity matrix.

Approximate eigenpairs of H are obtained by solving the $n_b \cdot k \times n_b \cdot k$ eigenvalue problem

$$\mathbf{T}_k q = \theta q. \tag{4.12}$$

It follows from (4.11), (4.12) and the fact that $\mathbf{V}_k^T \mathbf{V}_k = I_{n_b k}$, $\mathbf{V}_k^T W_k = \mathbf{0}$ that the relative residual norm associated with an approximate eigenpair $(\theta, V_k q)$ can be estimated by

$$\frac{\|H(V_k q) - \theta(V_k q)\|}{|\theta|} = \frac{\|W_k\|_F \cdot \|E_k^T q\|}{|\theta|}, \tag{4.13}$$

Algorithm 4.1 outlines the main steps of the block Lanczos algorithm. Both the Lanczos and block Lanczos algorithms produce approximation to the desired eigenvector in the form of

$$z = p_d(H) v_0,$$

where v_0 is some starting vector and d is the degree of the polynomial.

However, for the same number of multiplications of the sparse matrix H with a vector (SpMVs), denoted by m, the degree of the polynomial generated in a block Lanczos algorithm is $d = m/n_b$, whereas, in the standard Lanczos algorithm, the degree of the polynomial is $d = m$. Because the accuracy of the approximate eigenpairs obtained from the Lanczos and block Lanczos methods is directly related to d, we expect more SpMVs to be used in a block Lanczos algorithm to reach convergence. On the other hand, in a block Lanczos method, one can perform n_b SpMVs as a matrix-matrix multiplication (SpMM) on a tall skinny matrix consisting of a block of n_b vectors, which is generally more efficient than performing n_b SpMVs in succession. As a result, the block Lanczos method can take less time even if it performs more SpMVs.

4.1.3 LOBPCG

It is well known that the invariant subspace associated with the smallest n_{ev} eigenvalues and spanned by columns of $X \in \mathbb{R}^{n \times n_{ev}}$ is the solution to the trace minimization problem

$$\min_{X^T X = I} \text{trace}(X^T H X). \tag{4.14}$$

Algorithm 4.1 The block Lanczos algorithm.

Input: The sparse matrix H
Input: The number of desired eigenvalues n_{ev}
Input: An initial guess to the eigenvectors associated with the lowest $n_b \geq n_{ev}$ eigenvalues $X^{(0)} \in \mathbb{R}^{n\times n_b}$
Input: Convergence tolerance (tol)
Input: Maximum number of iteration allowed ($maxiter$)
Output: (Λ, X), where Λ is a $n_{ev} \times n_{ev}$ diagonal matrix containing the desired eigenvalues, and $X \in \mathbb{R}^{n\times n_{ev}}$ contains the corresponding eigenvector approximations

1: Generate $V_1 \in \mathbb{R}^{n\times n_b}$ that contains an orthonormal basis of $X^{(0)}$
2: $\mathbf{V}_1 \leftarrow (V_1)$
3: $\mathbf{T}_0 \leftarrow \mathbf{V}_0^T H \mathbf{V}_0$
4: **for** $i = 1, 2, \ldots,$maxiter **do**
5: $\quad W_i \leftarrow (I - \mathbf{V}_i \mathbf{V}_i^T) H V_i$
6: $\quad$ Generate V_{i+1} that contains an orthonormal basis of W_i
7: $\quad \mathbf{V}_{i+1} \leftarrow (\mathbf{V}_i \; V_i)$
8: $\quad$ Update $\mathbf{T}_{i+1} \leftarrow \mathbf{V}_{i+1}^T H \mathbf{V}_{i+1}$
9: $\quad$ Solve the projected eigenvalue problem $\mathbf{T}_{i+1} U = U\Theta$, where $U^T U = I$, Θ is a diagonal matrix containing eigenvalues of T_{i+1} in an ascending order
10: $\quad X_i = \mathbf{V}_{i+1} U(:, 1 : n_{ev})$
11: $\quad$ Determine number of converged eigenpairs n_c by checking the Ritz residual estimate (4.13)
12: $\quad$ **if** $n_c \geq n_{ev}$ **then**
13: $\quad\quad$ exit loop
14: $\quad$ **end if**
15: **end for**
16: $\Lambda \leftarrow \Theta$
17: $X \leftarrow X_i$

The LOBPCG algorithm developed by Knyazev [32] seeks to solve (4.14) by using the updating formula

$$X^{(i+1)} = X^{(i)} C_1^{(i+1)} + W^{(i)} C_2^{(i+1)} + P^{(i-1)} C_3^{(i+1)}, \tag{4.15}$$

to approximate the eigenvector corresponding to the n_{ev} leftmost eigenvalues of H, where $W^{(i)} \in \mathbb{R}^{n\times n_{ev}}$ is the preconditioned gradient of the Lagrangian

$$\mathcal{L}(X, \Lambda) = \frac{1}{2}\,\mathrm{trace}(X^T H X) - \frac{1}{2}\,\mathrm{trace}\left[(X^T X - I)\Lambda\right] \tag{4.16}$$

associated with (4.14) at $X^{(i)}$, and $P^{(i-1)}$ is the search direction obtained in the $(i-1)$st iterate of the optimization procedure, and $C_1^{(i+1)}$, $C_2^{(i+1)}$, $C_3^{(i+1)}$ are a set of coefficient matrices of matching dimensions that are obtained by minimizing (4.16) within the subspace

$S^{(i)}$ spanned by

$$S^{(i)} \equiv \left(X^{(i)} \quad W^{(i)} \quad P^{(i-1)} \right). \tag{4.17}$$

To improve the convergence of the LOBPCG algorithm, one can include a few more vectors in $X^{(i)}$ so that the number of columns in $X^{(i)}$, $W^{(i)}$ and $P^{(i)}$ is $n_b \geq n_{ev}$.

The preconditioned gradient $W^{(i)}$ can be computed as

$$W^{(i)} = K^{-1}(HX^{(i)} - X^{(i)}\Theta^{(i)}) \tag{4.18}$$

where $\Theta^{(i)} = X^{(i)T}HX^{(i)}$, and K is a preconditioner that approximates H in some way. The subspace minimization problem that yields the coefficient matrix $C_1^{(i+1)}$, $C_2^{(i+1)}$, $C_3^{(i+1)}$, which are three block rows of a $3n_b \times n_b$ matrix $C^{(i+1)}$, can be solved as a generalized eigenvalue problem

$$\left(S^{(i)T}HS^{(i)} \right) C^{(i+1)} = \left(S^{(i)T}S^{(i)} \right) C^{(i+1)} D^{(i+1)}, \tag{4.19}$$

where $D^{(i+1)}$ is a $n_b \times n_b$ diagonal matrix containing n_b leftmost eigenvalues of the projected matrix pencil
$\left(S^{(i)T}HS^{(i)}, S^{(i)T}S^{(i)} \right)$. The procedure that forms the projected matrices $S^{(i)T}HS^{(i)}$ and $S^{(i)T}S^{(i)}$ and solves the projected eigenvalue problem (4.19) is often referred to as the *Rayleigh–Ritz* procedure [44]. Note that the summation of the last two terms in (4.15) represents the search direction followed in the ith iteration, i.e.,

$$P^{(i)} = W^{(i)}C_2^{(i+1)} + P^{(i-1)}C_3^{(i+1)}. \tag{4.20}$$

Algorithm 4.2 outlines the main steps of the basic LOBPCG algorithm. The most computationally costly step of Algorithm 4.2 is the multiplication of H with a set of vectors. Although it may appear that we need to perform such calculations in steps 8 (where the projected matrix $S^{(i)T}HS^{(i)}$ is formed) and 10, the multiplication of H with $X^{(i)}$, $X^{(i+1)}$ and $P^{(i)}$ can be avoided because $HX^{(i+1)}$ and $HP^{(i)}$ satisfy the following recurrence relationships

$$HX^{(i+1)} = HX^{(i)}C_1^{(i+1)} + HW^{(i)}C_2^{(i+1)} + HP^{(i-1)}C_3^{(i+1)}, \tag{4.21}$$

$$HP^{(i)} = HW^{(i)}C_2^{(i+1)} + HP^{(i-1)}C_3^{(i+1)}. \tag{4.22}$$

Algorithm 4.2 The basic LOBPCG algorithm.

Input: The sparse matrix H
Input: A preconditioner K
Input: An initial guess to the eigenvectors associated with the lowest $n_b \geq n_{ev}$ eigenvalues $X^{(0)} \in \mathbb{R}^{n \times n_b}$
Input: Number of desired eigenvalues (n_{ev})
Input: Convergence tolerance (tol) and maximum number of iteration allowed ($maxiter$)
Output: (Λ, X), where Λ is a $n_{ev} \times n_{ev}$ diagonal matrix containing the desired eigenvalues, and $X \in \mathbb{R}^{n \times n_{ev}}$ contains the corresponding eigenvector approximations

1: $[C^{(1)}, \Theta^{(1)}] \leftarrow \texttt{RayleighRitz}(H, X^{(0)})$
2: $X^{(1)} \leftarrow X^{(0)} C^{(1)}$
3: $R^{(1)} \leftarrow HX^{(1)} - X^{(1)}\Theta^{(1)}$
4: $P^{(0)} \leftarrow \emptyset$
5: **for** $i = 1, 2, \ldots,$maxiter **do**
6: $\quad W^{(i)} \leftarrow K^{-1} R^{(i)}$
7: $\quad S^{(i)} \leftarrow \left[X^{(i)}, W^{(i)}, P^{(i-1)}\right]$
8: $\quad [C^{(i+1)}, \Theta^{(i+1)}] \leftarrow \texttt{RayleighRitz}(H, S^{(i)})$
9: $\quad X^{(i+1)} \leftarrow S^{(i)} C^{(i+1)}$
10: $\quad R^{(i+1)} \leftarrow HX^{(i+1)} - X^{(i+1)}\Theta^{(i+1)}$
11: $\quad P^{(i)} \leftarrow W^{(i)} C_2^{(i+1)} + P^{(i-1)} C_3^{(i+1)}$
12: Determine number of converged eigenpairs n_c by comparing the relative norms of the leading n_{ev} columns of $R^{(i+1)}$ against the convergence tolerance tol
13: **if** $n_c \geq n_{ev}$ **then**
14: exit loop
15: **end if**
16: **end for**
17: $\Lambda \leftarrow \Theta^{(i)}(1 : n_{ev}, 1 : n_{ev})$
18: $X \leftarrow X^{(i)}(:, 1 : n_{ev})$

Therefore, the only SpMM we need to perform is $HW^{(i)}$. For the nuclear CI calculations of interest, the dimension n of the sparse symmetric matrix H can be several billions, whereas $W^{(i)}$ is a tall skinny $n \times n_b$ matrix with n_b typically of the order of 8 to 16.

4.1.4 Chebyshev filtering

An mth-degree Chebyshev polynomial of the first kind can be defined recursively as

$$T_m(t) = 2tT_{m-1}(t) - T_{m-2}(t), \tag{4.23}$$

with $T_0(t) = 1$ and $T_1(t) = t$. The magnitude of $T_m(t)$ is bounded by 1 within $[-1, 1]$ and grows rapidly outside of this interval.

By mapping the unwanted eigenvalues (e.g., the unoccupied states) of the many-body Hamiltonian H enclosed by $[\lambda_{\mathrm{F}}, \lambda_{\mathrm{ub}}]$ to $[-1, 1]$ through the linear transformation $(t-c)/e$, where $c = (\lambda_{\mathrm{F}}+\lambda_{\mathrm{ub}})/2$ and $e = (\lambda_{\mathrm{ub}}-\lambda_{\mathrm{F}})/2$, we can use $\hat{T}_m(H) = T_m((H-cI)/e)v$ to amplify the eigenvector components in v that correspond to eigenvalues outside of $[\lambda_{\mathrm{F}}, \lambda_{\mathrm{ub}}]$. Figure 4.1 shows a 10th degree Chebyshev polynomial defined on the spectrum of a Hamiltonian matrix and how the leftmost eigenvalues λ_i, $i = 1, 2, ..., 8$ are mapped to $T_{10}(\lambda_i)$.

Applying $T_m((H-cI)/e)$ repeatedly to a block of vectors V filters out the eigenvectors associated with eigenvalues in $[\lambda_{\mathrm{F}}, \lambda_{\mathrm{ub}}]$. The desired eigenpairs can be obtained through the standard Rayleigh–Ritz procedure [44].

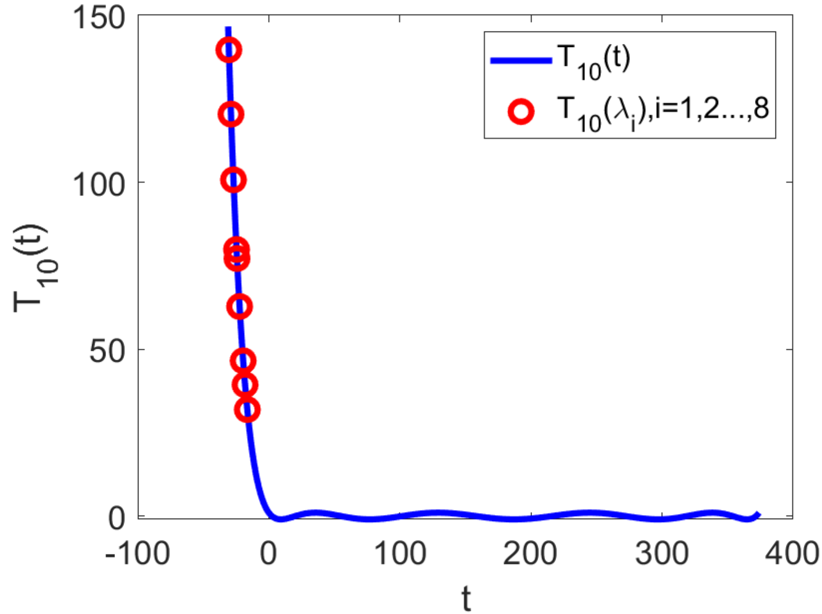


Figure 4.1 Chebyshev polynomials of the first kind.

To obtain an accurate approximation to the desired eigenpairs, a high degree Chebyshev polynomial may be needed. Instead of applying a high degree polynomial once to a block of vectors, which can be numerically unstable, we apply Chebyshev polynomial filtering within a subspace iteration to iteratively improve approximations to the desired eigenpairs. We will refer to this algorithm as a Chebyshev filtering based subspace iteration (CheFSI). The basic steps of this algorithm is listed in Algorithm 4.3.

Algorithm 4.3 The Chebyshev filtering based subspace iteration (CheFSI).

Input: The sparse matrix H
Input: An initial guess to the eigenvectors associated with the lowest $n_b \geq n_{ev}$ eigenvalues $X^{(0)} \in \mathbb{R}^{n \times n_b}$
Input: Number of desired eigenvalues (n_{ev})
Input: The degree of the Chebyshev polynomial d
Input: Convergence tolerance (tol)
Input: Maximum number of subspace iteration allowed ($maxiter$)
Output: (Λ, X), where Λ is a $n_{ev} \times n_{ev}$ diagonal matrix containing the desired eigenvalues, and $X \in \mathbb{R}^{n \times n_{ev}}$ contains the corresponding eigenvector approximations

1: $e \leftarrow (\lambda_{\text{ub}} - \lambda_{\text{F}})/2$
2: $c \leftarrow (\lambda_{\text{ub}} + \lambda_{\text{F}})/2$
3: $[Q, R] \leftarrow \texttt{CholeskyQR}(H, X^{(0)})$
4: **for** $i = 1, 2, \ldots,$maxiter **do**
5: $\quad W \leftarrow 2(HQ - cQ)/e$
6: $\quad$ **for** $j = 2, \ldots, d$ **do**
7: $\quad\quad Y \leftarrow 2(HW - cW)/e - Q$
8: $\quad\quad Q \leftarrow W$
9: $\quad\quad W \leftarrow Y$
10: $\quad$ **end for**
11: $\quad [Q, R] \leftarrow \texttt{CholeskyQR}(Y)$
12: $\quad T \leftarrow Q^T HQ$
13: $\quad$ Solve the eigenvalue problem $TS = S\Theta$, where Θ is diagonal, and update Q by $Q \leftarrow QS$
14: $\quad$ Determine number of converged eigenpairs n_c by comparing the relative norms of the leading n_{ev} columns of $R = HQ - QD$ against the convergence tolerance tol
15: $\quad$ **if** $n_c \geq n_{ev}$ **then**
16: $\quad\quad$ exit loop
17: $\quad$ **end if**
18: **end for**
19: $\Lambda \leftarrow \Theta(1 : n_{ev}, 1 : n_{ev})$
20: $X \leftarrow Q(:, 1 : n_{ev})$

Owing to the three-term recurrence in (4.23), $W = \hat{T}_m(H)V$ can be computed recursively without forming $\hat{T}_m(H)$ explicitly in advance. Lines 5 to 9 of Algorithm 4.3 illustrates how this step is carried out in detail. To maintain numerical stability, we orthonormalize vectors in W. The orthonormalization can be performed by a (modified) Gram–Schmidt process or by a Householder transformation based QR factorization [22].

In Algorithm 4.3, the required inputs are a filter degree, d, an estimated upper bound of the spectrum of H, λ_{ub}, and an estimated spectrum cutoff level, λ_{F}. The estimation of the

upper bound λ_{ub} can be calculated by running a few Lanczos iterations [18, 65, 39], and λ_{F} can often be set at 0 or the estimation of the n_b+1st leftmost eigenvalue of H obtained from the Lanczos algorithm. In the subsequent subspace iterations, λ_{F} can be modified based on more accurate approximations to the desired eigenvalues. See the work of Saad [49] and Zhou *et al.* [66] for more details on Chebyshev filtering.

4.1.5 RMM-DIIS

The residual minimization method (RMM) [64, 28] accelerated by direct inversion of iterative subspace (DIIS) [45] was developed in the electronic structure calculation community to solve a linearized Kohn-Sham eigenvalue problem in each self-consistency field (SCF) iteration. Given a set of initial guesses to the desired eigenvectors, $\{x_j^0\}$, $j = 1, 2, ..., n_{ev}$, the method produces successively more accurate approximations by seeking an optimal linear combination of previous approximations to the jth eigenvector by minimizing the norm of the corresponding sum of residuals. To be specific, let $x_j^{(i)}$, $i = 0, 1, ..., \ell - 1$ be approximations to the jth eigenvector of H obtained in the previous $\ell - 1$ steps of the RMM-DIIS algorithm, and $\theta_j^{(i)}$ be the corresponding eigenvalue approximations. In the ℓth iteration (for $\ell > 1$), we first seek an approximation in the form of

$$\tilde{x}_j = \sum_{i=\min\{0,\ell-s\}}^{\ell-1} \alpha_i x_j^{(i)}, \tag{4.24}$$

where

$$\sum_{i=\ell-s}^{\ell-1} \alpha_i = 1, \tag{4.25}$$

for some fixed $1 \leq s \leq s_{\max}$. The coefficients α_i's are obtained by solving the following constrained least squares problem

$$\min \| \sum_{i=\min\{\ell-s\}}^{\ell-1} \alpha_i r_j^{(i)} \|^2, \tag{4.26}$$

where $r_j^{(i)} = Hx_j^{(i)} - \theta_j^{(i)} x_j^{(i)}$ is the residual associated with the approximate eigenpair $(\theta_j^{(i)}, x_j^{(i)})$, subject to the same constraint defined by (4.25). The constrained minimiza-

tion problem can be turned into an unconstrained minimization problem by substituting $\alpha_{\ell-1} = 1 - \sum_{i=\min\{\ell-s\}}^{\ell-2} \alpha_i$ into (4.26).

Once we solve (4.26), we compute the corresponding residual

$$\tilde{r}_j = H\tilde{x}_j - \tilde{\theta}_j \tilde{x}_j,$$

where $\tilde{\theta}_j = \langle \tilde{x}_j, H\tilde{x}_j \rangle / \langle \tilde{x}_j, \tilde{x}_j \rangle$. A new approximation to the desired eigenvector is obtained by projecting H into the two-dimensional subspace W_j spanned by $\tilde{x}_j$ and $\tilde{r}_j$, and solving the 2×2 generalized eigenvalue problem

$$(W_j^T H W_j)g = \theta(W_j^T W_j)g. \tag{4.27}$$

Such an approximation can be written as

$$x_j^{(\ell)} = W_j g, \tag{4.28}$$

where g is the eigenvector associated with the smaller eigenvalue of the matrix pencil $(W_j^T H W_j, W_j^T W_j)$.

Although Rayleigh-Ritz procedure defined by (4.27) and (4.28) are often used to compute the lowest eigenvalue of H, the additional constraint specified by (4.24) and (4.25) keeps $x_j^{(\ell)}$ close to the initial guess of the jth eigenvector. Therefore, if the initial guess is sufficiently close to the jth eigenvector, $x_j^{(\ell)}$ can converge to this eigenvector instead of the eigenvector associated with the smallest eigenvalue of H.

Algorithm 4.4 outlines the main steps of the RMM-DIIS algorithm.

4.1.6 Comparison summary

The computational cost of all iterative methods discussed above is dominated by the the number of sparse Hamiltonian matrix vector multiplications (SpMVs).

In the Lanczos algorithm, the number of SpMVs is the same as the number of iterations. In block algorithms such as the block Lanczos algorithm and the LOBPCG algorithm, the number of SpMVs is the product of the block size and the number of iterations. The number of SpMVs used in the Chebyshev filtering subspace iteration is the product of the number of

Algorithm 4.4 The RMM-DIIS algorithm.

Input: The sparse matrix H
Input: An initial guess to the n_{ev} desired eigenvectors $\{x_j^{(0)}\}$, $j = 1, 2, ...n_{ev}$
Input: Maximum dimension of DIIS subspace s
Input: Convergence tolerance (tol)
Input: Maximum number of iteration allowed ($maxiter$)
Output: $\{(\theta_j, x_j)\}$, $j = 1, 2, ..., n_{ev}$, where θ_j is the approximation to the jth lowest eigenvalue, and x_j is the corresponding approximate eigenvector

1: **for** j=1,2,. . . ,n_{ev} **do**
2: $x_j^{(0)} \leftarrow x_j^{(0)}/\|x_j^{(0)}\|$
3: $\theta^{(0)} \leftarrow \langle x_j^{(0)}, Hx_j^{(0)}\rangle$
4: $r_j^{(0)} \leftarrow Hx_j^{(0)} - \theta_j^{(0)}x_j^{(0)}$
5: $\tilde{x}_j^{(1)} \leftarrow x_j^{(0)}$
6: $\tilde{r}_j^{(1)} \leftarrow r_j^{(0)}$
7: **for** $i = 1, 2, \ldots$,maxiter **do**
8: **if** $i > 1$ **then**
9: Solve the residual minimization least squares problem (4.26)
10: Set $\tilde{x}_j^{(i)}$ according to (4.24)
11: $\tilde{x}_j^{(i)} \leftarrow \tilde{x}_j^{(i)}/\|\tilde{x}_j^{(i)}\|$
12: Set the residual $\tilde{r}_j^{(i)} \leftarrow \sum_{\ell=\min\{0,i-s\}}^{i-1} \alpha_i x_j^{(\ell)}$
13: **end if**
14: Set $W_j \leftarrow (\tilde{x}_j^{(i)}, \tilde{r}_j^{(i)})$
15: Solve the Rayleigh-Ritz problem (4.27) and obtain $(\theta_j^{(i)}, x_j^{(i)})$
16: Compute the residual $r_j^{(i)} \leftarrow Hx_j^{(i)} - \theta_j^{(i)}x_j^{(i)}$
17: **if** $\|r_j^{(i)}\|/|\theta_j^{(i)}| < tol$ **then**
18: exit loop
19: **end if**
20: **end for**
21: **end for**

subspace iterations, the block size and the degree of the Chebyshev polynomial used. The number of SpMVs used in RMM-DIIS is the sum of the RMM-DIIS iterations for all desired eigenpairs.

In addition to SpMVs, some dense linear algebra operations are performed in these algorithms to orthonormalize basis vectors and to perform the Rayleigh-Ritz calculations. The cost of orthonormalization can become large if too many Lanczos iterations or block Lanczos iterations are performed. Such cost is relatively small in LOBPCG, RMM-DIIS and Chebyshev polynomial filtering subspace iterations.

In Table 4.1, we compare the memory usage of each method discussed above. Note that the first term for Lanczos, block Lanczos, LOBPCG and CheFSI in this table is generally the dominant term. We use $\mathcal{O}(c)$ to denote a small multiple (i.e., typically 2 or 3) of c. The number of iterations taken by a block Lanczos iteration k_{blocklan} is typically smaller than the number of Lanczos iterations k_{lan} when the same number of eigenpairs are computed by these methods. However, $k_{\text{blocklan}} \cdot n_b$ is often larger than k_{lan}. It is possible to use a smaller amount of memory in LOBPCG and CheFSI at the cost of performing more SpMMs. For example, if we were to explicity compute $HX^{(i+1)}$ and $HP^{(i)}$ in the LOBPCG algorithm to perform the Rayleigh-Ritz calculation instead of updating these blocks according to (4.21) and (4.22) respective, we can reduce the LOBPCG memory usage to $4n \cdot n_b + \mathcal{O}(9n_b^2)$. For the RMM-DIIS algorithm, we assume that we compute one eigenpair at a time. The parameter $s_{\max}$ is the maximum dimension of the DIIS subspace constructed to correct an approximate eigenvector. This parameter is often chosen to be between 10 and 20. If we batch the refinement of several eigenvector together to make use of SpMMs as we will discuss below, the memory cost of RMM-DIIS will increase by a factor of n_{ev}.

Method	**Memory cost**
Lanczos	$n \cdot (k_{Lan} + n_{ev}) + \mathcal{O}(k_{Lan}^2)$
block Lanczos	$n \cdot n_b \cdot (k_{blocklan} + n_{ev}) + \mathcal{O}((n_b \cdot k_{blocklan})^2)$
LOBPCG	$7n \cdot n_b + \mathcal{O}(9n_b^2)$
CheFSI	$4n \cdot n_b + \mathcal{O}(n_b^2)$
RMM-DIIS	$n \cdot (3n_{ev} + s_{\max})$

Table 4.1 A comparison of memory footprint associated with the Lanczos, block Lanczos, LOBPCG, CheFSI and RMM-DIIS methods.

4.2 Hybrid Algorithms

Among all methods discussed above, the Lanczos, block Lanczos and the LOBPCG methods as well as the Chebyshev polynomial filtered subspace iteration can all proceed with an arbitrary starting guess of the desired eigenvectors although they can all benefit from the availability of a good starting guess. As an eigenvector refinement method, the RMM-DIIS method requires a relatively more accurate approximation of the desired eigenvectors.

Therefore, a more effective way to use the RMM-DIIS method is to combine it with the other methods, i.e., we can start with Lanczos, block Lanczos, LOBPCG or CheFSI method and switch to RMM-DIIS when the approximate eigenvectors become sufficiently accurate.

The basis orthogonalization cost as well as the memory requirement become progressively higher in the Lanczos and block Lanczos method. Therefore, a notable benefit to switch from the Lanczos or block Lanczos methods to RMM-DIIS is to lower the orthogonalization cost and memory requirement.

Although the orthogonalization cost and memory requirement for the LOBPCG method is fixed throughout all LOBPCG iterations, the subspace (4.17) from which eigenvalue and eigenvector approximations are drawn becomes progressively more ill-conditioned as the norms of the vectors in (4.18) become smaller. The ill-conditioned subspace can make the LOBPCG algorithm numerically unstable even after techniques proposed in [26, 19] are applied. Therefore, it is desirable to switch from LOBPCG to RMM-DIIS, when the condition number of the subspace is not too large.

The orthogonalization cost and memory requirement for CheFSI are also fixed. The method is generally more efficient in the early subspace iterations when $T_n(H)$ is applied to a block of vectors in each iteration. As the approximate eigenvectors converge, applying $T_n(H)$ to a block of vectors in a single iteration results in a higher cost compared to RMM-DIIS that can refine each approximate eigenvector separately. Therefore, it also seems to be plausible to switch to RMM-DIIS, when approximate eigenvectors become sufficiently accurate in CheFSI.

4.3 Numerical examples

In this section, we compare and analyze the performance of algorithms presented in Section 4.1 using numerical examples. Through these examples, we demonstrate the advantage of using a hybrid LOBPCG/RMM-DIIS or block Lanczos/RMM-DIIS algorithm to solve nuclear configuration interaction eigenvalue problems. We also discuss how to address some of the practical problems to make the hybrid algorithm more efficient. We initially started with

the MATLAB experiments where testing the effectiveness of algorithms in terms of sheer SpMV numbers required was much simpler. Having seen the merit of those algorithms, we then proceeded to implement, and experiment with them in MFDn.

4.3.1 Test problems and computing platform

The test problems we use are the many-body Hamiltonian matrices associated with four different types of nuclei Li_6, Li_7, B_{11} and C_{12} , where the subscripts indicate the number of nucleons (protons and neutrons) in the nuclei. These Hamiltonian matrices are constructed in different configuration interaction model spaces labeled by the N_{max} parameter. In Table 4.2, we list the dimension of each matrix as well as the number of nonzero matrix elements in these matrices.

System	N_{max}	**Matrix dimension**	**#Non-zeros**
Li_6	6	197,822	106,738,802
Li_7	6	663,527	421,938,629
B_{11}	4	814,092	389,033,682
C_{12}	4	1,118,926	555,151,572

Table 4.2 Test problems used in the numerical experiments.

Before solving eigenvalue problems for Hamiltonians listed in Table 4.2, we first construct a good initial guess of the desired eigenvectors by computing the lowest few eigenvalues and the corresponding eigenvectors of smaller Hamiltonian matrices constructed from a lower dimensional configuration interaction model space labelled by smaller N_{max} values. Table 4.3 shows the dimensions and number of nonzeros in these smaller Hamiltonians. The initial guesses to the desired eigenvectors of the Hamiltonian matrices listed in Table 4.2 are obtained by padding the eigenvectors of the smaller Hamiltonian matrices by zeros to match the dimension of the original problems to be solved. As we can see, since the dimension of the problems listed in Table 4.3 are an order of magnitude or two smaller than the corresponding problems listed in Table 4.2, they can be solved relatively easily and quickly by almost any method.

All algorithms presented in Section 4.1 have been implemented in MATLAB which is ideal

System	N_{max}	Matrix Size	#Non-zeros
Li_6	4	17,040	4,122,448
Li_7	4	48,917	14,664,723
B_{11}	2	16,097	2,977,735
C_{12}	2	17,725	3,365,099

Table 4.3 The dimensions and number of nonzeros of Hamiltonian matrices for Li_6, Li_7, B_{11} and C_{12} that are constructed in a lower dimensional configuration model space labelled by a smaller N_{max} value.

for prototyping new algorithms. We perform the numerical experiments presented below on a single AMD EPYC 7763 socket on a Perlmutter CPU node maintained at the National Energy Research Scientific Computing (NERSC) Center. The EPYC socket contains 64 cores and we disabled hyperthreading so 64 OpenMP threads were used. For each test problem, we typically perform two sets of experiments for each algorithm. In the first set of experiments, we compute $n_{\text{ev}} = 5$ lowest eigenvalues and their corresponding eigenvectors. In the second set, we increase the number of eigenpairs to be computed to $n_{\text{ev}} = 10$. All calculations are performed in double precision arithmetic.

4.3.2 The performance of single method solvers

In this section, we report and compare the performance of the Lanczos, block Lanczos, LOBPCG, CheFSI and RMM-DIIS methods when they are applied to the test problems listed in Table 4.2.

For block methods such as the block Lanczos, LOBPCG and CheFSI methods, we set the block size, i.e., the number of vectors in the matrix $\mathbf{V}_j$ in (4.9), the matrix $X^{(i)}$ in (4.15), to $n_b = 8$ when computing the $n_{\text{ev}} = 5$ lowest eigenpairs of H, or to $n_b = 16$ when computing the $n_{\text{ev}} = 10$ lowest eigenpairs of H. SpMM is performed to multiply H with 8 or 16 vectors all at once. Even though RMM-DIIS is not a block method, n_{ev} SpMVs performed in the algorithm can be fused together as a single SpMM as we explain below.

For block methods, we choose the starting guess for each method as the eigenvectors of the Hamiltonian constructed in a smaller configuration space (with a smaller N_{max} value) listed in Table 4.3 padded with zeros to match dimensions of the Hamiltonian in the larger

configuration space (with a larger N_{max} value) as mentioned earlier. This is also used in the RMM-DIIS method which only requires a starting guess for each of the desired eigenpairs.

For the Lanczos algorithm, we take the initial guess v_0 to be the linear combination of augmented eigenvectors associated with the lowest n_{ev} eigenvalues of the Hamiltonian constructed from the smaller configuration space, i.e.,

$$v_0 = (\sum_{i=1}^{n_{\text{ev}}} \hat{z}_i)/n_{\text{ev}},$$

where $\hat{z}_i$ is the zero padded eigenvector associated with the ith eigenvalue of the Hamiltonian constructed from the smaller configuration space.

All methods are terminated when the relative residual norms or estimated residual norm associated with all desired eigenpairs are below the threshold of $\tau = 10^{-6}$. A relative residual norm for an approximate eigenpair (θ, z) is defined to be

$$\|Hz - z\theta\|/|\theta|.$$

For the Lanczos and block Lanczos methods, we use (4.7) and (4.13) to estimate the relative residual norm without performing additional Hamiltonian matrix and vector multiplications.

We use a 10th degree Chebyshev polynomial in the CheFSI method. The convergence of the algorithm depends on the choice of several parameters. The upper bound of the spectrum λ_{ub} is determined by first running 10 Lanczos iterations and using Rayleigh Ritz approximation to the largest eigenpairs (θ_{10}, u_{10}) to set λ_{ub} to $\theta_{10}+\|r_{10}\|$, where $r_{10} = Hu_{10} - \theta_{10}u_{10}$. We set the parameter λ_F simply to 0 because the desired eigenvalues are bound states of the nucleus of interest and are expected to be negative. We apply the technique of deflation for converged eigenvectors, i.e., once the relative residual norm of an approximate eigenpair falls below the convergence tolerance of 10^{-6}, we "lock" the approximate eigenvector in place and do not apply H to this vector in subsequent computations. These vectors will still participate in the Rayleigh-Ritz caculation performed in steps 11 and 12 of Algorithm 4.3 and be updated as part of the Rayleigh-Ritz procedure.

In Tables 4.4 and 4.5, we compare the performance of Lanczos, block Lanczos, LOBPCG, CheFSI and RMM-DIIS in terms of the total number SpMVs performed in each of these methods. It is clear from these tables that the Lanczos method uses the least number of SpMVs. However, the number of SpMVs used by both the LOBPCG, block Lanczos and RMM-DIIS is within a factor of 3 when $n_{ev} = 5$ eigenpairs are computed. Because 8 SpMVs can be fused as a single SpMM, which is more efficient, in the block Lanczos and LOBPCG method, the total wall clock time used by these methods can be less than that used by the Lanczos method. Five SpMVs can also be fused in the RMM-DIIS method, even though the algorithm targets each eigenvalue separately. Because different eigenvalues may converge at a different rate, we may switch to using SpMVs when some of the eigenpairs converge. We discuss whether switching to SpMVs is beneficial later in this section.

Table 4.5 shows that the number of SpMVs used in the Lanczos algorithm increases only slightly when we compute $n_{ev} = 10$ eigenpairs. However, the number of SpMVs required in the block Lanczos, LOBPCG, CheFSI and RMM-DIIS increase at a higher rate. This is mainly due to the fact that once a sufficiently large Krylov subspace has been constructed, we can easily obtain approximations to more eigenpairs without enlarging the subspace much further.

System	**Lanczos**	**Block Lanczos**	**LOBPCG**	**ChebFSI**	**RMM-DIIS**
Li_6	95	208	184	480	271
Li_7	109	280	240	960	291
B_{11}	82	240	192	950	152
C_{12}	106	248	192	890	181

Table 4.4 SpMV count for different algorithms on MATLAB to compute five lowest eigenvalues.

Figures 4.2, 4.3, 4.4, 4.5, 4.6 show the convergence history of the Lanczos, block Lanczos, LOBCPG, CheFSI and RMM-DIIS methods. In these figures, we plot the relative residual norm of each approximate eigenpair with respect to the iteration number. We can clearly see that accurate approximations to some of the eigenpairs start to emerge in the Lanczos method when the dimension of the Krylov subspace (i.e., iteration number) is sufficiently

System	Lanczos	Block Lanczos	LOBPCG	ChebFSI	RMM-DIIS
Li_6	114	464	464	690	686
Li_7	192	512	464	1350	884
B_{11}	180	480	432	2470	Convergence failure
C_{12}	164	480	400	2300	499

Table 4.5 SpMV count for different algorithms on MATLAB to compute ten lowest eigenvalues.

large. Eigenpairs do not converge at the same rate. The smallest eigenvalue converges first, followed by the second, third, fourth and the fifth eigenvalues. However, these eigenvalues do not necessarily have to converge in order. Although the relative residual for each eigenpair eventually goes below the convergence threshold of 10^{-6}, the reduction of the relative residual norm is not monotonic with respect to the Lanczos iteration number. The relative residual can sometimes increase after the Krylov subspace becomes sufficiently large and new spectral information becomes available.

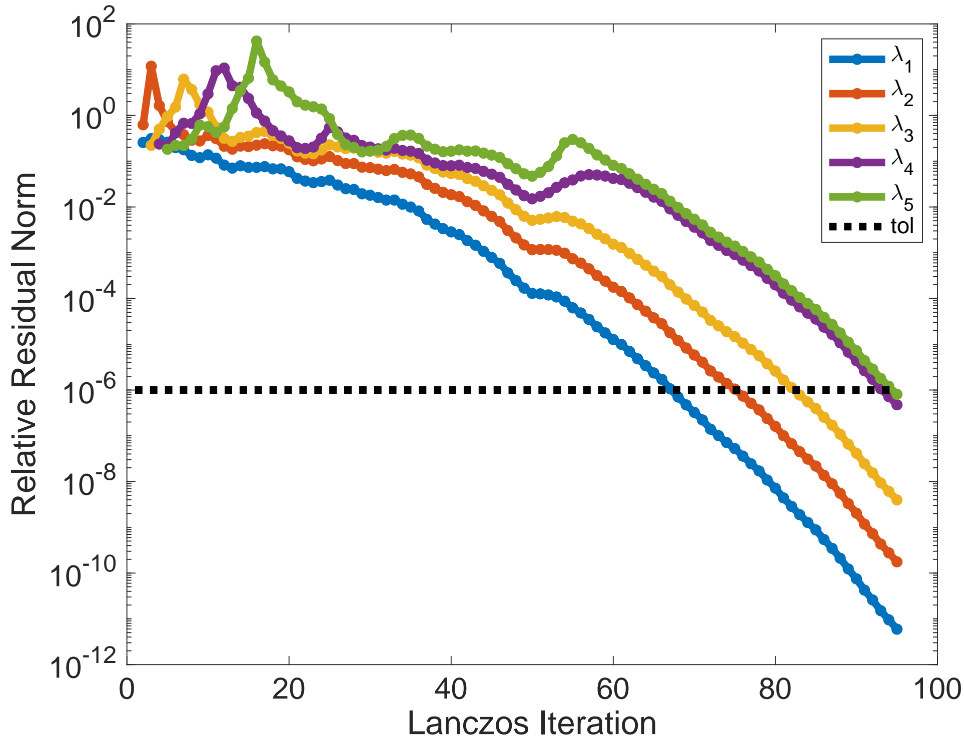


Figure 4.2 The convergence of the lowest five eigenvalues of the Li6 Hamiltonian in the Lanczos algorithm.

All approximate eigenpairs appear to converge at a similar rate in the block Lanczos and

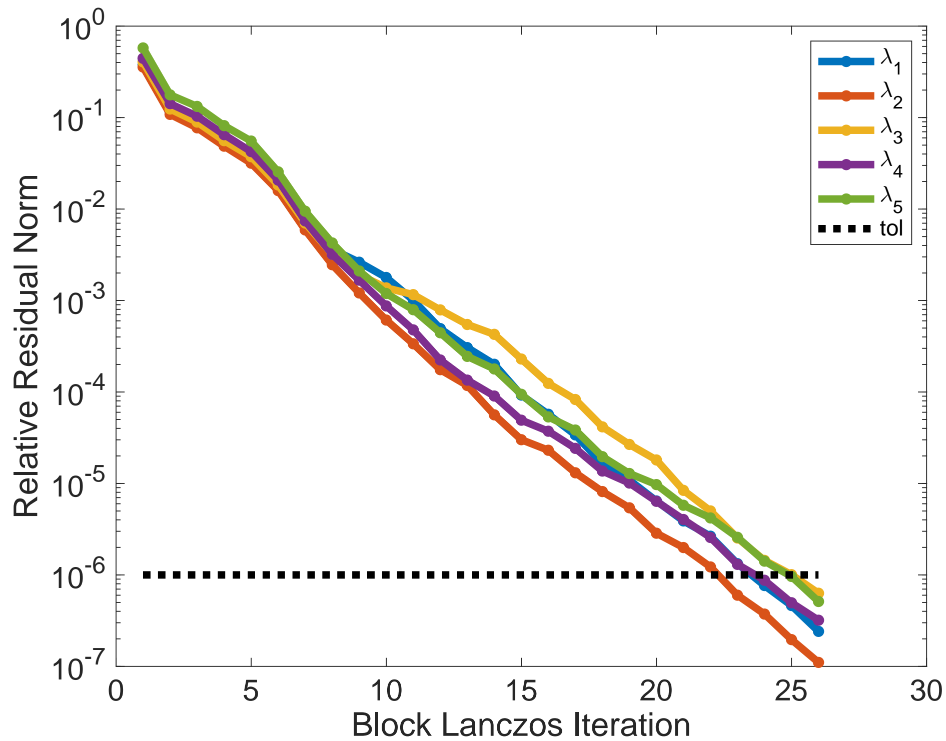


Figure 4.3 The convergence of the lowest five eigenvalues of the Li6 Hamiltonian in the block Lanczos algorithm.

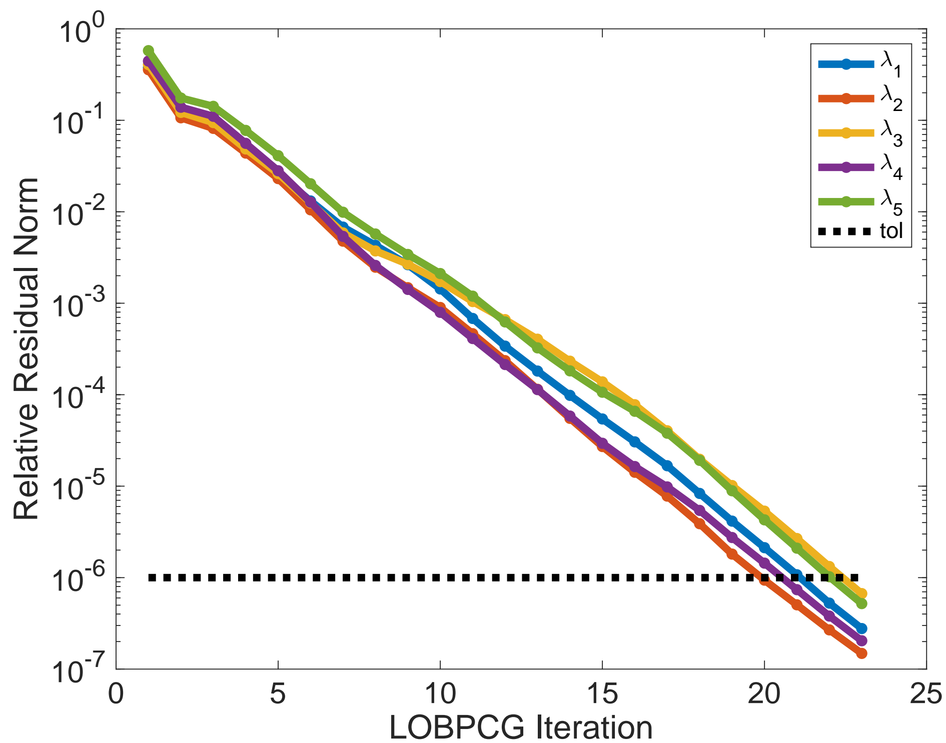


Figure 4.4 The convergence of the lowest five eigenvalues of the Li6 Hamiltonian in the LOBPCG algorithm.

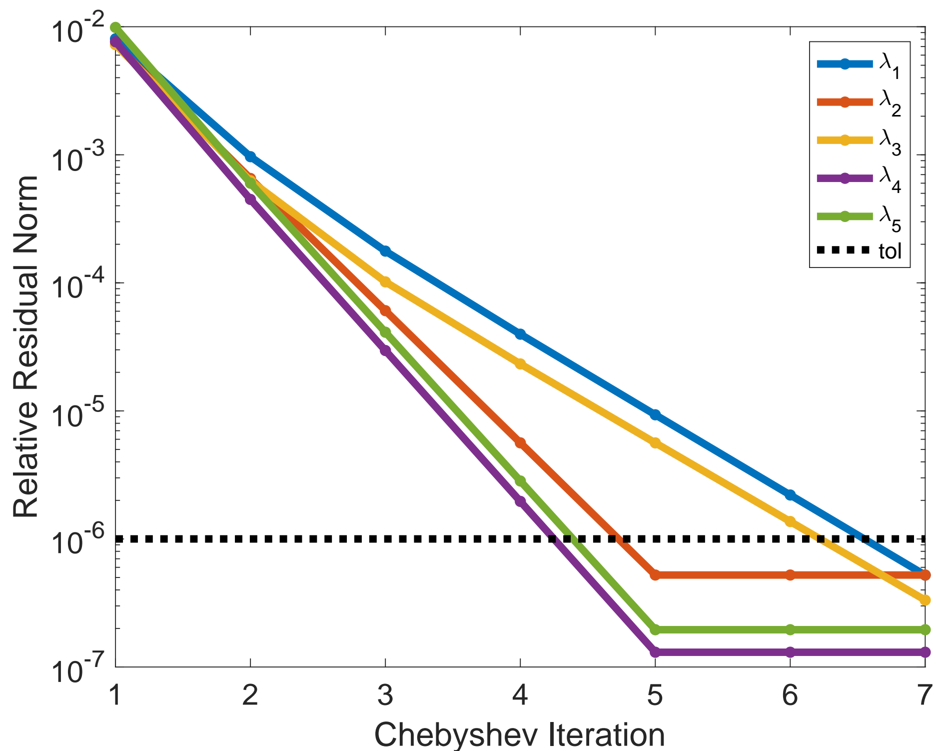


Figure 4.5 The convergence of the lowest five eigenvalues of the Li6 Hamiltonian in the Chebyshev algorithm.

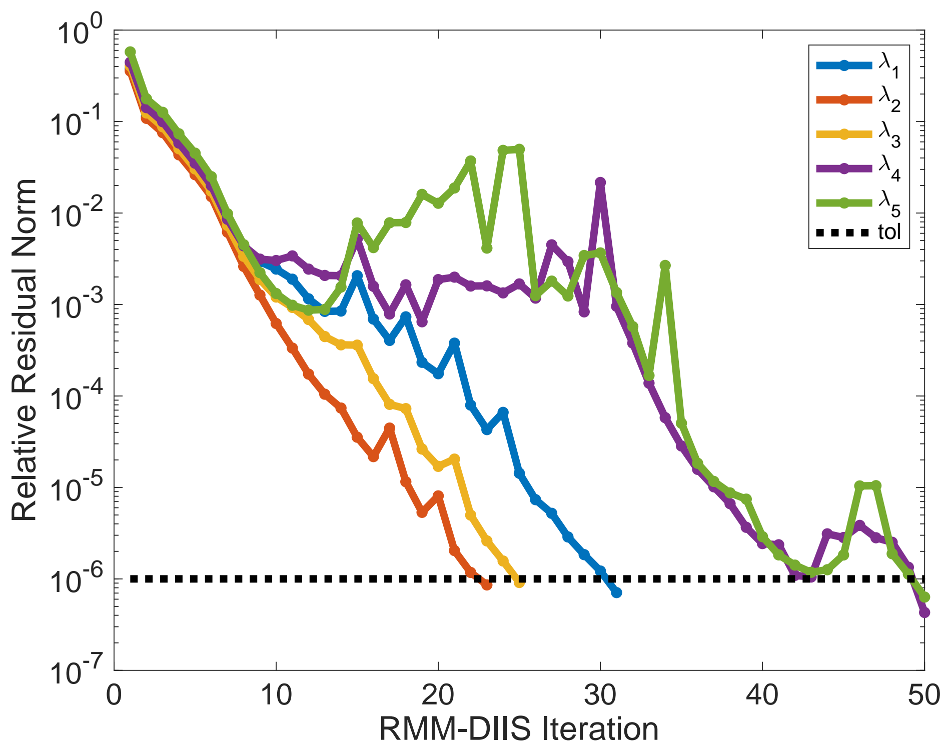


Figure 4.6 The convergence of the lowest five eigenvalues of the Li6 Hamiltonian in the RMM-DIIS algorithm.

LOBPCG methods. This is one of the advantages of a block method. The LOBPCG method performs slightly better in terms of the total number of SpMMs (or the number of iterations) used. This is due to the fact that LOBPCG makes use of an effective preconditioner.

The CheFSI is also a block method. Figure 4.5 shows that only five subspace iterations are required to obtain converged λ_2, λ_4 and λ_5. Two more subspace iterations are required to obtain converged λ_1 and λ_3. The difference in the convergence rates for different eigenvalues is likely due to the variation in the contributions from different eigenvectors in the initial subspace constructed from the eigenvectors of the Hamiltonian associated with a smaller configuration space. Although the number of subspace iterations used in CheFSI is relatively small, each subspace iteration needs to perform $n_b \cdot m$ SpMVs, where n_b is the number of vectors in the initial subspace and m is the degree of the Chebyshev polynomial. When $n_b = 8$ and $m = 10$, a total of 480 SpMVs are used to compute the lowest five eigenpairs as reported in Table 4.4. Note that this count is less than $8 \times 10 \times 7 = 560$ because a deflation scheme that locks the converged eigenvector is used in CheFSI. When $n_b = 16$, $m = 10$, a total of 690 SpMVs are used to compute 10 lowest eigenpairs, as reported in Table 4.5. Because these SpMV counts are significantly higher than those used in other methods, CheFSI appears to be not competitive for solving this type of eigenvalue problem. Therefore, from this point on, we will not discuss this method any further.

The convergence of the RMM-DIIS method is interesting. We observe from Figure 4.6 that the first three eigenvalues of the Li_6 Hamiltonian converge relatively quickly. The number of RMM-DIIS iterations required to reach convergence is 31 for the first eigenvalue, 25 for the third eigenvalue and 23 for the second eigenvalue. Altogether, 79 SpMVs are used to obtain accurate approximations to the three smallest eigenvalues and their corresponding eigenvectors, which is almost same as the 78 SpMVs used in the Lanczos algorithm for obtaining these three eigenpairs. However, the fourth and fifth eigenvalues take much longer to converge. Furthermore, we also observe that eigenvalues do not converge in order. In fact, the second smallest eigenvalue converges first, followed by the third and the first small-

est eigenvalues. The RMM-DIIS procedure that starts with the initial guess to the fourth smallest eigenvalue ends up converging to the fifth eigenvalue, whereas the procedure that starts with initial guess to the fifth eigenvalue converges to the fourth eigenvalue. However, it is important to note that RMM-DIIS iterations that start with different initial guesses all converge to different eigenpairs even though no orthogonalization is performed between approximate eigenvectors produced in different RMM-DIIS iterations. Table 4.5 shows that more RMM-DIIS iterations are needed to obtain accurate approximations to larger eigenvalues deeper inside the spectrum of Li_6 Hamiltonian, partly because the initial guesses to the interior eigenvalues are poor in this case.

4.3.3 Hybrid methods

As we discussed in Section 4.1, the RMM-DIIS algorithm can be very effective when a good initial guess to the target eigenvector is available. Figure 4.7 shows that RMM-DIIS converges rapidly when the initial guesses to the eigenvectors associated with the lowest 5 eigenvalues are chosen to be the approximated eigenvectors produced from 10 LOBPCG iterations. As a result, we can combine RMM-DIIS with either the block Lanczos or the LOBPCG algorithm to devise a hybrid algorithm that first runs a few block Lanczos or LOBPCG iterations and then use the RMM-DIIS method to refine approximated eigenvectors returned from the block Lanczos or the LOBPCG algorithm simultaneously.

Although it is possible to combine the RMM-DIIS algorithm with the Lanczos algorithm, such a hybrid algorithm is less attractive, partly because convergence rates for different eigenpairs in a Lanczos algorithm are different. As we can see from Figure 4.2 that the left most eigenvalues typically converge much faster than larger eigenvalues. As a result, the RMM-DIIS method can only be effectively used to refine the eigenvectors associated with larger eigenvalues when they become sufficiently accurate. At that point, the left most eigenvalues are likely to have converged already. Furthermore, because the Lanczos algorithm tends to be less efficient than the block Lanczos or the LOBPCG algorithms when the multiplication of the sparse Hamiltonian H with several vectors can be implemented

efficiently, we will not consider the hybrid Lanczos and RMM-DIIS method here.

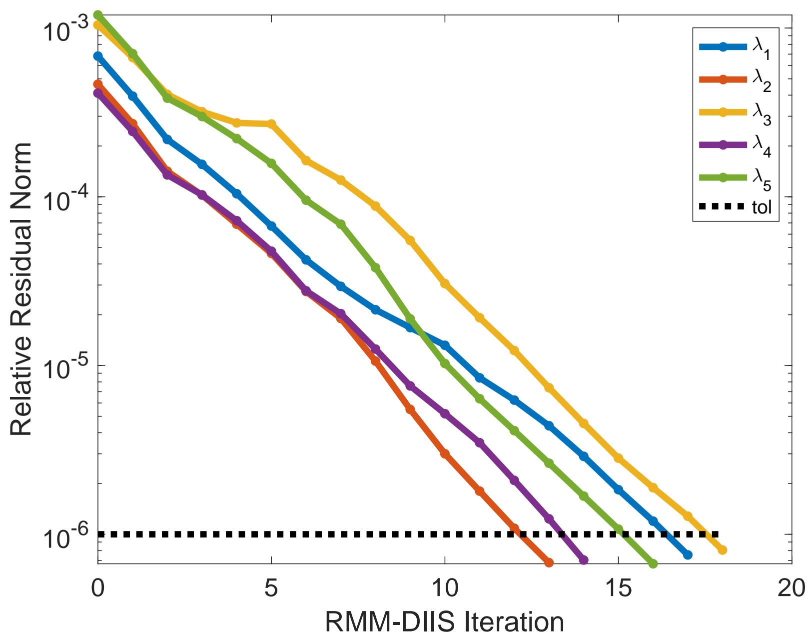


Figure 4.7 The convergence of the lowest five eigenvalues of the Li6 Hamiltonian in the RMM-DIIS algorithm after 10 iterations of LOBPCG.

4.3.3.1 When to switch

A practical question we need to address in order to implement a hybrid block Lanczos /RMM-DIIS or LOBPCG/RMM-DIIS eigensolver is to decide when to switch from one algorithm to another. Running more block Lanczos or LOBPCG iterations will produce more accurate approximations to the desired eigenvectors that can then be quickly refined by the RMM-DIIS algorithm. But the cost of running block Lanczos or LOBPCG may dominate the overall cost of the computation. On the other hand, running fewer block Lanczos or LOBPCG iterations may produce a set of approximate eigenvectors that require more RMM-DIIS iterations. There is clearly a trade off.

Tables 4.6 shows the total number of SpMVs required in a hybrid block Lanczos/RMM-DIIS algorithm in which 5, 10, 15 or 20 block Lanczos iterations are performed first and followed by the RMM-DIIS procedure. We observe that this hybrid scheme uses fewer total SpMVs for all test problems than that used in a simple block Lanczos or RMM-DIIS run. A similar observation can be made from Table 4.7 when 10 lowest eigenvalues are computed

except that for Li_7, the hybrid algorithm that starts with 5 block Lanczos iterations uses more SpMVs than the non-hybrid block Lanczos only algorithm. The optimal number of block Lanczos iterations we should perform (in terms of the total SpMV count) before switching to the RMM-DIIS procedure is problem dependent. Because good approximations to interior eigenvalues only emerge when the Krylov subspace produced by the block Lanczos iteration is sufficiently large, more block Lanczos iterations are required in the hybrid algorithm to yield an optimal SpMV count when 10 eigenvalues and the corresponding eigenvectors are needed, as we can see in Table 4.7.

System	**5 block Lan**	**10 block Lan**	**15 block Lan**	**20 block Lan**	**all block Lan**
Li_6	183	163	173	185	208
Li_7	205	218	234	221	280
B_{11}	170	178	195	199	240
C_{12}	179	180	179	201	248

Table 4.6 SpMV count in hybrid block Lanczos/RMM-DIIS for computing five lowest eigenvalues.

System	**5 block Lan**	**10 block Lan**	**15 block Lan**	**20 block Lan**	**all block Lan**
Li_6	422	361	347	367	464
Li_7	544	422	418	402	512
B_{11}	419	422	392	382	480
C_{12}	430	415	369	369	480

Table 4.7 SpMV count in hybrid block Lanczos/RMM-DIIS for computing 10 lowest eigenvalues.

A hybrid LOBPCG and RMM-DIIS algorithm also performs better when a sufficiently large number of LOBPCG iterations are performed first and followed by the RMM-DIIS procedure as can be seen in Tables 4.8 and 4.9.

System	**5 LOBPCG**	**10 LOBPCG**	**15 LOBPCG**	**20 LOBPCG**	**all LOBPCG**
Li_6	191	158	163	165	184
Li_7	211	200	190	194	240
B_{11}	260	162	159	165	192
C_{12}	186	157	153	166	192

Table 4.8 SpMV count in hybrid LOBPCG/RMM-DIIS for computing five lowest eigenvalues.

System	5 LOBPCG	10 LOBPCG	15 LOBPCG	20 LOBPCG	all LOBPCG
Li_6	432	344	337	356	464
Li_7	529	398	364	365	464
B_{11}	758	397	334	355	423
C_{12}	523	340	317	334	400

Table 4.9 SpMV count in hybrid LOBPCG/RMM-DIIS for computing 10 lowest eigenvalues.

From Tables 4.6, 4.7, 4.8 and 4.9, we observe that the total SpMV count is optimal when the number of block Lanczos or LOBPCG iterations is sufficiently large, but not too large. However, the optimal number of block Lanczos/LOBPCG iterations is problem dependent.

One practical way to determine when to switch from block Lanczos or LOBPCG to RMM-DIIS is to examine the change in the approximate eigenvalue. Because RMM-DIIS can be viewed as an eigenvector refinement method, it works well when an approximate eigenvalue becomes sufficiently accurate while the corresponding approximate eigenvector still needs to be corrected.

Since we do not know the exact eigenvalues in advance, we use the average changes in the relative difference between approximate eigenvalues obtained in two consecutive iterations (e.g., the $k-1$st and the kth iterations) defined as

$$\tau = \frac{1}{n_{\rm ev}} \sqrt{\sum_{j=1}^{n_{\rm ev}} \left(\frac{\theta_j^{(k)} - \theta_j^{(k-1)}}{\theta_j^{(k)}} \right)^2 }. \tag{4.29}$$

as a metric to decide when to switch from block Lancos or LOBPCG to RMM-DIIS.

Tables 4.10 and 4.11 show the optimal SpMV count of the hybrid block Lanczos/RMM-DIIS and LOBPCG/RMM-DIIS algorithms when computing five or 10 lowest eigenvalues, respectively. Unlike Tables 4.6, 4.7, 4.8 and 4.9 where we show the results only for $5, 10, 15$ or 20 iterations, these optimal numbers are found by exhaustively trying every possible iteration count of block Lanczos and LOBPCG before switching to RMM-DIIS. Along with the optimal numbers, Tables 4.10 and 4.11 also show the SpMV count of the hybrid algorithms when block Lanczos and LOBPCG procedures are stopped as τ goes below 10^{-4} and 10^{-7}.

Table 4.11 shows that the number of SpMV counts in both the hybrid block Lanc-

zos /RMM-DIIS and LOBPCG/RMM-DIIS algorithms appear to be nearly optimal when $\tau \leq 10^{-7}$. However, when fewer eigenvalues are needed, we can possibly stop the block Lanczos procedure in the hybrid block Lanczos/RMM-DIIS algorithm when τ is below 10^{-4} as shown in Table 4.10. The $\tau \leq 10^{-7}$ criterion still seems to be a good one for the hybrid LOBPCG/RMM-DIIS algorithm even when fewer eigenpairs are needed.

System	**Opt Hyb Block Lan**	$\tau \leq 10^{-4}$	$\tau \leq 10^{-7}$	**Opt Hyb LOBPCG**	$\tau \leq 10^{-4}$	$\tau \leq 10^{-7}$
Li_6	157	194	169	156	187	159
Li_7	205	211	224	190	210	190
B_{11}	166	176	195	158	188	159
C_{12}	172	180	179	152	169	153

Table 4.10 SpMV count for hybrid algorithms wrt stopping threshold on MATLAB to compute five lowest eigenvalues.

System	**Opt Hyb Block Lan**	$\tau \leq 10^{-4}$	$\tau \leq 10^{-7}$	**Opt Hyb LOBPCG**	$\tau \leq 10^{-4}$	$\tau \leq 10^{-7}$
Li_6	340	384	346	330	393	335
Li_7	401	515	418	363	478	363
B_{11}	379	525	384	332	625	333
C_{12}	362	408	368	312	437	317

Table 4.11 SpMV count for hybrid algorithms wrt stopping threshold on MATLAB to compute 10 lowest eigenvalues.

4.3.3.2 Optimal implementation

As we indicated earlier, the SpMV count may not be the best metric to evaluate the performance of a block eigensolver that performs SpMVs for a block of vectors as a SpMM operation. It has also been shown in [2, 51] that on many-core processors, the LOBPCG algorithm outperforms the Lanczos algorithm even though its SpMV count is much higher. The reason that a block algorithm can outperform a single vector algorithm such as the Lanczos method on a many-core processor is that SpMM can take advantage of high concurrency and memory locality.

The performance of SpMV and SpMM can be measured in terms of number of floating point operations performed per second (GFLOPS). Tables 4.12 and 4.13 show the SpMM GFLOPS measured in the LOBPCG algorithm is much higher than the SpMV GFLOPS measured in the Lanczos algorithm.

As a result, we can evaluate the performance of a block eigensolver by diving the actual SpMV count by the ratio between SpMM and SpMV GFLOPS to obtain an "effective" SpMV count.

For example, because the SpMM and SpMV GFLOPS ratio is $24.2/4.1 \approx 5.9$ for Li_6, the effective SpMV count for performing 23 iterations of the LOBPCG algorithm with a block size 8 is $23 \times 8/5.9 \approx 31$ which is much lower than the 95 SpMVs performed in the Lanczos method, even though the actual number of SpMVs performed in the LOBPCG iteration is $23 \times 8 = 184 > 95$.

System	SpMV GFLOPS	SpMM GFLOPS	Ratio
Li_6	4.1	24.2	5.9
Li_7	6.7	36.0	5.4
B_{11}	6.4	33.5	5.2
C_{12}	6.0	28.0	4.7

Table 4.12 GFLOPs achieved by the SpMV/SpMM kernels within MFDn on one AMD EPYC node with 64 threads. The sparse Hamiltonian is applied to 8 vectors in SpMM.

System	SpMV GFLOPS	SpMM GFLOPS	Ratio
Li_6	4.1	38.9	9.5
Li_7	6.7	56.0	8.4
B_{11}	6.4	50.3	7.9
C_{12}	6.0	47.5	7.9

Table 4.13 GFLOPs achieved by the SpMV/SpMM kernels within MFDn on one AMD EPYC node with 64 threads. The sparse Hamiltonian is applied to 16 vectors in SpMM.

Strictly speaking, RMM-DIIS is a single vector method, i.e., in each RMM-DIIS run, we refine one specific eigenvector associated with a target eigenvalue which has become sufficiently accurate as discussed earlier. However, because the refinement of different eigenvectors can be performed independently from each other, we can perform several RMM-DIIS refinements simultaneously. The simultaneous RMM-DIIS runs can be implemented by batching the SpMVs in each RMM-DIIS iteration together as a single SpMM. This step constitutes the major cost of the RMM-DIIS method. The least squares problems given in

Eqs. 4.26- 4.28 for different eigenvectors can be solved in sequence in each step, since they do not cost much computation.

Because different eigenvectors may converge at different rates as we have already seen in Figure 4.6, we need to decide what to do when one or a few eigenvectors have converged. One possibility is to decouple the batched RMM-DIIS method after a certain number of eigenvectors have converged, and switch from using a single SpMM in a coupled RMM-DIIS implementation to using several SpMVs in a decoupled RMM-DIIS implementation. Another possibility is to just keep using the coupled RMM-DIIS with a single SpMM in each step without updating the eigenvectors that have already converged. In this case, the SpMM calculation performs more floating point operations than necessary. However, because an SpMM can be carried out at a much higher GFLOPs than an SpMV, the overall performance of the computation may not be degraded even with the extra computation.

In Figure 4.8, we show the effective number of SpMVs performed in several hybrid block Lanczos and RMM-DIIS runs for the Li_6 testcase. The horizontal axis represents the number of block Lanczos iterations performed before we switch to RMM-DIIS. The blue dots show the number of effective SpMVs used in the hybrid method when we switch from SpMM to SpMV after one eigenvector has converged. Similarly, the red, orange and magenta dots show the effective SpMV counts when we switch from SpMM to SpMV after two, three and four eigenvectors have converged respectively. The green dots show the effective number of SpMVs when we always use SpMM regardless of how many eigenvectors have converged. As we can see from this figure, the number of effective SpMVs is relatively high when we switch to RMM-DIIS after a few block Lanczos iterations. This is because it will take RMM-DIIS longer to converge if the initial eigenvector approximations produced from the block Lanczos iterations are not sufficiently accurate. Regardless of when we switch from block Lanczos to RMM-DIIS, using SpMM throughout the RMM-DIIS algorithm appears to almost always yield the lowest effective SpMV count. We can also see that the difference in the effective SpMV count is relatively large when we switch from block Lanczos to RMM-

DIIS too early. This is understandable because some of the approximate eigenvectors are more accurate than others when we terminate the block Lanczos iteration too early. As a result, the number of RMM-DIIS steps required to reach convergence may vary significantly from one eigenvector to another. The difference in the rate of convergence prevents us from batching several SpMVs into a single SPMM. If we switch after more block Lanczos iterations have been performed, this difference becomes quite small as all approximate eigenvectors are sufficiently accurate and converge more or less at the same rate. The relative difference (τ) between eigenvalue approximations from two consecutive RMM-DIIS iterations falls below 10^{-7} at the 13th block Lanczos iteration. If we switch to RMM-DIIS at that point and use SpMM throughout the RMM-DIIS iteration, the number of effective SpMVs used in the hybrid scheme is about 40, which is slightly higher than the optimal 32 effective SpMVs required if we were to switch to RMM-DIIS after 23 block Lanczos iterations.

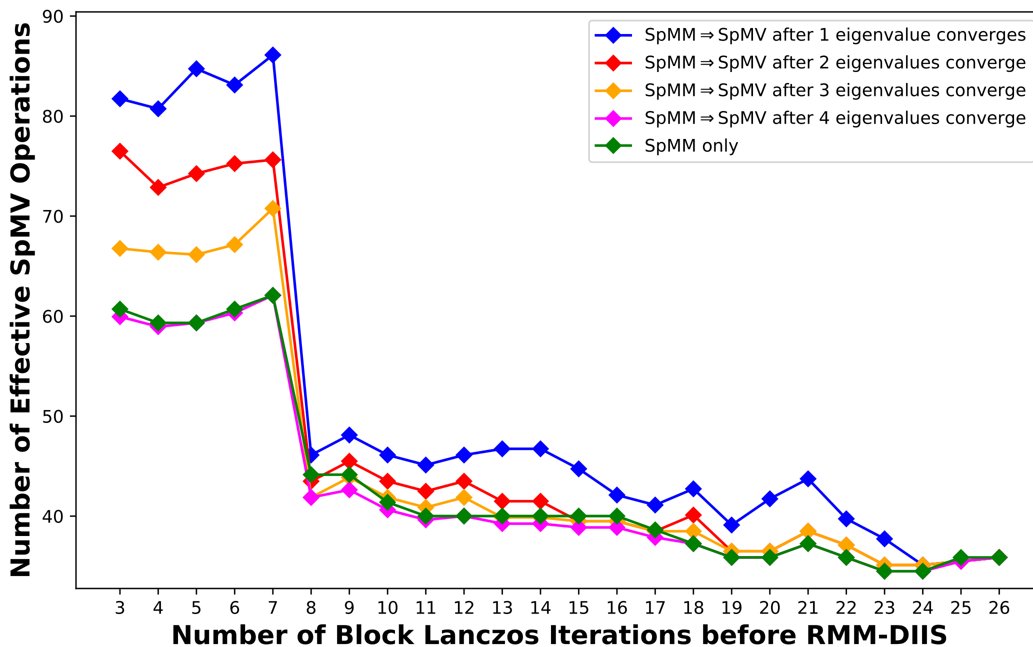


Figure 4.8 Total effective SpMV cost of the block Lanczos/RMM-DIIS algorithm to compute the lowest five eigenvalues for Li_6 w.r.t. switching strategy from SpMM to SpMV within RMM-DIIS.

An early switch allows us to keep the basis orthogonalization cost of the block Lanczos algorithm as low as possible. As we can see from Figure 4.9, the cost of orthogonalization as a percentage of the SpMM cost can become significantly higher as we perform more block Lanczos iterations. In particular, for all test problems, the orthogonalization cost exceeds 50% of the SpMM cost after 20 block Lanczos iterations. We note that the performance shown in Figure 4.9 is measured from the wallclock time of an implementation of the block Lanczos algorithm in the MFDn software executed on a single node of the Perlmutter using 64 threads.

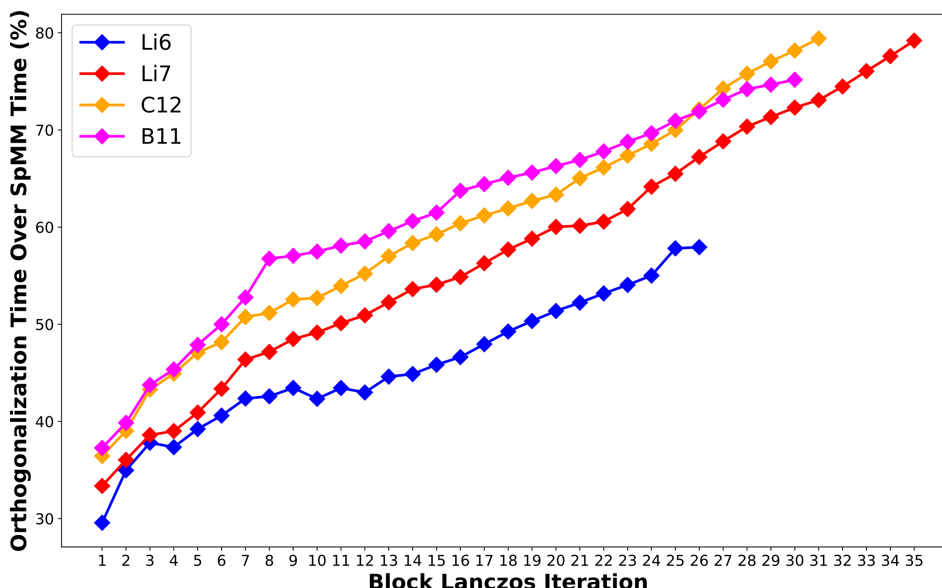


Figure 4.9 The percentage of cumulative time spent on orthogonalization compared to SpMM in the block Lanczos algorithm for all test problems.

Figure 4.10 shows that it is beneficial to use SpMM throughout the hybrid LOBPCG /RMM-DIIS algorithm even after some of the eigenvectors have converged. When a sufficient number of LOBPCG iterations have been performed, the total SPMV count does not vary much. In this case, we should terminate LOBPCG as early as possible to avoid potential numerical instabilities that can be introduced by the numerical rank deficiency of the preconditioned residual vectors [19]. Figure 4.11 shows that the estimated condition number

of the subspace from which approximated eigenvalues and eigenvectors are extracted in the LOBPCG algorithm increases rapidly as we perform more LOBPCG iterations. Although the optimal SpMV count is attained when we switch to RMM-DIIS after 20 LOBPCG iterations, it is not unreasonable to terminate LOBPCG sooner, for example, after 12 iterations, when the estimated condition number of the LOBPCG subspace is around 10^9. This is also the point at which the average relative change in the approximations to the desired eigenvalues τ just moves below 10^{-7}. Therefore, the previously discussed strategy of using $\tau < 10^{-7}$ to decide when to switch to RMM-DIIS works well. Even though this strategy would lead to a slight increase in the number of effective SpMV operations compared with the optimal effective SpMV count achieved when we switch to RMM-DIIS after 20 iterations, it makes the hybrid algorithm more robust and stable. We should also note that in a practical calculation the optimal effective SpMV count and when the optimality is achieved is unknown a priori. This optimality is problem dependent, and is also architecture dependent.

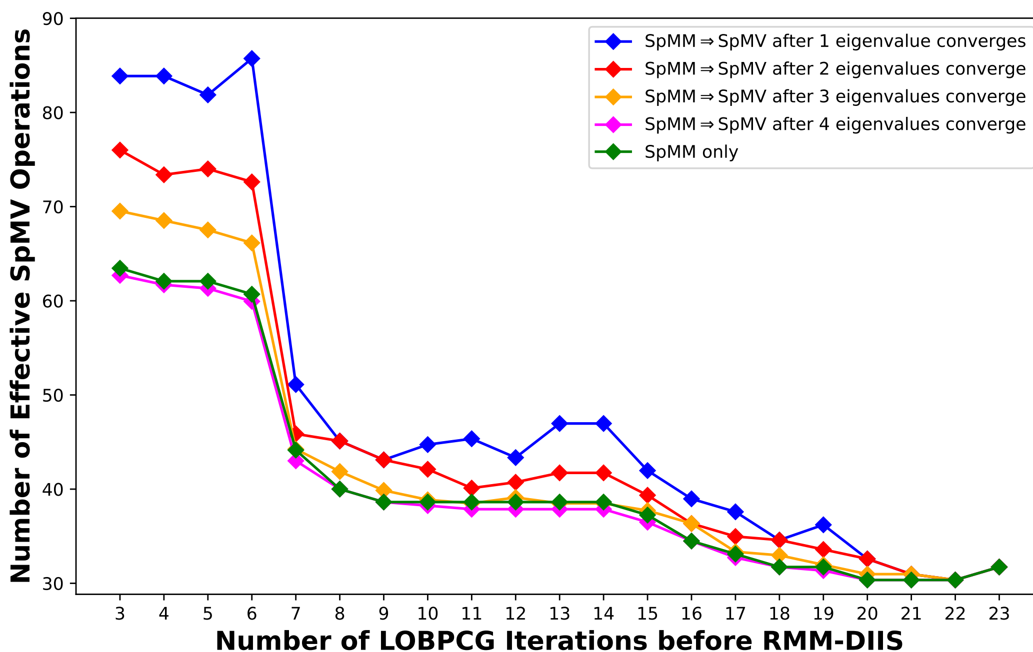


Figure 4.10 Total effective SpMV cost of the LOBPCG/RMM-DIIS algorithm to compute the lowest five eigenvalues for Li_6 w.r.t. switching strategy from SpMM to SpMV within RMM-DIIS.

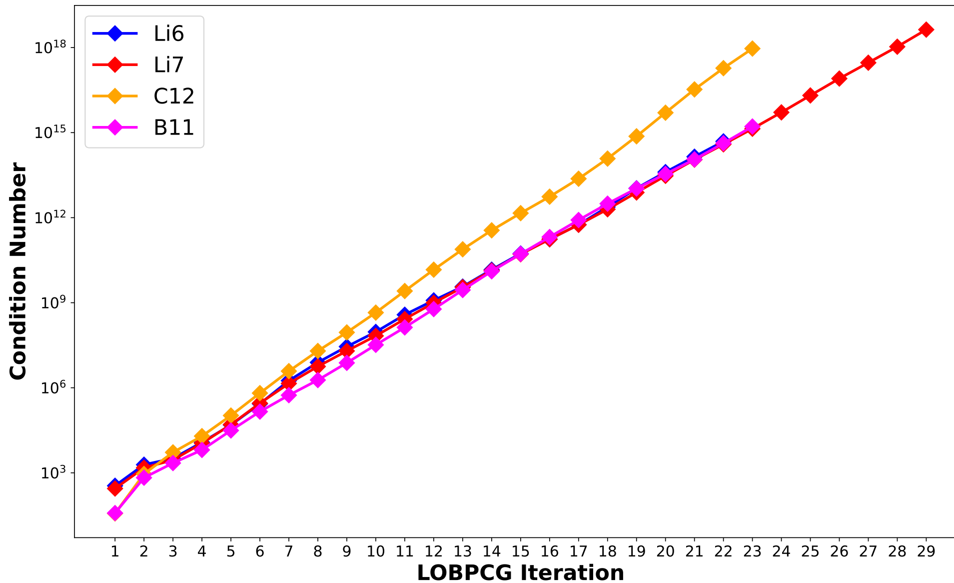


Figure 4.11 The condition number of the LOBPCG subspace from which approximate eigenpairs are extracted to compute the lowest 5 eigenvalues of the Li_6 Hamiltonian.

4.4 Conclusion of This Work

In this work, we examine and compare a few iterative methods for solving large-scale eigenvalue problems arising from nuclear structure calculations. We observe that the block Lanczos and LOBPCG methods are generally more efficient than the standard Lanczos method and the stand-alone RMM-DIIS method in terms of the effective number of SpMVs. The Chebyshev filtering based subspace iteration method is not competitive with other methods even though it requires the least amount of memory and has been found to be very efficient for other applications. We show that by combining the block Lanczos or LOBPCG algorithm with the RMM-DIIS algorithm, we obtain a hybrid solver that can outperform existing solvers. The hybrid LOBPCG/RMM-DIIS method is generally more efficient than block Lanczos/RMM-DIIS when a good preconditioner is available. The use of RMM-DIIS in the block Lanczos/RMM-DIIS hybrid algorithm allows us to limit the orthogonalization cost in the block Lanczos iterations. In the LOBPCG/RMM-DIIS hybrid algorithm, the use of RMM-DIIS allows us to avoid the numerical instability that may arise

in LOBPCG when the residuals of the approximate eigenpairs become small. We discuss the practical issue of how to decide when to switch from block Lanczos or LOBPCG to RMM-DIIS. A strategy based on monitoring the averaged relative changes in the desired approximate eigenvalues has been found to work well. Although the RMM-DIIS method is a single vector refinement scheme, we show the SpMVs in multiple independent RMM-DIIS iterations targeting different eigenpairs can be batched together and implemented as a single SpMM. Such a batching scheme significantly improves the performance of the hybrid solver and is found to be useful even after some of the approximate eigenpairs have converged.

CHAPTER 5

SCALING EIGENSOLVERS TO HUNDREDS OF GPUS

In previous chapters, we focused on accelerating our sparse eigensolvers through asynchronous execution models and hybrid algorithms. Now, we turn our attention to accelerating them on heterogeneous architectures through an efficient use of GPU accelerators and inter-GPU communication calls. This efficient use allows us to scale Lanczos and LOBPCG to hundreds of GPUs. For this work, we particularly focus on the implementation of these eigensolvers within the distributed memory version of Many-body Fermion Dynamics for nuclei (MFDn).

The direct solution of the quantum many-body problem transcends several areas of physics and chemistry. MFDn is the state-of-the-art Configuration Interaction (CI) code that aim to solve for the structure of light nuclei by direct diagonalization of the nuclear many-body Hamiltonian in a harmonic oscillator single-particle basis [54]. In MFDn, the nuclear Hamiltonian is evaluated in a large harmonic oscillator single-particle basis and diagonalized by iterative techniques to obtain the low-lying eigenvalues and eigenvectors.

One key feature of a CI calculation is the large dimension of the Hamiltonian matrix it can produce. The dimension of the matrix characterizes the size of the many-body basis used to represent a nuclear many-body Hamiltonian. This problem size highly depends on the N_{max} value, which is the maximum number of oscillator quanta above the minimum for a given nucleus. Higher N_{max} values yield more precise results, but at the expense of an exponential growth in problem size. The dimension of the matrices can easily exceed a billion, and the number of nonzeros a trillion, with a higher number of N_{max} and with more realistic interactions [2].

The Lanczos and LOBPCG algorithms have been implemented in MFDn to obtain the low-lying eigenpairs. Since the growing problem size makes it infeasible for the Hamiltonian to be stored on a single node, these algorithms have been implemented for distributed memory architectures. To be precise, the implementation of Lanczos and LOBPCG uses a hybrid

MPI/OpenMP parallelization scheme, and has been optimized [14] to achieve scalable performance for distributed memory many-core CPU systems such as the Cori KNL system, who retired in 2022 after 13 years at the National Energy Research Scientific Computing (NERSC) Center.

The increased availability of high performance computing platforms equipped with general purpose GPUs has motivated the MFDn developers to consider modifying the many-core CPU implementation, which is written in Fortran 90, to enable it to run efficiently on accelerator based systems. Rewriting the code using, e.g., CUDA Fortran [47] or OpenCL programming models [41] would take a substantial amount of work. Therefore, the developers decided to use the OpenACC directive based programming model in combination with CUDA-aware MPI [60, 23]. They show how to incorporate OpenACC for sparse eigensolvers in [40].

In their work, they also present numerical examples that demonstrate the performance improvement achieved by using OpenACC directives. They obtained a significant speedup with their MPI/OpenACC approach over their previous hybrid MPI/OpenMP scheme. Although their work showed promising initial results in regards to utilizing GPUs for eigensolvers, further analysis indicated that there was room for improvement on large-scale to capitalize GPU resources to a better extent.

With this work, we aim to show the limitations of this initial GPU-parallel version that uses MPI/OpenACC scheme. We then propose numerous schemes to improve the performance of Lanczos and LOBPCG on large-scale multi-node multi-GPU experiments. For these experiments, we use Perlmutter nodes equipped with NVIDIA A100 GPUs at NERSC. We note that we apply the proposed improvements to SpMV and SpMM, which are by far the most time-consuming portions of these algorithms.

The contribution of this work can be listed as follows:

- demonstrating a factor of two speedup in the expensive SpMV kernel of Lanczos by switching from OpenACC to CUDA,

- benchmarking the cost of all four MPI collectives used in the distributed SpMV algorithm: Allgatherv, Bcast, Reduce and Reduce_scatter,
- showing *MPI_Reduce* and *MPI_Reduce_scatter* to be the performance bottleneck of Lanczos on multi-GPU experiments due to local reductions needed by these two calls being performed on the host side,
- proposing a Hybrid/CUDA scheme by employing non-blocking MPI point-to-point (P2P) communication for Reduce and Reduce_scatter, and MPI collectives for Allgatherv and Bcast to overcome the bottleneck,
- proposing a NCCL/CUDA scheme by replacing all four MPI calls with NVIDIA Collective Communications Library (NCCL) collectives to overcome the same bottleneck,
- applying optimizations on the NCCL/CUDA scheme to truly overlap communication with computation and to utilize the memory bandwidth to a better extend,
- evaluating the performance of Hybrid/CUDA and optimized NCCL/CUDA schemes for large matrices with up to a trillion nonzeros on large-scale experiments using up to 1128 NVIDIA A100 GPUs,
- achieving significant performance improvements in overall Lanczos solver time with Hybrid/CUDA (up to 2.9×) and NCCL/CUDA (up to 4.9×) schemes over MPI/OpenACC,
- analyzing the strong-scaling efficiency of the default and proposed schemes to highlight the impact of increasing communication volume on the overall performance,
- showing the performance of the proposed schemes on LOBPCG and investigating the lack of speedup observed.

This chapter is organized as follows. In Section 5.1, we explain in detail the existing distributed GPU-parallel SpMV algorithm used in MFDn. In Section 5.2, we describe our

proposed schemes. Next, in Section 5.3, we share the performance results of the default as well as the proposed schemes on Lanczos. We then give the performance of these schemes on LOBPCG in Section 5.4. We conclude our work by giving final remarks in Section 5.5.

5.1 MFDn's Proprietary Distributed SpMV Algorithm

SpMV is by far the most time-consuming part of the Lanczos algorithm. An efficient implementation of this step is crucial in order to obtain a high-performance Lanczos code. This is particularly true on distributed-memory where the communication part can easily become bottleneck and hinder performance if not addressed properly.

In this section, we first describe the custom data distribution of the sparse matrix, referred to as H, as well as the input and output vectors, referred to as W and U. We then outline the steps of the proprietary SpMV algorithm employed in MFDn to accommodate this custom data distribution.

5.1.1 Data Distribution

Since the size of the sparse matrix H in a nuclear CI calculation, both in terms of dimension and number of nonzeros, can be extremely large, H is partitioned and distributed across multiple processes [54]. Additionally, in MFDn, only half of the symmetric matrix H is stored, utilizing a specialized data distribution scheme, as described below.

We first divide the rows and columns of H into $n_d \times n_d$ matrices with 2D partitioning. Since the matrix is symmetric, we then only map $n_d \times (n_d + 1)/2$ of these submatrices to $n \times (n_d + 1)/2$ processes where each process is assigned to a submatrix. When choosing $n_d \times (n_d + 1)/2$ out of $n_d \times n_d$ submatrices, ensuring that we select exactly $(n_d + 1)/2$ submatrices on each row and column group is vital. This is because the processes are later organized into (sub)communicators based on the row and column indices of their respective submatrices. Well-balanced row and column (sub)communicators are key to achieving a meaningful performance for the distributed SpMV algorithm we will describe.

In Figure 5.1, we show an example of this partition and selection process. Here, H is partitioned into 5×5 matrices. We also highlight the selected submatrices, each of which

$H_{1,1}$			$H_{1,4}$	$H_{1,5}$
$H_{2,1}$	$H_{2,2}$			$H_{2,5}$
$H_{3,1}$	$H_{3,2}$	$H_{3,3}$		
	$H_{4,2}$	$H_{4,3}$	$H_{4,4}$	
		$H_{5,3}$	$H_{5,4}$	$H_{5,5}$

Figure 5.1 The partition of symmetric sparse matrix H into 5×5 submatrices. We store 15 of them while achieving load balance.

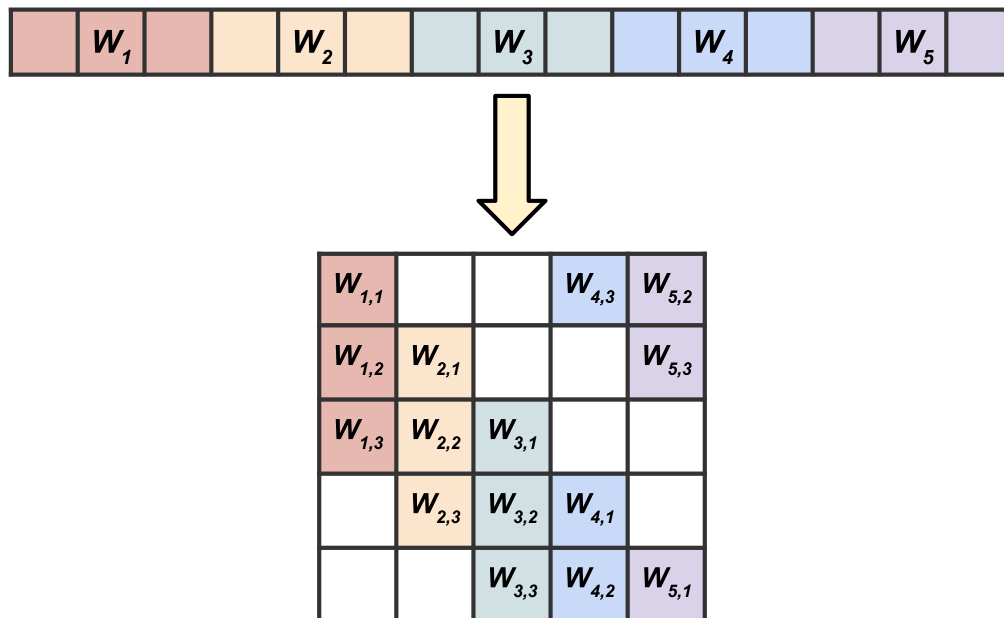


Figure 5.2 The partition of input vector W into 5 sub-vectors on each column. We partition each sub-vector further into 3 segments.

is assigned to its own process. Notice that each row and column holds 3 submatrices. This means there will be 5 row communicators and 5 column communicators, each containing 3 MPI processes.

We also partition the input vector W and the output vector U among $n_d \times (n_d + 1)/2$ processes in two stages. First, we partition W and U into n_d sub-vectors and distribute among column groups. Then, we further partition each sub-vector into $(n_d + 1)/2$ segments and distribute among the processes within each column group. By this way, each process is assigned to $1/(n_d \times (n_d + 1)/2)$ of W and U, providing load-balance. We show the partition of W in Figure 5.2 for the same example given in Figure 5.1.

5.1.2 Distributed SpMV

A specialized SpMV multiplication procedure has been developed to fit this unique data distribution scheme when multiplying H with W. Here, we refer to the jth block of sub-vectors of W as W_j. In MFDn, the distributed-memory parallel multiplication of H and W is performed as follows:

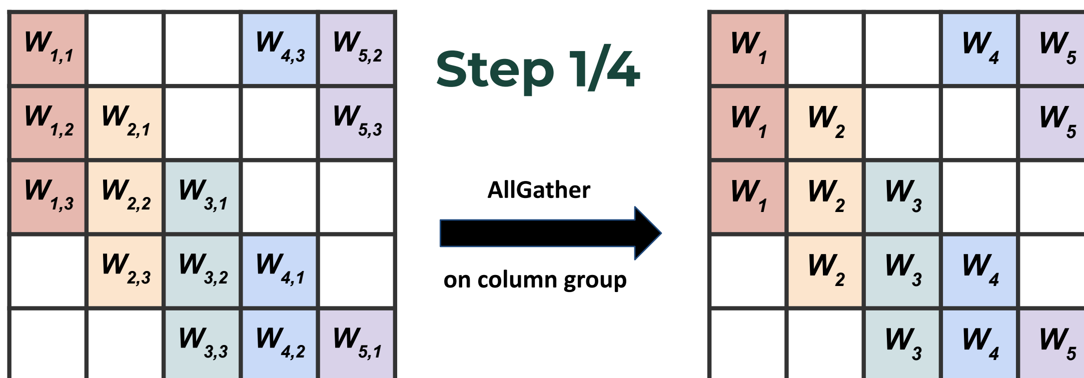


Figure 5.3 First step of the SpMV algorithm in MFDn where we gather the distributed input segments onto each process.

1. We first gather the distributed segments of the sub-vector W_j onto each process within the jth column communicator by using an *MPI_Allgatherv* call. This step is shown in Figure 5.3.

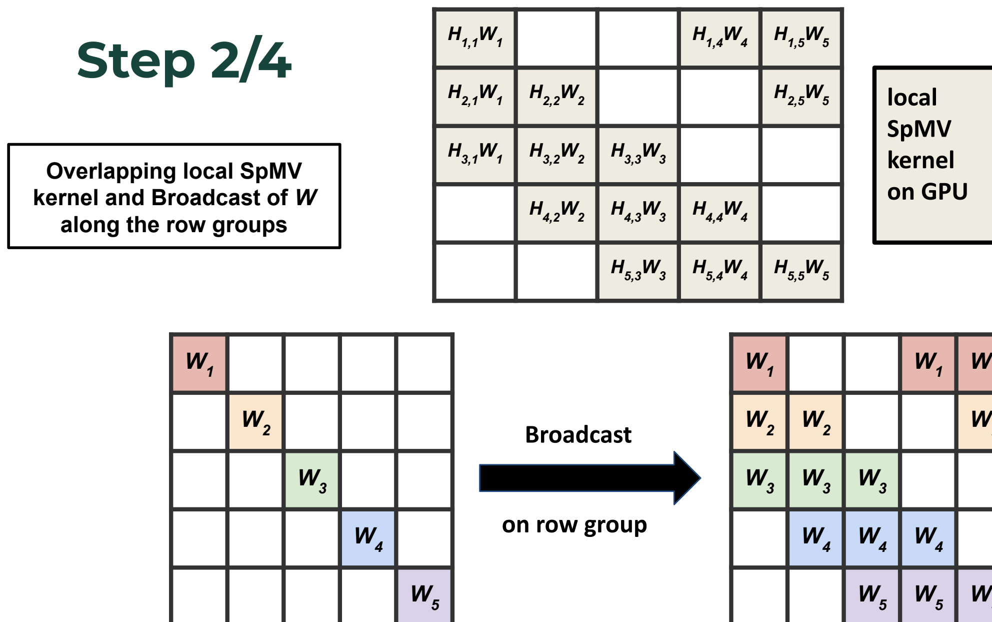


Figure 5.4 Second step of the proprietary SpMV algorithm employed in MFDn where we overlap the broadcast of the gathered input segments with local SpMV.

2. The ith diagonal process then broadcasts the gathered sub-vector W_j, where $i = j$ for the diagonal processes, across the ith row communicator. This is done in preparation for the distributed SpMV-transpose computations of the next step. We overlap this MPI_Bcast call with the local SpMV using the local sub-matrix $H_{i,j}$, which is $U_i = H_{i,j}W_j$, by using the gathered input sub-vector from the previous step. This step is shown in Figure 5.4.

3. By the end of the previous step, each process within the ith row communicator holds the partial result of the output sub-vector U_i. Now, we reduce the output sub-vectors U_i along each row communicator onto the ith diagonal process. We overlap this MPI_Reduce call with the local SpMV-transpose on the sub-vector W_i, which is $U_j = H_{i,j}^T W_i$, by using the broadcast input sub-vector from the previous step. This step is shown in Figure 5.5.

4. After the local SpMV-transpose, on the diagonal processes, we add the reduced output

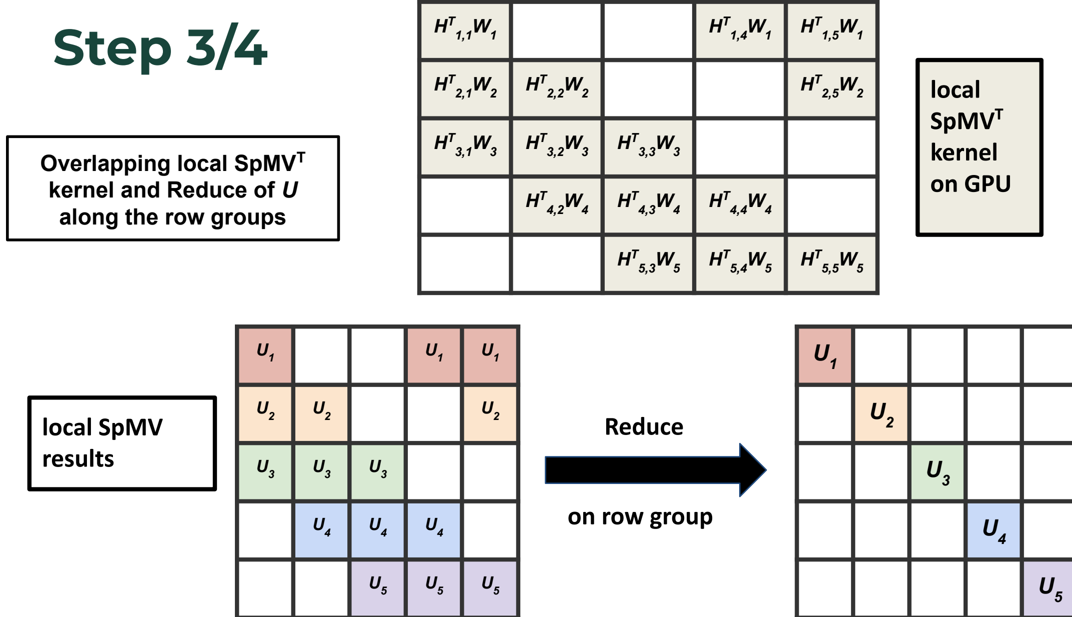


Figure 5.5 Third step of the proprietary SpMV algorithm employed in MFDn where we overlap the reduce of the partial SpMV outputs with local SpMV-transpose.

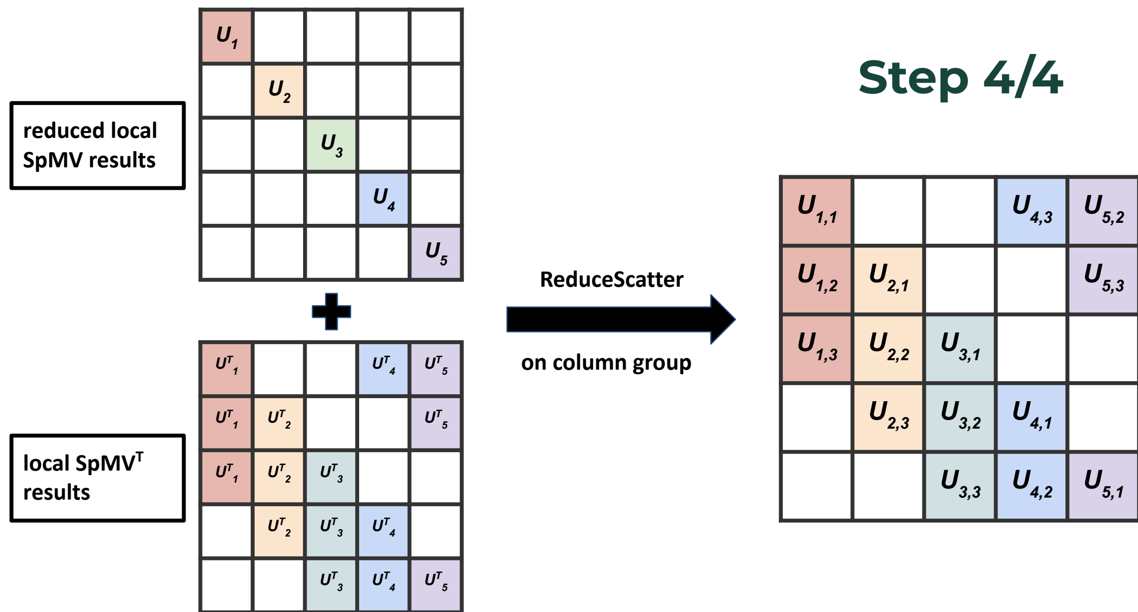


Figure 5.6 Fourth and last step of the proprietary SpMV algorithm employed in MFDn where we reduce and scatter the partial SpMV-transpose outputs after adding the reduced SpMV output on diagonal processes.

sub-vector U_i to the local output sub-vector U_j. Finally, we reduce and scatter the sub-vectors U_j into $(n_d+1)/2$ segments within the jth column communicator by using *MPI_Reduce_scatter*. At the end of this step, each process gets the corresponding segment of the output vector as intended, which concludes our SpMV algorithm. This step is shown in Figure 5.6.

5.1.3 Task-to-Processor Mapping

At large scales, communication overhead can limit the scalability of our eigensolvers. As such, investigating methods to reduce such overhead is of value. Topology-aware mapping of computational tasks to physical processors is one of these methods in particular since such mapping on large-scale clusters can significantly improve efficiency.

When mapping $n_d \times (n_d+1)/2$ submatrices of H and $n_d \times (n_d+1)/2$ segments of W and U to $n_d \times (n_d+1)/2$ MPI processes, we use a column-major mapping scheme developed in [3]. Figure 5.7 shows this column-major mapping for the same example where $n_d = 5$.

1			12	14
2	4			15
3	5	7		
	6	8	10	
		9	11	13

Figure 5.7 Using a column-major order for process ordering. Tasks mapped to the same column (row) of the grid belong to the same column (row) communication groups.

The reason for that is, we can partially or entirely hide the cost of *MPI_Bcast* and *MPI_Reduce* by overlapping them with local SpMV and SpMV-transpose. Both collectives are performed along row communicators. However, there is no compute task to overlap

MPI_Allgatherv and *MPI_Reduce_scatter* with. Therefore, performing these two collectives on column communicators as quickly as possible would be efficient in terms of overall performance [3].

MPI library itself considers the network topology when assigning ranks to processes. For example, processes within a single node will be given consecutive ranks by default. Therefore, ordering processes in column-major order will ensure processes in a column group and their corresponding tasks are assigned to physically nearby processors (or GPUs).

Although topology-aware mapping is shown to be an NP-complete problem [27], this heuristic suggested by [3] is proven to be quite efficient for the performance of the given SpMV algorithm. Please note that [3] shows the merit of this heuristic for the MPI/OpenMP implementation of MFDn on a distributed CPU cluster. However, [40] uses the same heuristic for the GPU-parallel MPI/OpenACC scheme as well due to the same underlying principles. As such, we will continue using this heuristic in our work as well.

5.2 Suggested Improvements and Proposed Schemes

MFDn uses the MPI/OpenACC scheme to implement the distributed SpMV algorithm outlined here. That is, it currently uses OpenACC as the GPU programming model to perform local SpMV and SpMV-transpose, and MPI collectives to perform *Allgatherv*, *Bcast*, *Reduce* and *Reduce_scatter* calls.

Our objective with this work is to improve the Lanczos performance by optimizing the distributed SpMV algorithm. We limit our improvements to SpMV mainly because we spend $75-90\%$ of the solver time on this algorithm as we will show later. Also, the orthogonalization kernel, where we spend the rest of the solver time, does not offer any room for improvement since it consists of load-balanced dense linear algebra operations.

Before defining our proposed schemes, we first list the improvements we suggest.

5.2.1 Improving Local SpMV with CUDA

OpenACC was the first choice to port the distributed solver code to GPUs within MFDn. Although OpenACC is easy to program with, this ease of programming comes at the expense

of efficiency. That is, one can easily implement a GPU-parallel code using OpenACC directives but this does not usually produce the most optimal performance. Using a low-level GPU programming language such as CUDA can improve the GPU kernel efficiency. This is mostly because we have more control over how the parallel loops, for example, are mapped to the software or hardware resources at the GPU stack.

Also, the main advantage of OpenACC is the portability. One can easily compile the code for different architectures to have a production-ready version quickly. Offering portability intrinsically means losing performance to a certain degree. CUDA is exclusively used for NVIDIA GPUs, meaning it offers no portability. This lack of portability allows the CUDA programs to exploit the exclusive features offered for NVIDIA GPUs.

Since we were running our experiments on NVIDIA GPUs at Perlmutter, we propose to improve the compute aspect of our proprietary SpMV algorithm by switching from OpenACC to CUDA in order to gain performance.

5.2.2 Improving Communication with P2P

As we will show later, *MPI_Reduce* and *MPI_Reduce_scatter* calls inside the SpMV algorithm become the bottleneck of our Lanczos solver on large-scale. This is due to the local arithmetic needed by these two calls being performed on the host side.

To overcome this problem, we propose using non-blocking point-to-point (P2P) communication. All MPI collectives can be implemented with simple *MPI_Send* and *MPI_Recv* calls between two processes. MPI also provides a non-blocking send and receive mechanism with *MPI_Isend* and *MPI_Irecv*. As such, we propose implementing these two MPI collectives using *MPI_Isend* on the send side with *MPI_Recv* on the receive side.

This mechanism adds asynchrony since it allows the send side to start the P2P call as early as possible and continue the work. It also allows the receive side to stall the blocking P2P call as much as possible while doing useful work. Moreover, the expensive additions performed by MPI with Reduce and Reduce_scatter collectives can be done locally on the GPU side when these collectives are converted to P2P calls.

There are two ways to implement the local reductions on the receive side. First, we can allocate a receive buffer large enough to store all the incoming data and perform reduction once when all *MPI_Recv* calls are complete. Secondly, we can allocate a receive buffer only large enough to store the incoming data from a single process and perform reduction after each *MPI_Recv*.

We choose to implement the second option due to its space efficiency. Also, we couple this implementation with *MPI_ANY_SOURCE* on the receive side so we do not impose any order among the sent data, which may help with time efficiency.

5.2.3 Improving Communication with NCCL

The NVIDIA Collective Communications Library (NCCL, pronounced "Nickel") is a library offering topology-aware inter-GPU communication primitives that can be seamlessly integrated into applications [15].

NCCL provides both point-to-point send/receive primitives and collective communication. While it is not a comprehensive parallel programming framework, it is a library specifically designed to accelerate inter-GPU communication.

NCCL has been widely adopted in Deep Learning Frameworks, where the AllReduce collective is extensively used for training neural networks. The multi-GPU and multi-node communication capabilities provided by NCCL enable efficient scaling of neural network training.

Regarding the use of NCCL within MFDn, the collective communication primitives supported by NCCL include all four collectives needed by our SpMV algorithm with minor differences and/or adjustments:

- There is no Allgatherv collective provided by NCCL. Instead, NCCL provides ncclAllGather where the aggregation is performed on equal-sized vectors. This is only a minor issue because the scattered input vectors that are accumulated by *MPI_Allgatherv* within MFDn have similar sizes. In fact, their size varies by less than 1%. As such, padding the participating vectors with zeros to accommodate for the lack of Allgatherv

in NCCL only introduces a small overhead both in memory and compute time.

- Although NCCL provides a collective that implements the same functionality as MPI's Reduce_scatter, which is ncclReduceScatter, this collective expects equal-sized vectors from all participating processes as in ncclAllGather. Likewise, this issue is overcome with only minor overhead due to the same reason.

5.2.4 Proposed Schemes

Based on these potential improvements, we propose three new schemes.

MPI/CUDA: This scheme uses MPI collectives as is but replaces the OpenACC kernels for local SpMV and SpMV-transpose with corresponding CUDA kernels.

Hybrid/CUDA: This scheme uses CUDA as well. It also replaces *MPI_Reduce* and *MPI_Reduce_scatter* collectives with non-blocking MPI P2P calls and local GPU reduction kernels.

NCCL/CUDA: This scheme uses CUDA and replaces MPI collectives with NCCL collectives.

We summarize the proposed schemes in Table 5.1 along with the default MPI/OpenACC scheme.

	MPI/OpenACC	MPI/CUDA	Hybrid/CUDA	NCCL/CUDA
Local SpMV	OpenACC	CUDA	CUDA	CUDA
Local SpMV^T	OpenACC	CUDA	CUDA	CUDA
Allgatherv	MPI Coll	MPI Coll	MPI Coll	NCCL Coll
Bcast	MPI Coll	MPI Coll	MPI Coll	NCCL Coll
Reduce	MPI Coll	MPI Coll	MPI P2P	NCCL Coll
Reduce_scatter	MPI Coll	MPI Coll	MPI P2P	NCCL Coll

Table 5.1 Summary of the default MPI/OpenACC as well as the proposed MPI/CUDA, Hybrid/CUDA and NCCL/CUDA schemes.

5.3 Lanczos Experiments

In this section, we share the experiment results of all four schemes shown in Table 5.1 for the Lanczos eigensolver.

5.3.1 Experimental Setup

We report the performance of Lanczos in MFDn when it is run for 100 iterations on the Perlmutter GPU system at the National Energy Research Scientific Computing (NERSC) Center. Each Perlmutter GPU compute node consists of a single AMD EPYC 7763 processor with 64 cores per processor and 4 NVIDIA A100 GPUs. We give the detailed hardware and system specifications of Perlmutter GPU system in Table 5.2. Also, we report the versions of the relevant software used in this work in Table 5.3. Finally, we plot the node architecture of Perlmutter GPU machines in Figure 5.8.

CPUs per node	1 x AMD EPYC 7763
GPUs per node	4 x NVIDIA A100
NICs per node	4 x HPE Slingshot 11 NICs
GPU-GPU interconnect	NVLink 3.0
CPU-GPU interconnect	PCIe 4.0
CPU-NIC interconnect	PCIe 4.0

Table 5.2 System specifications of Perlmutter GPU.

NVIDIA HPC SDK	23.9
CUDA	12.2
MPI	Cray-MPICH (8.1.28)
NCCL	NCCL 2.18.3

Table 5.3 Software versions used in the performance experiments on Perlmutter.

We use four test problems that correspond to realistic Hamiltonian matrices of different nuclei (with up to 12 nucleons) represented in different configuration interaction spaces. The dimensions of these matrices as well as the number of nonzero matrix elements (nnz) in half of each of these symmetric matrices are listed in Table 5.4. We see in this table that the dimension of these test problems ranges from 1.2×10^7 to 9.8×10^8 whereas their nnz varies between 5.4×10^9 and 1.3×10^{12}.

The smallest problem, referred to as XSmall, can fit within one NVIDIA A100 80GB GPU on a single node. As the problem size becomes larger and the memory footprint of the Lanczos algorithm exceeds the available high-bandwidth memory (HBM2e), we need to

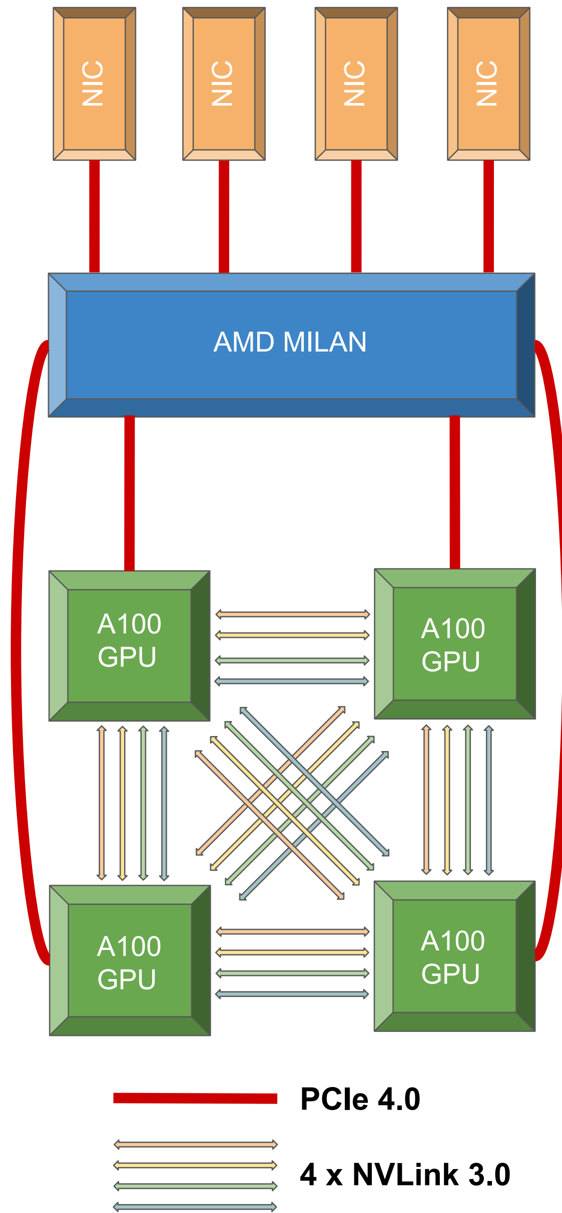


Figure 5.8 Node architecture of Perlmutter GPU.

distribute the Hamiltonian matrix and the input/output vector across multiple GPUs. As such, the Lanczos computations for Small, Medium and Large test cases have to be run on at least 28, 91 and 276 A100 80GB GPUs, respectively.

Test case	XSmall	Small	Medium	Large
System	^{10}B	^{10}B	^{12}C	^{9}Li
$N_{\max}$	6	8	8	11
Matrix dimension ($\times 10^6$)	12.0	165	575	978
# of nonzero elemens (nnz) ($\times 10^9$)	5.47	129	475	1305
# of A100 80GB GPUs needed	1	28	91	276

Table 5.4 The dimensions of sparse matrices used in the performance test and number of nonzero matrix elements in each matrix.

5.3.2 MPI/OpenACC Results

In Table 5.5, we show the overall solver time of Lanczos for 100 iterations with the MPI/OpenACC scheme. For each test problem, we start with the minimum GPU count needed and increase this number to assess this scheme's strong-scaling performance. For example, on Medium, we first use 91 GPUs on which 100 solver iterations take 44.5s and go up to 496 GPUs where the solver time drops down to 18.5s.

Test Case	# of Nodes	# of GPUs	Solver Time
Small	7	28	38.4
	12	45	26.1
	23	91	15.5
	48	190	11.8
	69	276	10.4
Medium	23	91	44.5
	48	190	30.4
	69	276	23.5
	95	378	21.5
	124	496	18.5
Large	69	276	43.0
	95	378	35.9
	124	496	30.0
	195	780	27.3
	282	1128	25.6

Table 5.5 Solver time in seconds for 100 Lanczos iterations with MPI/OpenACC.

We will discuss the strong-scaling efficiency of MPI/OpenACC on these matrices later in Section 5.3.7 along with the other schemes. For now, we turn our focus to the significance of the distributed SpMV algorithm in Lanczos.

In Table 5.6, we share the time spent in the SpMV algorithm for a subset of the experiments. We also compute the percentage of time spent in this algorithm over the entire Lanczos solver time. We notice that the SpMV account for $75 - 90\%$ of the solver time. Regardless of the scale of the experiment or the test problem used, SpMV is the most time-consuming part of Lanczos. This is the main reason why we invest our efforts into improving the SpMV time with our proposed schemes.

The performance of MPI/OpenACC shown in Table 5.5 will be our baseline to compare the new schemes against.

Test Case	# of Nodes	# of GPUs	SpMV Time	Solver Time	Percentage
Small	12	45	23.5	26.1	90.1
	23	91	13.2	15.5	85.2
	48	190	8.8	11.8	75.0
Medium	23	91	40.4	44.5	90.9
	48	190	27.9	30.4	92.0
	95	378	17.6	21.5	81.9

Table 5.6 Percentage of time spent in SpMV for 100 Lanczos iterations with MPI/OpenACC.

5.3.3 MPI/CUDA Results

We give the performance results of the MPI/CUDA scheme in Table 5.7 for the same set of experiments. In this table, we also show the speedup achieved over MPI/OpenACC.

We see that by changing the local SpMV and SpMV transpose kernels originally implemented using OpenACC directives to hand-tuned CUDA implementations, we achieve up to 2.00× speedup. However, we observe the highest speedup always on the lowest GPU count for each problem. For the Small problem, for example, the 2.00× speedup obtained on 45 GPUs gradually diminishes to an only 3% improvement on 276 GPUs. We essentially see no advantage of using CUDA over OpenACC on the highest GPU count in any test cases.

Test Case	# of Nodes	# of GPUs	Solver Time	Speedup
Small	7	28	19.2	2.00
	12	45	15.2	1.71
	23	91	11.5	1.34
	48	190	10.2	1.15
	69	276	10.1	1.03
Medium	23	91	32.8	1.35
	48	190	25.5	1.19
	69	276	22.0	1.07
	95	378	20.6	1.04
	124	496	18.2	1.02
Large	69	276	34.0	1.27
	95	378	37.0	1.17
	124	496	26.8	1.12
	195	780	26.6	1.03
	282	1128	25.5	1.00

Table 5.7 Solver time in seconds for 100 Lanczos iterations with MPI/CUDA and speedup achieved over MPI/OpenACC.

We suspected that this is due to the MPI collectives becoming the bottleneck as we increase the number of GPUs in our strong-scaling experiments. To verify that, we measured how much time we spend on each collective by each process throughout the 100 iterations of Lanczos. For an accurate measurement, we placed an $MPI_Barrier$ before these collectives. In Table 5.8, we share the maximum time spent on each collective by any process.

Test Case	# of Nodes	# of GPUs	Allgatherv	Bcast	Reduce	Reduce_scatter
Small	12	45	0.51	1.13	5.56	3.89
	23	91	0.36	1.07	3.85	3.36
	48	190	0.23	0.87	2.84	3.36
Medium	23	91	1.42	3.64	13.4	8.59
	48	190	0.76	3.07	9.61	6.94
	95	378	0.56	2.21	7.01	6.91

Table 5.8 Communication time spent on each collective in 100 Lanczos iterations with MPI collectives. Each collective is isolated with an $MPI_Barrier$. All processes take measurements separately and we report the maximum time among them for each collective.

These numbers suggest Reduce and Reduce_scatter collectives seem to be much more expensive than Allgatherv and Bcast. On the Small problem with 190 GPUs, for instance, we spend 6.2s total on Reduce and Reduce_scatter while Allgatherv and Bcast sum up to

1.1 seconds. Considering that MPI/CUDA scheme spends 10.2s for this problem and GPU count (see Table 5.7), spending 6.2s on Reduce and Reduce_scatter seems to be an issue we need to address.

Moreover, although we overlap *MPI_Bcast* with local SpMV and *MPI_Reduce* with local SpMV-transpose, we believe having an idea about the speedup we can achieve when MPI collectives take this much time can help when interpreting the MPI/CUDA performance. For that reason, we calculate in Table 5.9 the total time spent on MPI collectives as percentage of the overall solver time. We then use this percentage to arrive at the maximum theoretical speedup that can be achieved by optimizing the local GPU kernels.

Test Case	# of Nodes	# of GPUs	Comm. Time	% of Solver	Max. Speedup
Small	12	45	11.1	42.5	2.35
	23	91	8.64	55.7	1.79
	48	190	7.30	61.8	1.61
Medium	23	91	27.1	60.8	1.64
	48	190	20.4	67.1	1.49
	95	378	16.7	77.6	1.28

Table 5.9 Total communication time spent on collectives based on Table 5.8, percentage of communication time to solver time given in Table 5.5, and maximum theoretical speedup possible by keeping the communication cost fixed.

These numbers suggest that there is only so much speedup we can get by simply improving the local computations. For example, we can never see more than 1.28× speedup with MPI/CUDA for the Medium problem on 378 GPUs when the MPI collectives are this costly. This realization motivates us to understand the underlying issue with *MPI_Reduce* and *MPI_Reduce_scatter* calls and address it to achieve further improvements.

5.3.4 The Issue with Reduce and Reduce_scatter

Realizing that *MPI_Reduce* and *MPI_Reduce_scatter* take considerably longer than the other two collectives got us thinking about the compute part of these collectives, *i.e.,* local reductions. We thought these slow collectives might be performed on the host side despite using CUDA-aware MPI. To investigate this further, we did Nsight Systems profiling. Here is what our profiling analysis reveals:

- Regarding *MPI_Reduce*, we observe a single device-to-host data movement (*DtoH memcpy*) in the earlier stages of this call.
- For the same call, we observe a single host-to-device data movement (*HtoD memcpy*) in the later stages only on the root process.
- Regarding *MPI_Reduce_scatter*, we observe multiple device-to-host and host-to-device memory copy operations on all processes throughout this call.
- To give an example, when there are three processes participating in this call, we observe four device-to-host and two host-to-device copies. For seven processes, these numbers go up to nine device-to-host and six host-to-device copies.

In Figure 5.9, we give a snapshot of our Nsight Systems profiling where the bottom part shows the blocking nature of *MPI_Reduce* and *MPI_Reduce_scatter* calls on the CPU side. The top part, where we can see the activity on the CUDA hardware, reveals the copy operations between host and device.

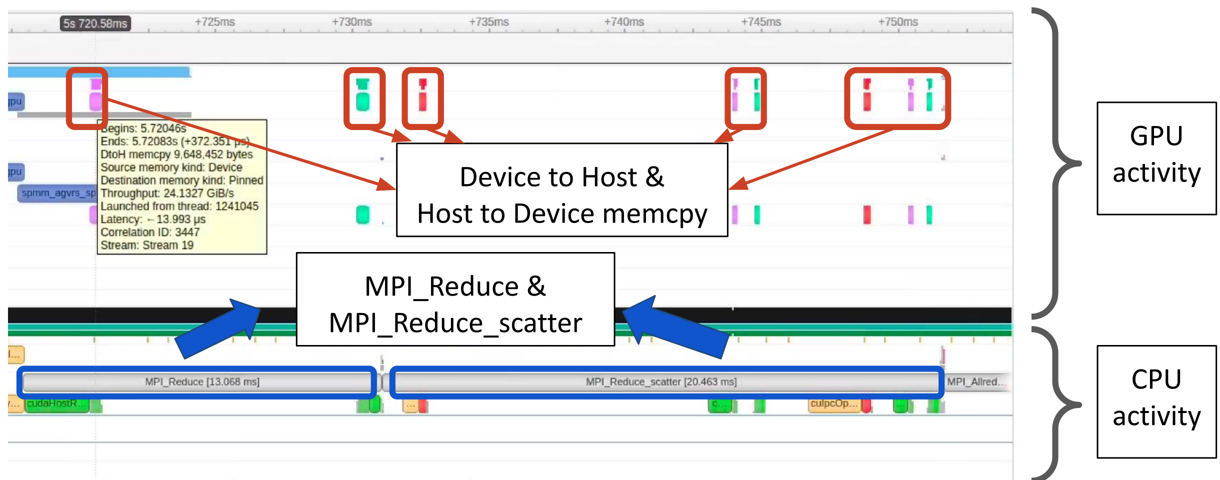


Figure 5.9 NSight Systems profiling showing device to host and host to device copy operations performed during *MPI_Reduce* and *MPI_Reduce_scatter* calls. The Lanczos is run for XSmall test case on 15 GPUs in this experiment.

Moreover, the size of these copy operations occurring concurrently with these collectives matches the size of the buffers used in these calls. These observations made it clear to us

that reductions are indeed happening on the host side. That is why these two collectives were notoriously slower than the other two as given in Table 5.8.

Please note that we use a CUDA-aware MPI implementation provided by Cray-MPICH (8.1.28), which is the default MPI library on Perlmutter. After realizing that reductions takes place on the CPU with this implementation, we wanted to test if it was a single-threaded or multi-threaded reduction operation.

In Table 5.10, we share the time *MPI_Reduce* and *MPI_Reduce_scatter* take in 100 Lanczos iterations for varying number of CPU threads. We see that using one, two, eight or 32 threads (as first allocated by Slurm's *–cpus-per-task* flag and then set by *export OMP_NUM_THREADS*) makes no difference in the performance.

Test Case	# of Nodes	# of GPUs	# of CPU Threads	Reduce	Reduce_scatter
XSmall	4	15	1	0.84	0.54
			2	0.82	0.60
			8	0.84	0.54
			32	0.82	0.56

Table 5.10 Reduce and Reduce_scatter time spent in 100 Lanczos iterations with MPI/OpenACC for varying number of CPU threads used.

Given that the reductions needed by these two collectives are very likely to use a single-threaded CPU implementation, we explored ways to improve the overall solver time by addressing this bottleneck through Hybrid/CUDA and NCCL/CUDA schemes.

5.3.5 Hybrid/CUDA Results

In Table 5.11, we give the speedup achieved on Reduce and Reduce_scatter operations by changing the MPI collectives with a set of non-blocking point-to-point MPI calls and local reductions on the GPU (see Section 5.2.2 for more details). The table shows that we can improve the Reduce time up to 2.3× and the Reduce_scatter time up to 5.9× with this technique. However, we notice a decline in the speedup numbers for Reduce as we go from Small to Medium and/or increase the number of GPUs. In fact, we observe no speedup at all for the Medium problem on 378 GPUs.

Test Case	# of Nodes	# of GPUs	Reduce	Speedup	Reduce_scatter	Speedup
Small	12	45	2.44	2.28	0.74	5.26
	23	91	2.39	1.61	0.75	4.48
	48	190	1.99	1.43	0.57	5.89
Medium	23	91	8.18	1.64	2.61	3.29
	48	190	8.80	1.09	2.05	3.39
	95	378	7.00	1.00	1.61	4.29

Table 5.11 Reduce and Reduce_scatter time spent in seconds for 100 Lanczos iterations with MPI point-to-point communication and speedup achieved over MPI collective communication based on Table 5.8. Both calls are isolated with an MPI_Barrier. All processes take measurements separately and we report the maximum time among them for each call.

Recall that the Reduce call is performed along a row communicator as overlapped with the local SpMV-transpose. The processes on row communicators are physically farther away from each other than those on column communicators, on which the Reduce_scatter call takes place. Due to the non-scaling nature of point-to-point communication, which is exacerbated by the distant communicating processes, our point-to-point Reduce implementation loses its advantage against the slow *MPI_Reduce* collective at scale.

Now we give the performance results of the Hybrid/CUDA scheme along with the speedup achieved over MPI/OpenACC in Table 5.12. In this table, we consistently observe around 2.8× speedup for the Small problem. For Medium, we start a with 2.3× speedup on 91 GPUs and end up with 1.8× speedup on 496 GPUs. For the Large problem, we can still see a 62% improvement on 1128 GPUs.

These numbers indicate the significance of performing local arithmetic needed by Reduce and Reduce_scatter on the device side. By simply getting rid of these two MPI collectives in the distributed SpMV algorithm, which perform a CPU-side reduction, we achieve between 1.6× and 2.9× speedup in the overall solver time.

5.3.6 NCCL/CUDA Results

In Table 5.13, we give the speedup achieved on Reduce and Reduce_scatter operations by using NCCL instead of MPI collectives. The Reduce_scatter time given here includes the process of zero-padding the send buffer before calling ncclReduceScatter (see Section 5.2.3

Test Case	# of Nodes	# of GPUs	Solver Time	Speedup
Small	7	28	13.8	2.77
	12	45	9.45	2.76
	23	91	5.66	2.74
	48	190	4.12	2.86
	69	276	3.60	2.89
Medium	23	91	19.7	2.25
	48	190	14.4	2.11
	69	276	12.4	1.90
	95	378	11.4	1.88
	124	496	10.6	1.75
Large	69	276	22.5	1.91
	95	378	20.4	1.76
	124	496	19.1	1.57
	195	780	17.1	1.59
	282	1128	15.7	1.62

Table 5.12 Solver time in seconds for 100 Lanczos iterations with Hybrid/CUDA and speedup achieved over MPI/OpenACC.

for more details).

This table shows that on Small and Medium test cases, we can improve the Reduce time up to 11.1× and the Reduce_scatter time up to 33.6× with this change. We again attribute the higher speedup numbers achieved on the Reduce_scatter call to the fact that processes participating in the Reduce call are more distant to each other. When the communicating processes are farther, the communication part of the Reduce call takes a bigger portion of this collectively. Consequently, we lose some of the gains obtained by using NCCL's GPU-side reduction over MPI's CPU-side reduction in the compute part.

Test Case	# of Nodes	# of GPUs	Reduce	Speedup	Reduce_scatter	Speedup
Small	12	45	0.50	11.1	0.34	11.4
	23	91	0.39	9.87	0.24	14.0
	48	190	0.30	9.47	0.10	33.6
Medium	23	91	2.75	4.89	1.90	4.52
	48	190	1.96	4.90	0.32	21.6
	95	378	1.62	4.33	0.24	28.7

Table 5.13 Reduce and Reduce_scatter time in seconds for 100 Lanczos iterations with NCCL and speedup achieved over MPI based on Table 5.8. Both collectives are isolated with barriers. All processes take measurements separately and we report the maximum time among them for each collective.

In Table 5.14, we give the Lanczos solver time for four different versions of NCCL/CUDA to highlight our optimization efforts. The description of each version is as follows:

NCCL-v1: This is our initial version where we use as few synchronization call as possible. That is, we do not use any blocking MPI or NCCL call since both CUDA kernels and NCCL calls are assigned to asynchronous CUDA streams. We also only synchronize streams when there is a data dependency.

NCCL-v2: This version is similar to NCCL-v1, except before each NCCL call, we add an *MPI_Barrier* on the corresponding communicator and *cudaDeviceSynchronize*(). We notice in Table 5.14 that adding barriers improve the solver time in 14 out of 15 cases (three test cases, five runs each). For Medium on 496 GPUs, for example, adding such synchronization leads to a 60% improvement (11.5s vs 7.1s).

We originally implemented this version to time NCCL calls accurately. Achieving an improvement with such additional synchronizations and barriers seemed counter-intuitive. As such, we examined the Nsight profiling data to understand the underlying reasons. According to the profiling analysis, there are two reasons for the observed improvements.

First, we did not truly overlap communication and computation in NCCL-v1 to begin with. This is because NCCL calls are blocking either on the CPU or GPU side even when we configure the NCCL communicators to be non-blocking. Since communication and computation are already serialized, adding a CUDA synchronization call incurs no penalty.

Secondly, for reasons unknown to us, adding an *MPI_Barrier* before NCCL calls appears to accelerate the communication time in our experiments. That is, the overall cost of an *MPI_Barrier* followed by a NCCL call is lower than a stand-alone NCCL call. This seems to be the case even when there is no concurrent GPU kernel to the NCCL call. As such, adding an *MPI_Barrier* before NCCL calls turns out to be helpful in our case.

NCCL-v3: We began exploring ways to achieve overlap with NCCL after realizing it was not the case. We found out that we can truly overlap a GPU kernel with a NCCL call when we (i) launch the GPU kernel first and follow it by the NCCL call on the CPU

side, and (ii) assign a higher priority to the stream used by NCCL. This way, the NCCL call cannot possibly block the GPU kernel while the CUDA runtime can still allocate the resources needed by the NCCL call due to its higher priority.

Note that after seeing its merit on NCCL-v2, we use an $MPI_Barrier$ before each NCCL call in this version as well. We see in Table 5.14 that this version performs better than the previous version in 13 out of 15 cases. For example, we see a 33% improvement over NCCL-v2 for the Large problem when we test it on 496 GPUs (22.9s vs 17.2s).

NCCL-v4: Our last optimization effort with NCCL/CUDA is to adjust the number of thread blocks used by the NCCL communicators ourselves. NCCL by default selects a thread block count that can saturate the memory bandwidth whenever we create a NCCL communicator. NCCL then assigns that many thread blocks to the communicator each time it is used for a NCCL call throughout the Lanczos iterations. However, this default number may not always yield the best performance.

In this version, we start with the optimizations used by NCCL-v3. We then configure the NCCL communicators to try five different thread block counts: two, four, eight, 12 and 16. We run our Lanczos solver separately for each configuration and report the optimal solver time in Table 5.14. This version performs better than the previous version in 14 out of 15 cases.

We summarize the impact of our optimization efforts in Table 5.15 where we show the speedup achieved over our baseline with NCCL/CUDA for both NCCL-v1 and NCCL-v4. We see that NCCL-v1, our initial version, offers a speedup ranging from 1.6× to 2.2×. As we introduce new optimizations with each version, our final speedup reaches a point between 2.6× to 4.9×.

A question that naturally arises when using NCCL-v4, the final version, is what thread block count to use on each experiment. To answer this question, we show what thread block count yields the optimal performance in Table 5.16. In this table, we also give the percentage of slowdown occurring when we use a non-optimal thread block count.

Test Case	# of Nodes	# of GPUs	NCCL-v1	NCCL-v2	NCCL-v3	NCCL-v4
Small	7	28	18.4	16.2	13.8	11.6
	12	45	14.9	10.5	9.11	7.67
	23	91	9.38	5.66	4.71	4.00
	48	190	6.21	3.08	2.62	2.43
	69	276	4.86	2.38	2.37	2.39
Medium	23	91	29.5	25.9	20.1	14.6
	48	190	17.8	13.2	10.1	8.86
	69	276	13.3	10.4	10.1	7.86
	95	378	13.3	8.69	8.79	6.96
	124	496	11.5	7.12	8.33	6.84
Large	69	276	23.2	22.9	17.2	13.5
	95	378	22.0	19.4	15.4	12.1
	124	496	13.4	16.4	13.7	10.3
	195	780	15.2	11.5	10.2	9.39
	282	1128	13.6	10.1	10.0	9.93

Table 5.14 Solver time in seconds for 100 Lanczos iterations with NCCL/CUDA for all versions.

Test Case	# of Nodes	# of GPUs	Initial Speedup	Final Speedup
Small	7	28	2.08	3.31
	12	45	1.75	3.40
	23	91	1.65	3.88
	48	190	1.90	4.85
	69	276	2.14	4.34
Medium	23	91	1.51	3.04
	48	190	1.71	3.43
	69	276	1.78	3.00
	95	378	1.61	3.09
	124	496	1.60	2.71
Large	69	276	1.85	3.19
	95	378	1.63	2.98
	124	496	2.24	2.93
	195	780	1.79	2.90
	282	1128	1.88	2.57

Table 5.15 Speedup achieved with NCCL/CUDA for the initial (NCCL-v1) and the most optimized (NCCL-v4) versions over MPI/OpenACC for 100 Lanczos iterations.

Test Case	# of GPUs	2 Blocks	4 Blocks	8 Blocks	12 Blocks	16 Blocks
Small	28	4.8%	1.8%	0.5%	**Opt**	0.3%
	45	1.9%	**Opt**	0.5%	0.7%	1.0%
	91	2.0%	**Opt**	1.0%	1.3%	1.8%
	190	4.6%	**Opt**	1.9%	1.6%	2.2%
	276	16.1%	0.5%	**Opt**	3.3%	6.4%
Medium	91	33.4%	12.5%	3.0%	3.5%	**Opt**
	190	35.3%	14.2%	7.0%	3.6%	**Opt**
	276	30.3%	2.2%	0.3%	**Opt**	0.3%
	378	48.7%	20.0%	**Opt**	11.8%	5.9%
	496	23.0%	13.8%	3.6%	**Opt**	6.9%
Large	276	53.5%	15.9%	**Opt**	4.0%	0.1%
	378	39.0%	4.6%	0.1%	**Opt**	0.3%
	496	43.8%	15.0%	6.2%	**Opt**	7.3%
	780	29.6%	12.9%	3.4%	2.2%	**Opt**
	1128	13.5%	2.4%	2.1%	**Opt**	0.7%

Table 5.16 Percentage of slowdown observed over the optimal performance when the NCCL communicators are configured to use 2, 4, 8, 12 and 16 thread blocks for NCCL-v4.

These numbers suggest that configuring the communicators to use two thread blocks yields the worst performance in general. It is worth noting that this configuration can be up to 54% slower on the Large matrix. Using four thread blocks, on the other hand, seems to work well on the Small test case although it can be up to 20% and 16% slower than the best option on Medium and Large, respectively.

Regarding the remaining three options, there is probably a trade-off when we go from using eight to 16 thread blocks for NCCL calls. Using eight blocks might correspond to a slightly slower NCCL communication but it leaves more streaming multiprocessors (SMs) to complete the concurrent GPU kernels slightly faster. This might be the reason why the overall solver time does not vary much when we go from eight to 16.

We find eight, 12 and 16 thread block configurations to be commendable for our experiments since all three configurations are at most 12% slower than the optimal configuration in any case. However, we note that the optimal thread block count to configure a NCCL communicator with depends on the network architecture and topology, problem size and communication pattern.

5.3.7 Discussion

In Figure 5.10, 5.11 and 5.12, we show the strong-scaling plot of MPI/OpenACC, Hybrid/CUDA and NCCL/CUDA, respectively. In these figure, the y-axis denotes the solver time and the x-axis indicates the GPU count. Along with the observed solver time for each experiment, we also plot dashed lines to indicate the ideal scaling, which is what we would observe if the speedup achieved matched the scale-up in GPUs used.

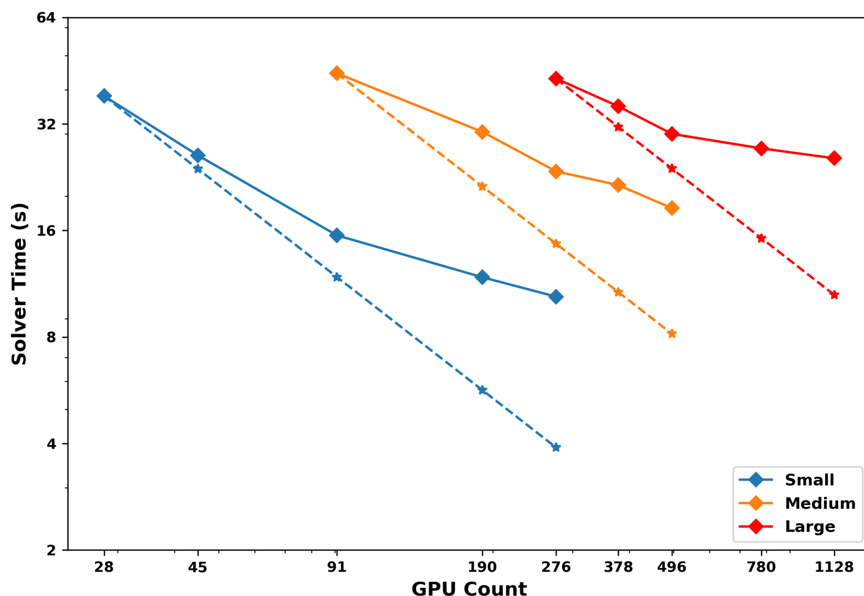


Figure 5.10 Strong-scaling plot of MPI/OpenACC for 100 Lanczos iterations. Solid lines correspond to solver time with respect to the GPU count for each problem whereas dashed lines show ideal scaling.

These plots show that we slowly deviate from the ideal scaling with each scheme. To support this, we show in Table 5.17 the strong-scaling efficiency attained with each scheme for each test case where the percentage numbers shown in this table correspond to the ratio between the speedup achieved and the ideal speedup.

We see that our efficiency never goes below 33% but we cannot ever reach 50% either, which is far from ideal. We know that going from MPI/OpenACC to MPI/CUDA yields an only marginal return whereas changing both the communication and the compute aspect

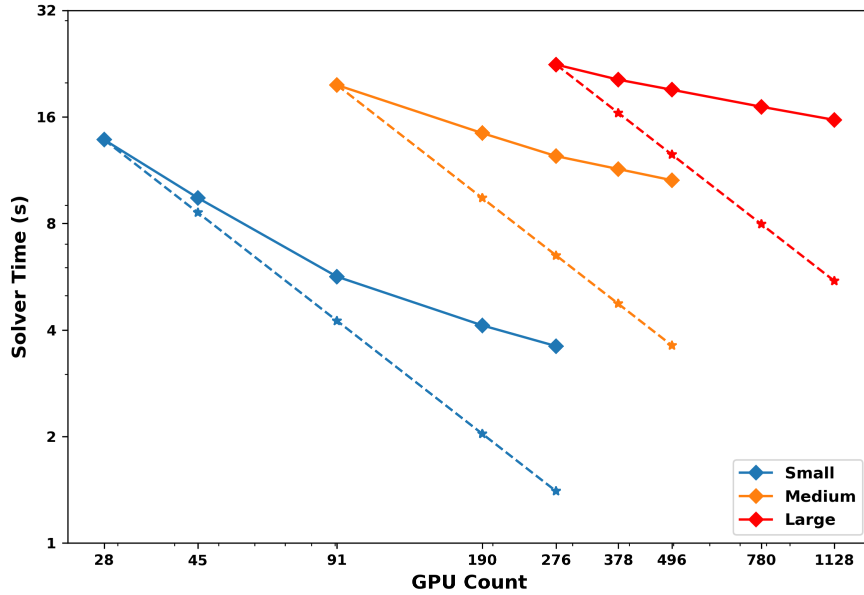


Figure 5.11 Strong-scaling plot of Hybrid/CUDA for 100 Lanczos iterations. Solid lines correspond to solver time with respect to the GPU count for each problem whereas dashed lines show ideal scaling.

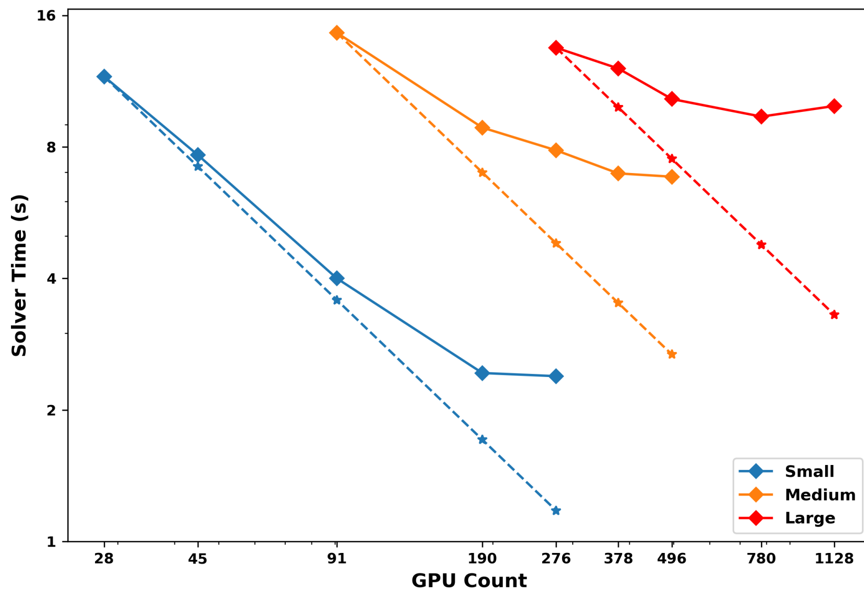


Figure 5.12 Strong-scaling plot of NCCL/CUDA (NCCL-v4) for 100 Lanczos iterations. Solid lines correspond to solver time with respect to the GPU count for each problem whereas dashed lines show ideal scaling.

produces significant speedups with Hybrid/CUDA and NCCL/CUDA. The strong-scaling efficiency numbers, on the other hand, suggest that the proposed schemes do not scale as well as the baseline. On the Large problem, for instance, MPI/OpenACC attains a 41% efficiency against the 35% and 33% of Hybrid/CUDA and NCCL/CUDA.

The higher efficiency numbers observed on Medium and Large with our baseline can be attributed to the compute aspect of the *MPI_Reduce* and *MPI_Reduce_scatter* collectives being their main bottleneck, which do not necessarily scale poorly. An increase in the number of processes means an increase in the CPU threads performing reductions as well. This is not exactly the case for the Hybrid/CUDA and NCCL/CUDA schemes because we anticipate the communication aspect of these collectives to be more dominant than their compute aspect in the first place.

Test Case	MPI/OpenACC	Hybrid/CUDA	NCCL/CUDA
Small	37%	38%	49%
Medium	44%	34%	39%
Large	41%	35%	33%

Table 5.17 Strong-scaling efficiency of the MPI/OpenACC, Hybrid/CUDA and NCCL/CUDA schemes on Small, Medium and Large matrices.

Apart from the comparison between these schemes, we believe the lack of efficiency across the board is due to the communication aspect of the distributed SpMV algorithm scaling worse than its compute aspect in our strong scaling experiments. To understand this issue, we numerically quantify the total number of floating point operations (FLOP) and the communication volume observed in the distributed SpMV. We specifically define these two numbers for the parts where computation and communication can be overlapped as follows:

Total FLOP: This refers to the total number of floating point operations performed in local SpMV and SpMV-transpose kernels across all GPUs for a given symmetric matrix H. Regardless of the GPU count,

$$Total\ FLOP = 2 \times 2 \times nnz_H \tag{5.1}$$

where nnz_H is the number of nonzero elements in half of the symmetric matrix. This is because we perform one addition and one multiplication per nonzero element in both kernels.

Total Communication Volume: This refers to the total communication volume of Bcast and Reduce, overlapped collectives, across all GPUs for a given vector W. For the sake of simplicity, we calculate this number for the *ncclBroadcast* and *ncclReduce* calls using the ring algorithm. Under these assumptions,

$$Total\,Communication\,Volume = 2 \times 4 \times \left(\frac{n_d - 1}{2}\right) \times \frac{S_W}{n_d} \times \left(\frac{n_d + 1}{2}\right)^{-1} \tag{5.2}$$

where S_W is the number of elements in W. Recall that W is distributed among n_d column groups, where each group has $(n_d + 1)/2$ processes, or GPUs (see Section 5.1.1). First two terms in this equation come from the number of collectives, which is two, and the byte per element, which is four since we store numbers in single-precision in MFDn. The remaining terms account for each sub-vector of W being partitioned into $(n_d + 1)/2$ segments where each process sends or receives $(n_d - 1)/2$ of them during the ring algorithm.

Test Case	# of GPUs	Total FLOP	Total Comm Vol	Ratio (FLOP/B)
Small	28	5.16E11	3.97GB	130
	276	5.16E11	14.6GB	35.4
Medium	91	1.90E12	27.6GB	68.9
	496	1.90E12	69.1GB	27.6
Large	276	5.22E12	86.1GB	60.7
	1128	5.22E12	180GB	29.0

Table 5.18 Number of floating point operations performed per byte communicated in the distributed SpMV algorithm.

In Table 5.18, we show the ratio between the total FLOP and total communication volume, which corresponds to the number of operations performed per byte communicated. We see that this FLOP/B ratio goes down in our strong scaling experiments. This is because our compute volume stays the same although our communication volume grows linearly with respect to n_d according to Eq. 5.2.

Let P denote the number of processes or GPUs used.

$$P = n_d \times (n_d + 1)/2 \implies n_d = \frac{1}{2}\sqrt{8P+1} \tag{5.3}$$

This means our communication volume grows by the square root of the number of GPUs used. Given that the compute volume stays constant, all evaluated schemes eventually become communication-bound. Therefore, we believe it is reasonable to see these schemes to deviate from the ideal solver time in our strong-scaling experiments.

Moreover, we only considered the overlapped communication calls so far, *i.e.,* Bcast and Reduce. We also perform an Allgatherv and a Reduce_scatter that are not overlapped at all. Considering that these collectives have the same growth factor as the other two, their presence is another factor that deters us from reaching ideal scaling.

Lastly, we perform our experiments on Perlmutter where we share the network with other jobs that are scheduled to run concurrently as SLURM sees fit. Using a higher number of nodes, or GPUs, makes our runs more prone to network contention and noise. This is because we request more nodes and the allocated nodes may be scattered around the cluster. As such, the observed network bandwidth is likely to become lower when we increase the number of GPUs in the strong-scaling scenario, which is yet another factor to consider when interpreting the efficiency numbers in Table 5.18.

5.4 LOBPCG Experiments

In this section, we share the experiment results of all four schemes show in Table 5.1 for the LOBPCG eigensolver.

5.4.1 Experimental Setup

We report the performance of LOBPCG in MFDn when it is run for 20 iterations on Perlmutter. Please refer to Section 5.3.1 to see the hardware and system specifications of Perlmutter, software versions or the test cases used in our experiments.

5.4.2 Results

In Table 5.19, we show the experiment results for 20 LOBPCG iterations to compute eight eigenvalues with MPI/OpenACC and MPI/CUDA. These numbers suggest that changing the local SpMM kernels from OpenACC to CUDA does not correspond to any improvement in the execution time as the speedup achieved is never more than 5%. In fact, we observe occasional slowdowns with the CUDA code up to 2%.

Test Case	# of Nodes	# of GPUs	MPI/OpenACC	MPI/CUDA
XSmall	4	15	4.93	4.69
	12	45	2.41	2.38
	23	91	2.29	2.20
Small	12	45	21.56	21.41
	23	91	15.09	15.09
	48	190	10.64	10.82
Medium	23	91	50.15	50.05
	48	190	35.51	36.03
	95	378	25.82	26.08

Table 5.19 Solver time in seconds for 20 LOBPCG iterations to compute eight eigenvalues with MPI/OpenACC and MPI/CUDA.

The employment of Hybrid/CUDA and NCCL/CUDA schemes have proven to be useful in Section 5.3. As such, we used these schemes to improve the communication aspect and thereby performance of the LOBPCG solver. In Table 5.20, we show the speedup numbers achieved with these schemes over MPI/OpenACC.

Test Case	# of Nodes	# of GPUs	Hybrid/CUDA	NCCL/CUDA
XSmall	4	15	3.02	3.16
	12	45	1.80	2.08
	23	91	1.78	2.01
Small	12	45	2.06	1.51
	23	91	1.87	1.27
	48	190	1.59	1.57
Medium	23	91	1.89	1.33
	48	190	1.67	1.60
	95	378	1.40	1.40

Table 5.20 Speedup achieved for 20 LOBPCG iterations to compute eight eigenvalues with Hybrid/CUDA and NCCL/CUDA over MPI/OpenACC based on Table 5.19.

We observe that these schemes do not appear to be as effective as they are on Lanczos. For the Small problem on 190 GPUs, for example, we have seen over 4.9× speedup achieved with NCCL/CUDA in Table 5.15. However, switching from SpMV-based Lanczos to SpMM-based LOBPCG brings the NCCL's improvement in solver time down to 1.6×. We also notice a similar trend with Hybrid/CUDA for this problem where our speedup decreases from 2.9× to 1.6× (see Table 5.12).

Note that SpMM with eight vectors yields an eight-fold increase in communication volume compared to SpMV. To understand the effect of this change on the performance, we show in Table 5.21 the Reduce time observed with MPI, P2P and NCCL for 20 LOBPCG iterations. We see that NCCL still appears to be faster than MPI on all problem sizes. However, we no longer see over an order of magnitude improvement with NCCL over MPI as shown in Table 5.13.

Test Case	# of Nodes	# of GPUs	MPI	P2P	NCCL
Small	12	45	8.64	3.06	2.36
	23	91	6.05	2.84	1.99
	48	190	4.43	2.79	1.40
Medium	23	91	20.7	9.77	8.13
	48	190	15.2	9.77	5.84
	95	378	10.5	9.74	4.36

Table 5.21 Reduce time in seconds for 20 LOBPCG iterations to compute eight eigenvalues with MPI, P2P and NCCL. Reduce calls are isolated with barriers. All processes take measurements separately and we report the maximum time among them for each collective.

In Table 5.22, we compare the average Reduce time in SpMV and in SpMM with NCCL. The ratio computed indicates that SpMM's Reduce is 13.4× - 25.5× costlier than SpMV's for the same problem. That is, increasing the communication volume by 8× slows down the Reduce call up to 25.5×. This corresponds to a 3.2× slowdown in the achieved bandwidth with NCCL. We believe this is the main reason why we only see a modest improvement with NCCL/CUDA over MPI/OpenACC for the LOBPCG solver.

The NCCL/CUDA scheme presented here for LOBPCG lacks the optimizations incorporated in NCCL-v3 and NCCL-v4 as mentioned in Section 5.3.6. That is, we do not truly

overlap communication and computation. Also, we do not configure NCCL communicators to use optimal thread block count. With these improvements, we may potentially achieve higher speedup numbers.

Test Case	# of Nodes	# of GPUs	Reduce in SpMV	Reduce in SpMM	Ratio
Small	12	45	0.005	0.118	23.60
	23	91	0.004	0.100	25.51
	48	190	0.003	0.070	23.33
Medium	23	91	0.028	0.407	14.78
	48	190	0.020	0.292	14.78
	95	378	0.016	0.218	13.46

Table 5.22 Reduce time in seconds for SpMV in a single Lanczos iteration, Reduce time for SpMM in a single LOBPCG iteration and the ratio of Reduce time in SpMM time over Reduce time in SpMV with NCCL/CUDA.

5.4.3 Pipelined SpMM to Hide Communication Costs

In our distributed SpMM implementation, which follows the same steps as the distributed SpMV as outlined in Section 5.1.2, we overlap the local SpMM with Bcast, and the local SpMM-transpose with Reduce. This means there is no compute task to overlap Allgatherv or Reduce_scatter with. Our experiments with MPI suggests that although the processes participating in Reduce_scatter are in close proximity to each other, this collective is still quite costly. We believe with a right pipelining technique, this Reduce_scatter cost can in fact be hidden.

We highlight in Section 5.1.2 that Reduce_scatter is performed on the output of local SpMM-transpose. However, Reduce_scatter calls can be replaced with multiple Reduce calls, each of which have a different root process. Turning a single Reduce_scatter to multiple Reduces provides us with the freedom of commencing a Reduce call earlier. That is, as soon as the local SpMM-transpose operations are completed for a partition of the output vectors, we can call MPI_Reduce.

We can partition the local SpMM-transpose with respect to the row dimension of the output vectors, which maps to partitioning the input matrix in the column dimension. Then,

launching a separate local SpMM-transpose kernel for each partition allows us to pipeline these kernels with Reduce calls.

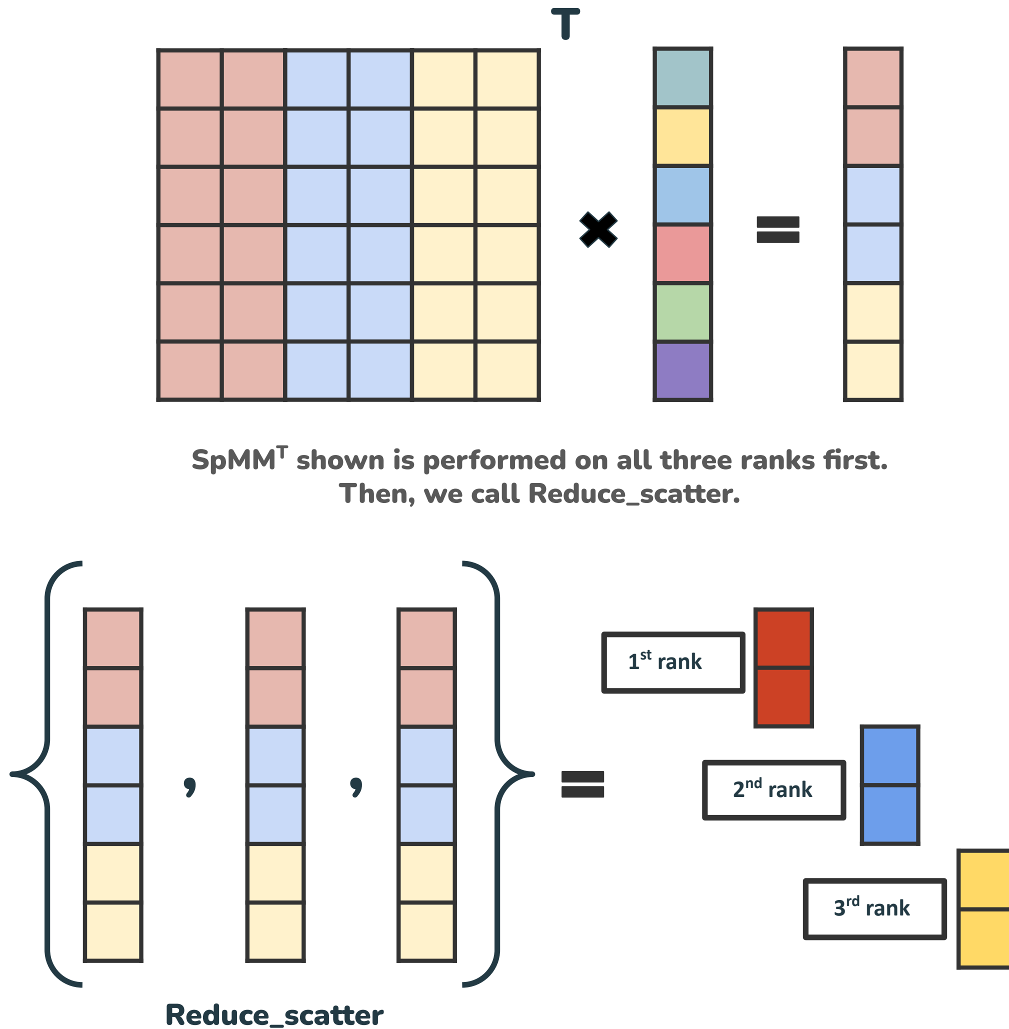
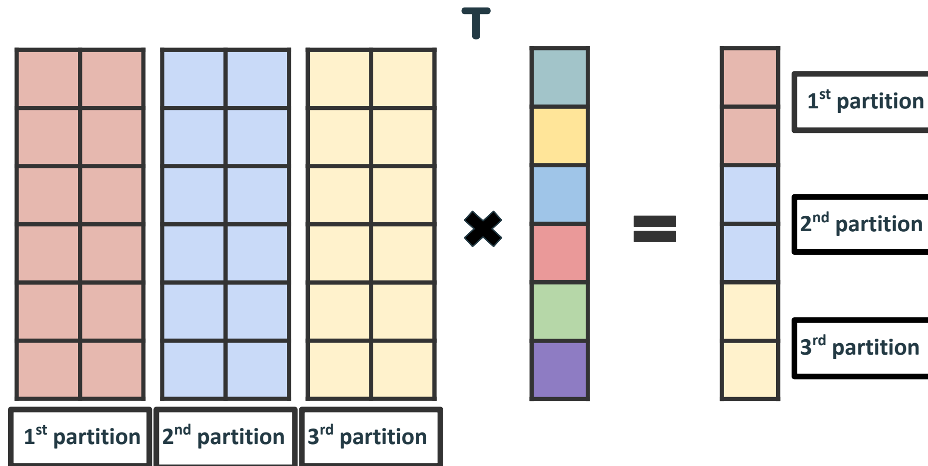


Figure 5.13 Local SpMM-transpose followed by Reduce_scatter in distributed SpMM.

To better illustrate this pipelining technique in our SpMM algorithm, we show a couple of drawings provided in Figure 5.13 and 5.14 on a small sample example. In these figures, three processes perform the local SpMM-transpose and participate in the Reduce_scatter call.

Figure 5.13 shows the strictly ordered nature of this kernel and collective call where we have to complete the local SpMM-transpose first before calling the Reduce_scatter. Fig-

ure 5.14, on the other hand, demonstrates that by partitioning the SpMM-transpose into smaller kernels and Reduce_scatter into multiple Reduce calls, we can pipeline these two operations. For instance, while the Reduce call for the first partition is taking place, GPU threads can work on the second partition.



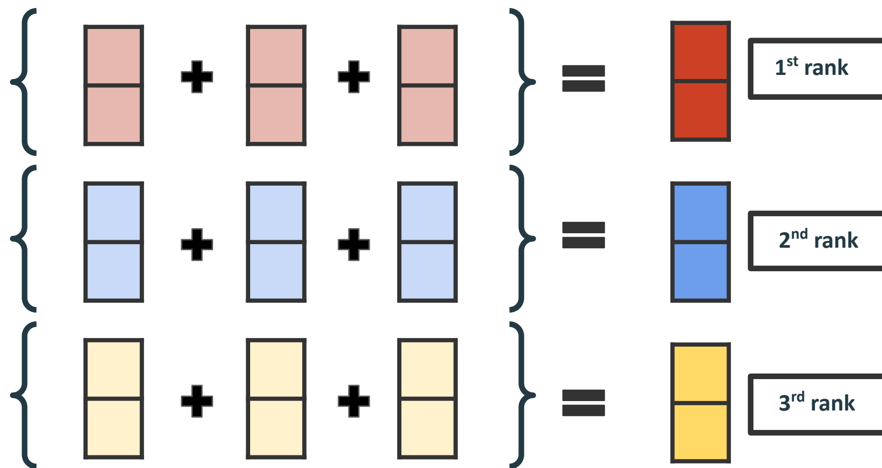


Figure 5.14 Partitioned local SpMM-transpose followed by Reduce calls in pipelined distributed SpMM.

Notice that we can use a similar idea to partition the local SpMM kernel into multiple kernels and pipeline these kernels with partitioned Reduce calls. This means that the local SpMM can be overlapped with *MPI_Bcast* and pipelined with multiple *MPI_Reduce* calls while the local SpMM-transpose can be pipelined with *MPI_Reduce_scatter* as described

above. We refer to the implementation achieving such overlap with this pipelining technique as Pipe/CUDA.

We have shown in Table 5.11 how costly an *MPI_Reduce* call can become and how effective replacing it call with P2P calls. As such, we replace every Reduce collective with corresponding P2P calls in our Pipe/CUDA scheme.

In Table 5.23, we share the experiment results achieved with Pipe/CUDA for 20 LOBPCG iterations, and we compare this scheme's performance to MPI/CUDA and Hybrid/CUDA. The reason why we compare Pipe/CUDA to Hybrid/CUDA is we are effectively using the same collective and P2P combination. The only difference is, Pipe/CUDA uses fine-grained calls for Reduce and Reduce_scatter in an effort to achieve a better overlap.

Test Case	# of Nodes	# of GPUs	MPI/CUDA	Hybrid/CUDA	Pipe/CUDA
XSmall	4	15	4.77	1.63	1.62
	12	45	2.49	1.34	1.31
	23	91	2.20	1.29	1.29
Small	12	45	21.5	10.4	10.3
	23	91	10.9	8.02	8.14
	48	190	10.9	6.71	6.93
Medium	23	91	49.8	26.5	26.9
	48	190	36.9	21.2	22.3
	95	378	30.9	18.4	19.7

Table 5.23 Solver time in seconds for 20 LOBPCG iterations to compute eight eigenvalues with MPI/CUDA, Hybrid/CUDA and Pipe/CUDA.

These results show that Pipe/CUDA performs similar to Hybrid/CUDA while obtaining a significant improvement over MPI/CUDA. Therefore, replacing the Reduce collectives with the P2P calls in Pipe/CUDA optimizes the performance. However, partitioning the local SpMM and SpMM-transpose operations to overlap with Bcast, Reduce and Reduce_scatter virtually yields no additional improvements.

We see the performance of Pipe/CUDA as underwhelming. We believe the lack of a meaningful speedup over Hybrid/CUDA calls for further investigation. The number of partitions used in the current implementation is equal to the number of processes in each row or column communication group. Tuning the pipelined code by experimenting with differ-

ent partition counts may potentially improve the performance. Moreover, pipelining NCCL instead of MPI may provide us with greater benefits. This is because our experiments with Lanczos and LOBPCG show that NCCL seems to prefer smaller communication volume, with which our pipelining technique can help. Lastly, we have to keep in mind the growing kernel launch overheads with pipelining, which may eventually become the bottleneck at large-scale.

5.4.4 Discussion

In Ch. 4 regarding hybrid eigensolvers, we examine the GFLOPS numbers for SpMV and SpMM on many-core shared memory architectures. We see in Fig 4.12 that SpMM yields a much higher GFLOPS value than SpMV for two reasons. First, SpMM has a higher arithmetic intensity than SpMV, which is a significant factor given the memory-bound nature of both kernels. Secondly, the inner-most loop that updates all output vectors for a given nonzero element can be executed within a single instruction cycle by using SIMD instructions on CPUs. SpMM therefore yields around $5\times$ more GFLOPS rate than SpMV on a many-core CPU architecture such as a 64-core AMD EPYC processor. This ratio gives a great advantage to block eigensolvers that utilize SpMM such as LOBPCG in comparison to SpMV-based eigensolvers such as Lanczos.

We do a similar comparison between the distributed SpMV of Lanczos and SpMM of LOBPCG for the MPI/OpenACC scheme within MFDn on GPU architectures. In Table 5.24, we show the overall execution time of SpMV in 100 Lanczos iterations and of SpMM in 20 LOBPCG iterations for the Small and Medium problems. In 20 LOBPCG iterations, we effectively perform 160 SpMVs. We define the ratio as how expensive an SpMV of Lanczos is compared to an SpMV of LOBPCG, which gives us the effective GFLOPS ratio between SpMV and SpMM in distributed memory.

These numbers indicate that LOBPCG no longer possesses the same advantage with SpMM on distributed GPU architectures. We believe this is due to the fact that the inner-most loop can no longer be executed in a single instruction cycle. Instead, each update to a

Test Case	# of Nodes	# of GPUs	SpMV Time	SpMM Time	Ratio
Small	12	45	22.8	18.5	1.97
	23	91	12.8	12.9	1.59
	48	190	8.37	8.76	1.53
Medium	23	91	40.3	44.6	1.44
	48	190	27.7	31.6	1.40
	95	378	20.5	22.7	1.45

Table 5.24 SpMV time in seconds for 100 Lanczos iterations, SpMM time in seconds for 20 LOBPCG iterations with MPI/OpenACC and ratio between 1/100 of SpMV time and 1/60 of SpMM time. SpMV and SpMM time is based on experiment results shown in Table 5.5 and 5.19.

vector in this loop is done through atomic operations [40]. The expensive atomic operations are likely to take away the advantage of a higher arithmetic intensity in SpMM over SpMV as well since atomics may become the bottleneck rather than the memory access to the sparse matrix or input and output vectors.

Lastly, the communication cost in these distributed algorithms is another reason that brings this GFLOPS ratio down. This is because the collectives play an equalizing role, negating the performance gains achieved by the local SpMM and SpMM-transpose kernels. The highest ratio that we see in Table 5.24 is still below two. Given that these low ratios render an average LOBPCG iteration relatively more expensive, Lanczos has been the default solver in MFDn's production code since the code has been ported to GPUs.

5.5 Conclusion of This Work

In this work, we started with MFDn that has these eigensolvers developed for GPUs by using MPI collectives for communication and OpenACC directives for computation. Next, we investigated the custom data distribution and the proprietary SpMV algorithm conformal to this distribution within the MFDn solver. Then, we showed the scaling issues observed with this initial MPI/OpenACC scheme and examined the potential root causes thereof. Finally, we offered an improvement in the overall solver time for both Lanczos and LOBPCG by optimizing the compute and communication aspect of SpMV.

To improve the compute aspect of SpMV, we rewrote the existing OpenACC kernels with

CUDA, a low-level language native to NVIDIA GPUs. Although we achieved decent speedup numbers on lower GPU counts in our Perlmutter experiments, we saw that the advantage of using CUDA disappeared on higher GPU counts. We attributed the vanishing improvement to the communication aspect of SpMV becoming the bottleneck on large-scale.

To improve the communication aspect, we proposed two approaches. First, we utilized asynchronous point-to-point (P2P) communication in place of expensive MPI collectives. Then, we switched to NCCL collectives in place of MPI's with minimal change. We showed significant performance improvements across the board for Lanczos with these changes. For LOBPCG, on the other hand, we observed more modest improvements. We attributed the lack of performance on this solver to an eight-fold increase in communication volume.

We analyzed the strong-scaling efficiency of our schemes on Lanczos. For this analysis, we modeled the compute and communication volume of the SpMV algorithm to calculate the floating point operations performed per byte communicated. We then realized that the total communication volume grows by the square root of the process count while the compute volume stays constant in the strong-scaling scenario. As such, we concluded that communication would eventually become the bottleneck in large-scale with our current SpMV algorithm.

Lastly, we compared the overall SpMV and SpMM time within Lanczos and LOBPCG to have an understanding of the GFLOPS values achieved by SpMV and SpMM. We noticed that although the ratio between these values favors SpMM on shared CPU architectures, it is no longer the case on distributed GPU architectures. We then attributed this change in GFLOPS ratio to (i) the difference between CPU and GPU architectures and (i) the presence of communication overheads in distributed scenario. We finally concluded that for these reasons, SpMM-based block eigensolvers such as LOBPCG lose their advantage against SpMV-based eigensolvers such as Lanczos on multi-GPU experiments.

CHAPTER 6

CONCLUSION AND FUTURE WORK

In this thesis, we described different ways to accelerate large-scale sparse eigensolvers. Towards this end, we explored asynchronous runtime systems, hybrid algorithms and heterogeneous architectures equipped with GPUs.

In the first part, we evaluated the performance of three task-parallel programming models, OpenMP, HPX and Regent on shared memory architectures. We compared their performance against the traditional BSP model for two iterative sparse eigensolvers: Lanczos and LOBPCG. We then demonstrated their merits on two architectures, Intel Broadwell (a multicore processor) and AMD EPYC (a modern manycore processor). We observed that these frameworks achieve up to 13.7× fewer cache misses over an efficient BSP implementation across L1, L2 and L3 cache layers. They also obtained up to 9.9× improvement in execution time over the same BSP implementation.

In the second part, after seeing HPX's success with our first work, we wanted to use HPX as the backbone of the large-scale sparse solver and graph analytics framework we wanted to develop. However, our experiments with HPX for the distributed Lanczos solver using up to 512 cores suggested that HPX was slower than the hybrid MPI+OpenMP model. Investigating this lack of improvement showed that HPX involves a considerable runtime overheads whenever a network communication call is used. As such, we gave up on utilizing HPX's asynchronous execution model in the distributed memory architectures and pivoted our work towards accelerating sparse eigensolvers at the algorithmic level, which laid the foundation for our next work.

In the third part, we examined and compared a few iterative methods for solving large-scale eigenvalue problems arising from nuclear structure calculations besides Lanczos and LOBPCG. In particular, we discussed the possibility of using block Lanczos method, a Chebyshev filtering based subspace iterations and the residual minimization method accelerated by direct inversion of iterative subspace (RMM-DIIS). Although the RMM-DIIS

method does not exhibit rapid convergence when the initial approximations to the desired eigenvectors are not sufficiently accurate, we showed that it can be effectively combined with either the block Lanczos and LOBPCG methods to yield a hybrid eigensolver that has several desirable properties. We described a few practical issues that need to be addressed to make the hybrid solver efficient and robust.

In the fourth part, we accelerated the Lanczos and LOBPCG solvers by improving the communication and computational part of their distributed SpMV/SpMM algorithm on heterogeneous architectures. We worked on problem sizes with up to 1.3 trillion nonzeros by using up to 1128 GPUs. With these large scale experiments, first, we obtained up to 2.0× improvement by going from an optimized OpenACC to a hand-optimized CUDA code for the local SpMM and SpMM-transpose kernels. We then improved the cost of communication by understanding the limitations of MPI collectives and employing asynchronous point-to-point (P2P) and NCCL calls instead. On Lanczos, P2P and NCCL improvements paired with CUDA kernels achieved up to 2.89× (P2P) and 4.85× (NCCL) speedup over the MPI/OpenACC scheme. On LOBPCG, which was more demanding due to the higher communication volume of SpMM over SpMV, these P2P and NCCL improvements led to up to 3.0× and 3.1× speedup in the overall solver time.

As future work, we aim to optimize the GPU-parallel solver codes further by overlapping communication and computation correctly via NVSHMEM. We believe NVSHMEM has the potential to hide the communication costs to a better degree through explicit thread block specialization. NVSHMEM can also help with the kernel launch overheads given the iterative nature of our solvers where GPU kernels are launched and teared down every iteration. Last but not the least, we are currently working towards porting our work to AMD GPUs on Frontier. As such, we consider utilizing HIP (Heterogeneous-Compute Interface for Portability) on the compute part and RCCL (ROCm Communication Collectives Library) on the communication part. We want to see how well HIP and RCCL perform on hundreds of AMD MI250x GPUs against OpenACC and MPI-based implementations.

BIBLIOGRAPHY

[1] Md Afibuzzaman, Fazlay Rabbi, M Yusuf Özkaya, Hasan Metin Aktulga, and Ümit V Çatalyürek. Deepsparse: A task-parallel framework for sparsesolvers on deep memory architectures. In *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pages 373–382. IEEE, 2019.

[2] Hasan Metin Aktulga, Aydin Buluç, Samuel Williams, and Chao Yang. Optimizing sparse matrix-multiple vectors multiplication for nuclear configuration interaction calculations. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 1213–1222. IEEE, 2014.

[3] Hasan Metin Aktulga, Chao Yang, Esmond G Ng, Pieter Maris, and James P Vary. Topology-aware mappings for large-scale eigenvalue problems. In *European Conference on Parallel Processing*, pages 830–842. Springer, 2012.

[4] Abdullah Alperen, Md Afibuzzaman, Fazlay Rabbi, M Yusuf Ozkaya, Umit Catalyurek, and Hasan Metin Aktulga. An evaluation of task-parallel frameworks for sparse solvers on multicore and manycore cpu architectures. In *Proceedings of the 50th International Conference on Parallel Processing*, pages 1–11, 2021.

[5] Abdullah Alperen, Hasan Metin Aktulga, Pieter Maris, and Chao Yang. Hybrid eigensolvers for nuclear configuration interaction calculations. *Computer Physics Communications*, 292:108888, 2023.

[6] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, S Hammerling, Alan McKenney, et al. Lapack users' guide, vol. 9. *Society for Industrial Mathematics*, 39, 1999.

[7] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Euro-Par - 15th International Conference on Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 863–874, Delft, The Netherlands, August 2009. Springer.

[8] Bruce R. Barrett, Petr Navratil, and James P. Vary. Ab initio no core shell model. *Prog. Part. Nucl. Phys.*, 69:131–181, 2013.

[9] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. Legion: Expressing locality and independence with logical regions. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. IEEE, 2012.

[10] L Susan Blackford, Jaeyoung Choi, Andy Cleary, Eduardo D'Azevedo, James Demmel, Inderjit Dhillon, Jack Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, et al. *ScaLAPACK users' guide.* SIAM, 1997.

[11] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J Dongarra. Parsec: Exploiting heterogeneity to enhance scalability. *Computing in Science & Engineering*, 15(6):36–45, 2013.

[12] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Thomas Herault, Pitior Luszczek, and Jack Dongarra. Dense linear algebra on distributed heterogeneous hardware with a symbolic dag approach. Technical report, Lawrence Berkeley National Lab.(LBNL), Berkeley, CA (United States), 2012.

[13] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244. ACM, 2009.

[14] Brandon Cook, Pieter Maris, Meiyue Shao, Nathan Wichmann, Marcus Wagner, John O'Neill, Thanh Phung, and Gaurav Bansal. High performance optimizations for nuclear physics code mfdn on knl. In *International Conference on High Performance Computing*, pages 366–377. Springer, 2016.

[15] NVIDIA Corporation. Overview of NCCL. https://docs.nvidia.com/deeplearning/nccl/user-guide/docs/overview.html, 2024. Accessed: 2024-07-21.

[16] Gregor Daiß, Parsa Amini, John Biddiscombe, Patrick Diehl, Juhan Frank, Kevin Huck, Hartmut Kaiser, Dominic Marcello, David Pfander, and Dirk Pflüger. From piz daint to the stars: simulation of stellar mergers using high-level abstractions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–37, 2019.

[17] Timothy A Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[18] Jos L. M. Van Dorsselaer, M. E. Hoschstenbach, and H. A. Van der Vorst. Computing probabilistic bounds for extreme eigenvalues of symmetric matrices with the Lanczos method. *SIAM J. Matrix Anal. Appl.*, 22(3):837–852, 2000.

[19] Jed A Duersch, Meiyue Shao, Chao Yang, and Ming Gu. A robust and efficient implementation of lobpcg. *SIAM Journal on Scientific Computing*, 40(5):C655–C676, 2018.

[20] MD Feit, JA Fleck Jr, and A Steiger. Solution of the schrödinger equation by a spectral method. *Journal of Computational Physics*, 47(3):412–433, 1982.

[21] G. H. Golub and R. Underwood. The block Lanczos method for computing eigenvalues. In J. R. Rice, editor, *Mathematical Software III*, pages 361–377, 1977.

[22] Gene H. Golub and Charles F. Van Loan. *Matrix Computations (3rd Ed.)*. Johns Hopkins University Press, 1996.

[23] Khaled Hamidouche, Ammar Ahmad Awan, Akshay Venkatesh, and Dhabaleswar K Panda. Cuda m3: Designing efficient cuda managed memory-aware mpi by exploiting gdr and ipc. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 52–61. IEEE, 2016.

[24] M. T. Heath. Sparse matrix computations. In *The 23rd IEEE Conference on Decision and Control*, pages 662–665, 1984.

[25] Thomas Heller, Patrick Diehl, Zachary Byerly, John Biddiscombe, and Hartmut Kaiser. Hpx–an open source c++ standard library for parallelism and concurrency. *Proceedings of OpenSuCo*, page 5, 2017.

[26] U. Hetmaniuk and R. Lehoucq. Basis selection in LOBPCG. *J. Comput. Phys.*, 218:324–332, 2006.

[27] Torsten Hoefler and Marc Snir. Generic topology mapping strategies for large-scale parallel architectures. In *Proceedings of the international conference on Supercomputing*, pages 75–84, 2011.

[28] Z. Jia and G. W. Stewart. An analysis of the Rayleigh-Ritz method for approximating eigenspaces. *Math. Comput.*, 70:637–647, 2001.

[29] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. Hpx: A task based programming model in a global address space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, pages 1–11, 2014.

[30] Laxmikant V Kale and Sanjeev Krishnan. Charm++ a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108, 1993.

[31] Jeremy Kepner, David Bade, Aydın Buluç, John Gilbert, Timothy Mattson, and Henning Meyerhenke. Graphs, matrices, and the graphblas: Seven good reasons. *arXiv preprint arXiv:1504.01039*, 2015.

[32] A. V. Knyazev. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM J. Sci. Comput.*, 23(2):517–541, 2001.

[33] Andrew V Knyazev. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM journal on scientific computing*, 23(2):517–541, 2001.

[34] Alexey Kukanov and Michael J Voss. The foundations for scalable multi-core software in intel threading building blocks. *Intel Technology Journal*, 11(4), 2007.

[35] Abhishek Kulkarni and Andrew Lumsdaine. A comparative study of asynchronous many-tasking runtimes: Cilk, charm++, parallex and am++. *arXiv preprint arXiv:1904.00518*, 2019.

[36] Cornelius 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.

[37] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[38] Wonchan Lee, Elliott Slaughter, Michael Bauer, Sean Treichler, Todd Warszawski, Michael Garland, and Alex Aiken. Dynamic tracing: Memoization of task graphs for dynamic task-based runtimes. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 441–453. IEEE, 2018.

[39] R. Li and Y. Zhou. Bounding the spectrum of large Hermitian matrices. *Linear Algebra and its Applications*, 435:480–493, 2011.

[40] Pieter Maris, Chao Yang, Dossay Oryspayev, and Brandon Cook. Accelerating an iterative eigensolver for nuclear structure configuration interaction calculations on gpus using openacc. *Journal of Computational Science*, 59:101554, 2022.

[41] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, and Dan Ginsburg. *OpenCL programming guide*. Pearson Education, 2011.

[42] Andrew Ng, Michael Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. *Advances in neural information processing systems*, 14, 2001.

[43] ARB OpenMP. Openmp application program interface version 4.0, 2013.

[44] B. N. Parlett. *The Symmetric Eigenvalue Problem*. Prentice Hall, Englewood Cliffs, 1980.

[45] P. Pulay. Convergence acceleration of iterative sequences: The case of SCF iteration. *Chem. Phys. Lett.*, 73:393–398, 1980.

[46] Arch D Robison. Cilk plus: Language support for thread and vector parallelism. *Talk at HP-CAST*, 18:25, 2012.

[47] Gregory Ruetsch and Massimiliano Fatica. *CUDA Fortran for scientists and engineers: best practices for efficient CUDA Fortran programming*. Elsevier, 2013.

[48] Y. Saad. *Numerical Methods for Large Eigenvalue Problems: Revised Edition*. Number 66 in Classics in Applied Mathematics. SIAM, Philadelphia, 2011.

[49] Youcef Saad. Chebyshev acceleration techniques for solving nonsymmetric eigenvalue problems. *Math. Comp.*, 42:567–588, 1984.

[50] Erik Saule, Hasan Metin Aktulga, Chao Yang, Esmond G Ng, and Ümit V Çatalyürek. An out-of-core task-based middleware for data-intensive scientific computing. In *Handbook on Data Centers*, pages 647–667. Springer, 2015.

[51] Meiyue Shao, H. Metin Aktulga, Chao Yang, Esmond G. Ng, Pieter Maris, and James P. Vary. Accelerating nuclear configuration interaction calculations through a preconditioned block iterative eigensolver. *Computer Physics Communications*, 222:1–13, 2018.

[52] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. Regent: a high-productivity programming language for hpc with logical regions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2015.

[53] Philip Sternberg, Esmond G Ng, Chao Yang, Pieter Maris, James P Vary, Masha Sosonkina, and Hung V Le. Accelerating configuration interaction calculations for nuclear structure. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC08)*, pages 1–12. IEEE, 2008.

[54] Philip Sternberg, Esmond G Ng, Chao Yang, Pieter Maris, James P Vary, Masha Sosonkina, and Hung Viet Le. Accelerating configuration interaction calculations for nuclear structure. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12. IEEE, 2008.

[55] Przemysław Stpiczyński. Language-based vectorization and parallelization using intrinsics, openmp, tbb and cilk plus. *The Journal of Supercomputing*, 74(4):1461–1472, 2018.

[56] Peter Thoman, Kiril Dichev, Thomas Heller, Roman Iakymchuk, Xavier Aguilar, Khalid Hasanov, Philipp Gschwandtner, Pierre Lemarinier, Stefano Markidis, Herbert Jordan, et al. A taxonomy of task-based parallel programming technologies for high-performance computing. *The Journal of Supercomputing*, 74(4):1422–1434, 2018.

[57] Hilario Torres, Manolis Papadakis, and Lluís Jofre Cruanyes. Soleil-x: turbulence, particles, and radiation in the regent programming language. In *SC'19: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–4, 2019.

[58] James P. Vary, Robert Basili, Weijie Du, Matthew Lockner, Pieter Maris, Dossay Oryspayev, Soham Pal, Shiplu Sarker, Hasan M. Aktulga, Esmond Ng, Meiyue Shao, and Chao Yang. Ab Initio No Core Shell Model with Leadership-Class Supercomputers. In A.M. Shirokov and A.I. Mazur, editors, *Proc. of Int. Conf. 'Nuclear Theory in the Supercomputing Era — 2016'*, pages 15–35, Khabarovsk, Russia, 2018.

[59] Chun-Kun Wang. Selection of parallel runtime systems for tasking models. In *2017 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1091–1096. IEEE, 2017.

[60] Hao Wang, Sreeram Potluri, Miao Luo, Ashish Kumar Singh, Sayantan Sur, and Dhabaleswar K Panda. Mvapich2-gpu: optimized gpu to gpu communication for infiniband clusters. *Computer Science-Research and Development*, 26(3):257–266, 2011.

[61] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel computing*, 27(1-2):3–35, 2001.

[62] Kyle B Wheeler, Richard C Murphy, and Douglas Thain. Qthreads: An api for programming with millions of lightweight threads. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8. IEEE, 2008.

[63] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC'07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12. IEEE, 2007.

[64] D. M. Wood and A. Zunger. A new method for diagonalising large matrices. *Journal of Physics A: Mathematical and General*, 18(9):1343, jun 1985.

[65] Y. Zhou, Y. Saad, M. L. Tiago, and J. R. Chelikowsky. Parallel self-consistent-field calculations via Chebyshev-filtered subspace acceleration. *Phy. Rev. E*, 74:066704, 2006.

[66] Yunkai Zhou, James R. Chelikowsky, and Yousef Saad. Chebyshev-filtered subspace iteration method free of sparse diagonalization for solving the Kohn–Sham equation. *J. Comput. Phys.*, 274:770–782, 2014.