Evaluation of Distributed Programming Models and Extensions to Task-based Runtime Systems

A Dissertation Presented for the

Doctor of Philosophy

Degree

The University of Tennessee, Knoxville

Yu Pei

December 2022

To my parents Wenhai Pei and Yinbian Fan,
my fiancee Anyi Wang for their love, trust, and support.

Acknowledgments

I would like to thank my advisor, Dr. Jack Dongarra, for giving me the opportunity to join the Innovative Computing Laboratory (ICL) as a Graduate Research Assistant (GRA). I am also grateful to Dr. Dongarra for supporting my research. It was a great privilege to work with him, while his experience and knowledge on many aspects of HPC benefited me greatly.

I would also like to thank my co-advisor Dr. George Bosilca, for believing in me from the beginning, and his support and guidance throughout this entire time. His vast knowledge and patience made my study a pleasant experience, and I feel lucky to be working with him on many projects. The fields of distributed computing and task-based runtime systems is an active research area and I have learned tremendous amount through working with the PaRSEC system that he spearheaded.

I am also grateful to Dr. Michael Berry and Dr. Ichitaro Yamazaki for serving on my dissertation committee. I greatly appreciate their time and invaluable guidance on my dissertation.

Dr. Piotr Luszczek helped me at the beginning of this journey when I was lost, and he also graciously offers to improve this draft. I would like to express my appreciation to my current and former colleagues at ICL, including Dr. Thomas Herault, Dr. Aurelien Bouteiller, Dr. Anthony Danalis, Dr. Reazul Hoque, Dr. Thananon Patinyasakdikul, Dr. David Eberius, Dr. Zhong Dong, Dr. Qinglei Cao, Yicheng Li and Jiali Li and others, for their help in debugging and profiling my programs as well as all the wonderful coffee chats. I will forever cherish those as timeless memories.

Lastly, I would like to express my deepest gratitude to my parents Wenhai Pei and Yinbian Fan, and my fiancee Anyi Wang for their love, trust, and unrelenting support. It was a long and winding journey, and their believe in me motivated me to continue my pursuit of knowledge.

Abstract

High Performance Computing (HPC) has always been a key foundation for scientific simulation and discovery. And more recently, deep learning models' training have further accelerated the demand of computational power and lower precision arithmetic. In this era following the end of Dennard's Scaling and when Moore's Law seemingly still holds true to a lesser extent, it is not a coincidence that HPC systems are equipped with multi-cores CPUs and a variety of hardware accelerators that are all massively parallel. Coupling this with interconnect networks' speed improvements lagging behind those of computational power increases, the current state of HPC systems is heterogeneous and extremely complex.

This was heralded as a great challenge to the software stacks and their ability to extract performance from these systems, but also as a great opportunity to innovate at the programming model level to explore the different approaches and propose new solutions. With usability, portability, and performance as the main factors to consider, this dissertation first evaluates some of the widely used parallel programming models (MPI, MPI+OpenMP, and task-based runtime systems) ability to manage the load imbalance among the processes computing the LU factorization of a large dense matrix stored in the Block Low-Rank (BLR) format. Next I proposed a number of optimizations and implemented them in PaRSEC's Dynamic Task Discovery (DTD) model, including user-level graph trimming and direct Application Programming Interface (API) calls to perform data broadcast operation to further extend the limit of STF model. On the other hand, the Parameterized Task Graph (PTG) approach in PaRSEC is the most scalable approach for many different applications, which I then explored the possibility of combining

both the algorithmic approach of Communication-Avoiding (CA) and the communication-computation overlapping benefits provided by runtime systems using 2D five-point stencil as the test case. This broad programming models evaluation and extension work highlighted the abilities of task-based runtime system in achieving scalable performance and portability on contemporary heterogeneous HPC systems. Finally, I summarized the profiling capability of PaRSEC runtime system, and demonstrated with a use case its important role in the performance bottleneck identification leading to optimizations.

Table of Contents

1 Introduction **1**

1.1 Motivation . . . 1

1.2 Contributions . . . 3

1.2.1 Programming Models Evaluations . . . 3

1.2.2 STF Improvements and Limitations . . . 4

1.2.3 Communication Avoiding with PTG for Sparse Algorithms . . . 5

1.2.4 Profiling Analysis for Performance Tuning . . . 5

1.3 Dissertation Outline . . . 6

2 Background and Literature Review of Related Work **7**

2.1 Current programming Models . . . 7

2.1.1 Distributed Memory Programming Models . . . 7

2.1.2 Shared Memory Programming Models . . . 8

2.2 Task-based Runtime Systems . . . 10

2.2.1 PaRSEC Runtime System . . . 11

2.2.2 Other Runtime Systems . . . 13

2.3 Numerical Linear Algebra . . . 16

3 Parallel Programming Models Evaluation **19**

3.1 Overview . . . 19

3.2 Related Work . . . 22

3.3 Block Low-Rank Factorization Algorithm . . . 23

3.4 Required Features . . . 26
3.5 Implementation with the Programming Models . . . 27
3.5.1 Flat MPI Programming Model . . . 28
3.5.2 Flat MPI with Charm++/AMPI . . . 29
3.5.3 OpenMP Task Programming Model . . . 29
3.5.4 PaRSEC DTD . . . 32
3.5.5 PaRSEC PTG . . . 35
3.6 Performance Evaluation . . . 38
3.6.1 Experimental Setup . . . 38
3.6.2 Experiment Results . . . 40
3.7 Conclusions . . . 45

4 Sequential Task Flow Runtime Model Improvements and Limitations 48
4.1 Overview . . . 48
4.2 User Graph Trimming and Broadcast Operations . . . 50
4.2.1 DTD Model . . . 50
4.2.2 PaRSEC DTD Tasks and Communications Tracking . . . 51
4.2.3 Graph Trimming . . . 53
4.2.4 Broadcast Operation . . . 53
4.3 Evaluation with the Cholesky and QR Factorizations . . . 56
4.3.1 Modifications to the user code . . . 58
4.3.2 Qualitative Analysis . . . 61
4.4 Performance Results and Analysis . . . 62
4.4.1 Description of HPC systems . . . 62
4.4.2 Broadcast Benchmark Performance . . . 62
4.4.3 Experiment performances . . . 64
4.5 Conclusions . . . 69

5 Extension to PTG - Testcase with Communication Avoiding 2D Stencils 70
5.1 Overview . . . 70
5.2 Related Work . . . 72
5.3 Background . . . 73
5.3.1 Stencil Problem Description . . . 73
5.3.2 Communication Avoiding Approach . . . 75
5.4 Implementations . . . 75
5.4.1 Standard Implementation with PETSc . . . 75
5.4.2 Task-based Implementation in PaRSEC . . . 77
5.5 Experiments Results . . . 79
5.5.1 Experimental Setup . . . 79
5.5.2 Network and Memory Bandwidth Benchmark . . . 81
5.5.3 Tuning of Tile Size for PaRSEC Performance . . . 81
5.5.4 Comparing Strong Scaling Performance . . . 83
5.5.5 Tuning of Kernel Time and Performance Impact of Communication Avoiding Scheme . . . 85
5.5.6 PaRSEC Profiling of the Two Versions . . . 85
5.6 Conclusions . . . 88

6 Profiling Analysis for Performance Tuning 91
6.1 Overview . . . 91
6.2 Related Work . . . 92
6.3 Background . . . 94
6.3.1 TLR Cholesky Factorization Basics . . . 94
6.4 Performance Tools . . . 95
6.4.1 Trace Collection Framework . . . 97
6.4.2 PINS: PaRSEC INStrumentation . . . 98
6.4.3 Dependency Analysis . . . 99
6.4.4 Trace Conversion Tools . . . 99

6.5 TLR Cholesky Case Analysis 101
6.6 Conclusions 105

7 Conclusions and Future Work **106**
7.1 Conclusions 106
7.2 Future Work 109

Bibliography **111**

Vita **128**

List of Tables

5.1 STREAM Benchmark Results (MB/s) for NaCl and Stampede2. 82

List of Figures

2.1 Bulk Synchronous Parallel model for parallel execution 9
2.2 Diagram of the main components of PaRSEC runtime system. 12
2.3 The four different kernels from Cholesky and QR respectively, during the 2nd iteration of kernel executions. 17

3.1 Low-rank matrix factorization and compression algorithms. 25
3.2 Illustration of algorithm updating a low-rank block. 25
3.3 OpenMP task implementation of BLR factorization 31
3.4 PaRSEC DTD implementation of BLR factorization, including insertion of the tasks in sequential order with the data usage information provided. . . . 33
3.5 PaRSEC PTG specification of the diagonal factorization tasks: defining the parameter space, data locality, and data dependencies, written in JDF. . . . 36
3.6 Initial block ranks for each test matrix, all have dense tiles near diagonal, but different off-diagonal low rank patterns 39
3.7 Test matrices information 39
3.8 The average wait time of a MPI process in a collective call for the flat MPI model, shown as percentage of total execution time. Minimum and maximum shown as well 41

3.9 Execution time of each model on different datasets, top) 338ts, middle) human_4x4, bottom) 1ms. Flat MPI performances on 1 node (28 cores)/4 nodes are used as base to show speed up of the models. They are 317, 221 and 1050 seconds respectively. Both X- and Y-axis are plotted on log2 scale. PTG speed up over MPI+OMP are 1.23, 1.24 and 1.40 at 16 nodes 43
3.10 Execution stream of the different sections for one selected process, top) Flat MPI, bottom) AMPI. Most of the BCastPanel time are likely idle time . . . 43
3.11 Top) Computation kernels occupancy summary of all the threads, Bottom) Detail breakdown of the diagonal factorization task for MPI+OpenMP model 46

4.1 Top: original DTD, each task has a unique key Bottom: send/recv level key. Grey square represents local task, white square represents remote task. Circle represents the remote_deps structure. In the new scheme, data flow ID is a combination of sender rank and sequence number to uniquely label each data transfer. As long as both the sender and the receiver has the dependent tasks inserted, the data ID will be assigned correctly for the two sides to match the data transferred. 52
4.2 Two-step broadcast with meta-data transfer as the first, and data payload transfer as the second. They propagate as two separate flows but data reception call can only be matched when the meta-data is received and global ID is known. 55
4.3 The four different kernels from Cholesky and QR respectively. Both runs on a 2X3 compute grid with 2-D block cyclic distribution. For QR, a super-tiling of 2 is used on the grid row to reduce cross node P2P communication. 57
4.4 Left, trimmed task graph without broadcast call; Right, explicit broadcast call to propagate POTRF data. Color scheme and data distribution follows that from Figure 4.3. Lighter red and purple represent remote tasks, yellow represents broadcast task. Data dependency between TRSM and GEMM omitted. 60

4.5 Since only the TSMQR tasks are of order $O(N^3)$, we can insert all the other tasks in all the nodes while inserting TSMQR only on ranks that are in the same row or column of the current panel tasks. Figure on the left, shows the situation for tasks inserted on rank 1, while figure on the right is for tasks on rank 4. 60
4.6 Benchmark of a broadcast operation for sending a square tile of double precision floating point values. I tested on two sets of nodes, and varied the message data size. For comparison, I have the default DTD P2P, the proposed DTD broadcast and finally the broadcast utilized in PTG (the two shared the same mechanism). 63
4.7 Performance on Shaheen II, 256 nodes. Left: Cholesky, Right: QR 65
4.8 Performance on Shaheen II, 512 nodes. Left: Cholesky, Right: QR 65
4.9 Performance on Fugaku, 256 nodes. Left: Cholesky, Right: QR 68
4.10 Performance on Fugaku, 512 nodes. Left: Cholesky, Right: QR 68

5.1 Common illustration of the Jacobi update scheme [44]. 74
5.2 The 2D five-point stencil operation using PA1 algorithm on a 10-by-10 grid, having a step size of 3 as illustrated in the original report [Demmel et al.]. For a single processor with the projected view. Red asterisks indicate remote values that need to be communicated. 76
5.3 Top) Diagram of the baseline version of the PaRSEC implementation. Three possible task locations and their data dependencies are shown. Black line indicates within node data copy while red line indicates remote communication., Bottom) Diagram of the communication avoiding version PaRSEC implementation. Three possible task locations and their data dependencies are shown. Black line indicates within node data copy while red line indicates remote communication. The boundary tiles will have a bigger ghost region to accommodate the extra layers of remote data. 78

5.4 Network Performance from NetPIPE on NaCl and Stampede2 with theoretical peak of 32Gb/s and 100 Gb/s, respectively. 82
5.5 Shared memory PaRSEC base version performance for a given tile size; (top) NaCl with problem size 20K, (bottom) Stampede2 with problem size 27K. . 84
5.6 Strong scaling speed up over single node baseline PaRSEC; (top) NaCl result with problem size 23k, tile size 288; (bottom) Stampede2 result with problem size 55k, tile size 864, running for 100 iterations. Steps size of 15 is used for CA version. 84
5.7 Tuned kernel performance: (top) NaCl result with problem size 23k, tile size 288; (bottom) Stampede2 result with problem size 55k, tile size 864, running for 100 iterations. Steps size of 15 is used for CA version. Running on 4, 16 and 64 nodes with squared compute grid. The ratio r indicates the ratio of m_b and n_b of tile being operated on, namely r^2 of the original number of points in a tile. Black lines indicate the base PaRSEC with original kernels' result. 86
5.8 Tuned step size performance: (top) NaCl results with problem size 23k, tile size 288; (bottom) Stampede2 results with problem size 55k, tile size 864, running for 100 iterations. Step sizes of 5, 15, 25 and 40 are used. 87
5.9 One node's profiling result, running on NaCl with 16 nodes, tuned ratio of 0.4, 11 computation threads on a node. (top) baseline PaRSEC; (bottom) CA PaRSEC. The boundary indicates the tiles that need to exchange data with remote nodes. 89

6.1 Left, TLR format for matrix A having 4-by-4 tiles of size nb-by-nb. Diagonal tiles are stored as dense. Off-diagonal tiles, are compressed to have U and V blocks, each has its own rank, k. Right, the corresponding DAG for TLR POTRF of the matrix. 96
6.2 Example DOT file entries. 103
6.3 Example HDF5 file entries. 103

6.4 Time between data is ready and TRSM starts for st-2D-sqexp synthetic kernel data. Left, without lookahead; right, with lookahead of 5; each point represents one TRSM; matrix has 100×100 tiles. 104

Chapter 1

Introduction

1.1 Motivation

With the latest release of the TOP500 List from June 2022[1], we have officially entered in the exascale era with the top machine, Frontier, located at the Oak Ridge National Laboratory, and supported in part by the Exascale Computing Project (ECP) [71]. The driving force behind the need of such a powerful machine is the pursuit to enable unprecedented scientific discovery. The exascale application areas range from chemistry, materials, energy, earth, and data science. Efficiently running these applications on the fastest machines means faster scientific discovery and solving problems that were previously intractable. This requires the software supporting the applications to be able to use the hardware efficiently, and that can scale to the entire system. Since the top systems can either be CPU-based or heterogeneous with NVIDIA or AMD GPUs or other types of hardware accelerators, their programming model needs to be able to extract performance from the different hardware on a given node, and ensure optimal inter-node communication as well.

Currently, most of the applications adopt the MPI+X approach with MPI [100] being the dominant library for cross-node communication and supporting portability across systems and hardware, and where the "X" can be any of POSIX Threads, OpenMP [89], Kokkos [52],

[1] https://www.top500.org/lists/top500/2022/06/

RAJAs [78], CUDA or other programming languages and library-based software. Porting the applications from MPI-based and CPU-based to their MPI+X counterparts might be the most straightforward approach, usually only requiring to convert the performance-critical kernels to the new accelerators using one of these node level abstractions. But to achieve good efficiency and scalability, performance tuning such as overlapping communication and computation is required. And this has to be tuned across the different leadership machines. As a result, many applications decide to build on the common libraries, e.g., AMReX [120], so that the burden of providing good performance across these complex systems can be shifted to the experts developing these libraries.

Task-based runtime systems serving as an alternative approach has been getting traction in the recent years, both as a way to utilize heterogeneous system efficiently but also increase productivity. It aims to separate the expression of algorithm from the performance optimization so that the domain scientists can focus on the scientific problems, while the runtime is in charge of efficient hardware utilization and message communication. There are many ways to express the algorithm to the runtime system, but essentially the runtime system tracks a directed acyclic graph (DAG) of tasks with data dependencies among them. With concurrent scheduling of computational tasks and the data transfers between them, this programming model naturally achieves computation and communication overlapping and is likely to result in performance portability. And there are roughly two ways to express the task graph: 1) explicit parallel program, where the data dependencies among the tasks are known – this usually requires the assistance of compilers; and 2) implicit parallel tasking that depends on dynamic dependence analysis to generate the task graph.

In this dissertation, I evaluated both the programmability as well as the performance of some of the leading programming models. I implemented two optimizations for the Dynamic Task Discovery (DTD) interface in PaRSEC [74], which adopts the common task-based runtime system interface called Sequential Task Flow (STF). The results demonstrated the benefits and limitations of the STF approach. With the strengths of Parameterized Task Graph (PTG) [42] interface within the PaRSEC framework [29], I evaluated the possibility of

building sparse iterative operations with communication avoiding techniques with PaRSEC to achieve further performance improvements. The importance of the profiling system for performance optimization is demonstrated as well.

1.2 Contributions

This dissertation contributes along several aspects of distributed programming models: from evaluating the programming styles and performance of different models, to optimizations of the STF model, in particular. I also explored the potential of combining communication avoiding technique with task-based runtime system in case of sparse solvers. Finally, I highlighted the key role of the profiling subsystem for performance optimization, and the flexible scripting approach that I adopted to pinpoint execution bottlenecks.

1.2.1 Programming Models Evaluations

Blocked low-rank LU is an efficient approach to perform matrix factorization by exploiting the numerical properties of matrices to compress their off-diagonal blocks. At the same time, from the parallel programming perspective, it is a challenging algorithm, because the resulting blocks have varing sizes, which creates both computational and communication imbalance. This in turn requires the programming model to be able to efficiently handle such situations at runtime. Using it as a test case, I set out to evaluate the performance and the productivity aspects of different parallel programming models. Starting from an existing MPI and MPI+OpenMP implementations, I converted and optimized the algorithm for task-based programming models and provided two implementations: the PaRSEC parameterized task graph (PTG) and the Dynamic Task Discovery (DTD insert_task) implementations. This process included profiling and performance analysis of the algorithm in a heterogeneous large-scale setup, while identifying performance bottlenecks as well as pinpointing and implementing parallel programming constructs critical for performance and scalability, e.g., the ability to send variable-sized messages occurring due to the numerical rank differences,

which was a new feature that I added into the interface. The novelty of this work is two-fold: 1) I provided the first efficient implementation of block low-rank LU factorization using a runtime system, and 2) I identified and quantified critical constructs in task-based parallel programming paradigms for performance and scalability. My conclusion highlights the fact that the sequential task flow (STF) model—the base model behind most MPI+X programming paradigms—can provide good performance and portability across machines, but it requires further optimizations to remain scalable and efficient.

1.2.2 STF Improvements and Limitations

Sequential task insertion model has been widely adopted and proven to be a user-friendly approach due to its ease of use. However, the overhead of building the graph of dependencies between tasks originates from the global knowledge of the distributed execution, and the lack of collective communications is highly detrimental to the performance and scalability of the programming model. To investigate different strategies for task dependencies' graph construction and their inherent cost and scalability, I modified the PaRSEC runtime, more specifically the DTD's Domain Specific Language (DSL), to add two new features: 1) graph trimming, and 2) collective communications. Graph trimming, or the ability to have a functional algorithm based on carefully-built local information, requires the availability of correct data dependencies between tasks being expressed, allowing the runtime to track the task dependencies without uniquely naming each task. Collective operations in sequential task insertion model is a novel concept. Unlike the MPI model, for which the communication group is known beforehand, the participants in task-based runtime have only partial knowledge of the collective operation. As a result, the participants need to rebuild the collective locally as the operation unfolds. Performance results from the Cholesky and QR factorizations showed the benefits and limitations of the adding these features at the runtime level. The goal was to reduce the programming complexity while maintaining the efficiency and flexibility of the parallel concepts and lower the bar and thus enable a wider adoption of distributed task insertion models.

1.2.3 Communication Avoiding with PTG for Sparse Algorithms

Stencil computations or general sparse matrix-vector products are key components in many scientific algorithms, but their low arithmetic intensity means that the memory bandwidth and network latency are the main performance limiting factors. Communication avoiding (CA) scheme aims to minimize the influence of the network latency in repeated sparse matrix-vector multiplications by replicating remote work in order to delay the communication that resides on the critical path. Although CA is a promising numerical technique, it has a very challenging implementation aspects, especially in the runtime system. Focusing on minimizing the communication bottleneck in distributed stencil computation, I combined CA scheme with the computation and communication overlapping that is inherent in a dataflow task-based runtime system such as PaRSEC to demonstrate their combined benefits. I implemented a version of the 5-point stencil workload in PaRSEC, that showed significant performance and scalability benefits (it was up to 57% faster than the second best implementation).

1.2.4 Profiling Analysis for Performance Tuning

Profiling is an essential part of evaluating the performance of a parallel application. For the MPI+X applications that mostly follow the Bulk-Synchronous Parallel (BSP) model, space-time plots are used to identify performance issues. Task-based runtime on the other hand explores the task graph dynamically, with tasks' execution and message communication all happening concurrently. Its profiling system needs to be able to collect data to understand task execution sequences from scheduling decisions, message transfers rate for network utilization, kernel execution time and data allocation time for memory and hardware utilization. Modern data analysis workflow (e.g. R, Python modules) is a flexible and ideal choice to analyze and visualize the many aspects of the execution trace and to help pinpoint performance bottlenecks. Using Tiled low rank (TLR) Cholesky as an example, where the diagonal tasks are more critical than in the dense case, I demonstrated this flexible approach's

ability to combine information from multiple sources in a novel way to identify the scheduling deviation from the critical path, which lead to the subsequent performance optimization.

1.3 Dissertation Outline

The rest of this dissertation is organized as follows:

- Chapter 2 provides the background on both the current HPC programming models and current task-based runtime systems developments. Also I show applications that adopted task-based approach.
- Chapter 3 uses the block low rank (BLR) LU as a test program to evaluate the programmability and performance of several programming models.
- Chapter 4 focuses on improving the STF model with graph trimming and collective operations as well as the limitations of the STF model.
- Chapter 5 adopts the PTG interface and extends it with the communication avoiding approach for sparse problems, using 5-point stencil in 2D as the test case.
- Chapter 6 provides the details of the PaRSEC profiling subsystem, and shares a case study focused on understanding the runtime execution by analyzing the outputs.
- Chapter 7 concludes the dissertation and outlines the future directions.

Chapter 2

Background and Literature Review of Related Work

This chapter covers some of the common programming models used in HPC applications, including both for distributed as well as for shared-memory systems. I then briefly summarize the actively developed task-based runtime systems. Since my evaluation programs are in the field of numerical linear algebra, I will also cover some recent developments in this area as well.

2.1 Current programming Models

2.1.1 Distributed Memory Programming Models

Message Passing Interface (MPI) [100] standard is the dominant programming model used for inter-node communication in high-performance computing (HPC). And it is widely expected that MPI will continue to serve that purpose of inter-node communication on the future HPC systems for most of the applications. MPI itself is a relatively low level model with APIs for point-to-point message passing, collective operations (blocking and non-blocking), synchronizations as well as one-sided operations. The application code is written in a single program, multiple data (SPMD) form where the algorithm writer needs

to manage the communications explicitly. And applications written with MPI are very likely to follow bulk-synchronous parallel (BSP) model [112, 111], whereby a sequence of parallel execution is followed with a (often global) synchronization to ensure consistency in data and program state as shown in Figure 2.1. There are many different implementations of the MPI standard specification either from hardware vendors or open-source community such as OpenMPI [62], MPICH [124], This ensures portability across underlying hardware, and constant optimizations of the provided functionalities for resilience, robustness, scalability and low-latency communication. As a result, it is used as the underlying communication layer for many of the distributed task-based runtime systems.

Partitioned Global Address Space (PGAS) programming model, such as Chapel [38], UPC++ [123] and OpenSHMEM [40], is another paradigm used by applications on distributed memory systems. In this model, a global memory address space abstraction is logically partitioned, where a portion is local to each process. If implemented as a software library, the application writer needs to manage the message communication and synchronizations explicitly, in a similar fashion to MPI codes, but with integration in a programming language, the PGAS compilers can handle some of the communication details and achieve higher productivity for the programmer by avoiding common coding mistakes.

2.1.2 Shared Memory Programming Models

For the MPI+X approach, the X usually represents a programming model working in shared memory space, either for CPU or hardware accelerators or both. In order to achieve performance and portability for intra-node programming, there are a variety of options. Among portable, and standard-backed programming models, OpenMP [89] is likely the most widely used one and is supported by hardware vendors on their platforms but the quality of the implementation might vary. Through parallel loops or tasking directives multiple threads can be executed concurrently, and target directive offers support for accelerator offloading.

Kokkos [52] and RAJA [78] provide another alternatives for portable, heterogeneous-node programming via C++ abstractions. They are designed to work on complex node architectures

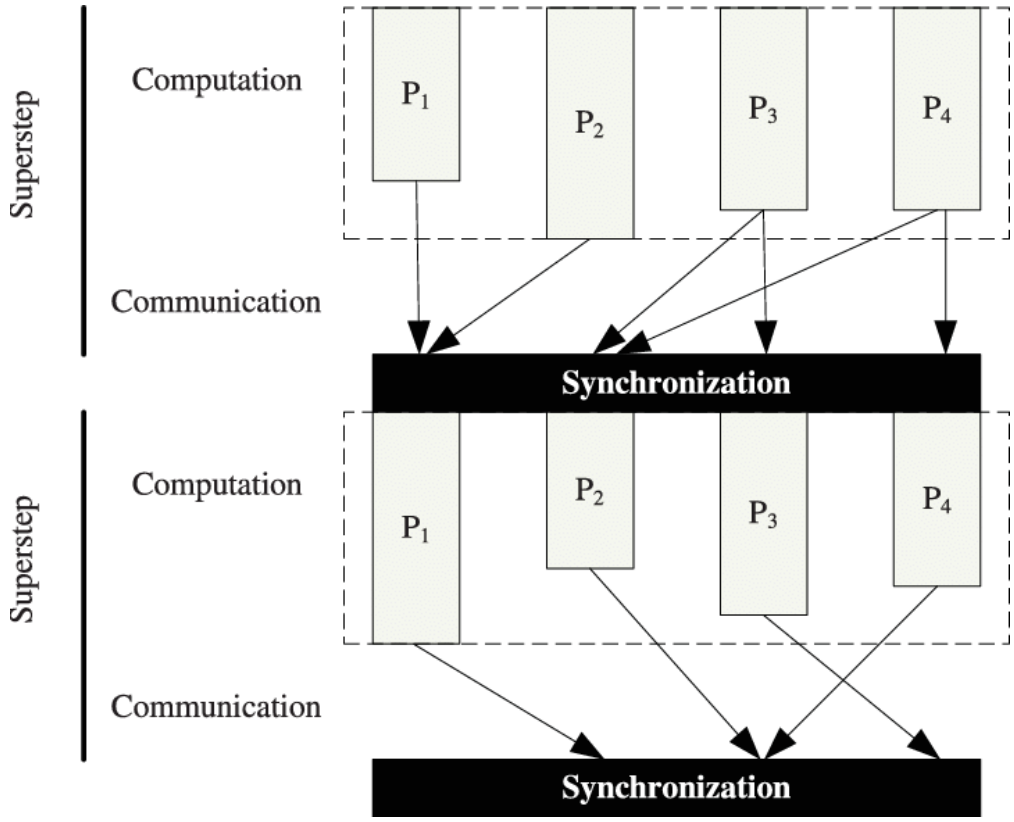


Figure 2.1: Bulk Synchronous Parallel model for parallel execution

with multiple types of execution resources and multi-level cache and memory hierarchies. Many ECP applications have successfully used Kokkos or RAJA to write portable parallel code that runs efficiently on GPUs [97]. Some vendor-supported options include CUDA and OpenACC for NVIDIA GPUs, SYCL/DPC++ for Intel GPUs, and HIP for AMD GPUs. OpenACC supports accelerator programming via compiler directives across many hardware platforms. And SYCL provides a C++ abstraction on top of the OpenCL standard, which itself is a portable alternative to CUDA, lower-level API for programming heterogeneous devices. Intel's DPC++ builds on SYCL by adding productivity extensions. HIP from AMD is an API and set of libraries similar to CUDA software stack. All of these models will need to work with MPI, resulting in a two-level programming paradigm with a relaxed synchronization requirement, especially at the node level. Still, combining these two levels to work efficiently is a challenge in itself [90].

2.2 Task-based Runtime Systems

Task-based runtime systems usually follow the dataflow programming model, where instead of writing a program as a sequence of local instructions and concurrency controls, we treat the computation on data as an indivisible task. The input data for this task, and the output from the task connect to other tasks in order to form the flow of application data and specify the dependencies among the tasks. A directed acyclic graph (DAG) based on that dataflow can be scheduled onto the HPC system with the maximum level of concurrency and minimum level of synchronization, with task-based runtime system acting as a middle layer on top of multi-threading, accelerators' management, and MPI communication. They execute tasks in an asynchronous fashion and break out from the overly constraining bulk-synchronous programming model. These runtime systems target shared and distributed-memory systems, possibly equipped with GPU accelerators.

2.2.1 PaRSEC Runtime System

PaRSEC [30] (Parallel Runtime Scheduling and Execution Controller) is a generic framework for architecture-aware scheduling and management of micro-tasks on distributed many-core heterogeneous architectures whose components are shown in Figure 2.2. The runtime contains multiple components: programming interface in form of DSLs, schedulers, communication engines, data interfaces, and a few other modules. PaRSEC uses a modular component architecture, allowing different modules to be selected, providing different capabilities to different instances of the runtime (such as scheduling policies, or support for various accelerators). A clear API for these modules allows interested developers or users to implement their own application-specific policies. The core components include all the management required for abstracting a computing resource to an application developer and is shared by all the interface entry points. At the top, the programming interface specifies the functionality that PaRSEC provides for developers to express their applications. Currently, there are 3 different interface options supported by PaRSEC: parameterized task graph (PTG) [42], Dynamic Task Discovery (DTD) [74], and more recently Template Task Graph (TTG) [32].

These DSLs create a dataflow model with dependencies between tasks and exploit the available parallelism present in applications and expressed to PaRSEC. The first historically and most commonly used DSL is PTG, which allows users to define a parameterized task graph with syntax known as Job Data Flow (JDF) that specifies the dependencies between user tasks. As a DSL coder, the user needs to specify the possible task classes, including the body of each task performing the computational work, the data usage and how they flow among the various task classes, and finally the affinity of each task, which indicates the rank to execute a given task. The possible tasks are identified by their name and the parameters provided to each of the task classes. In this formulation, the task graph is specified a priori and as a result each node has a global view of the task graph. Due to its flexibility, many applications have obtained state-of-the-art performance and scaling results [36] [35] [34].

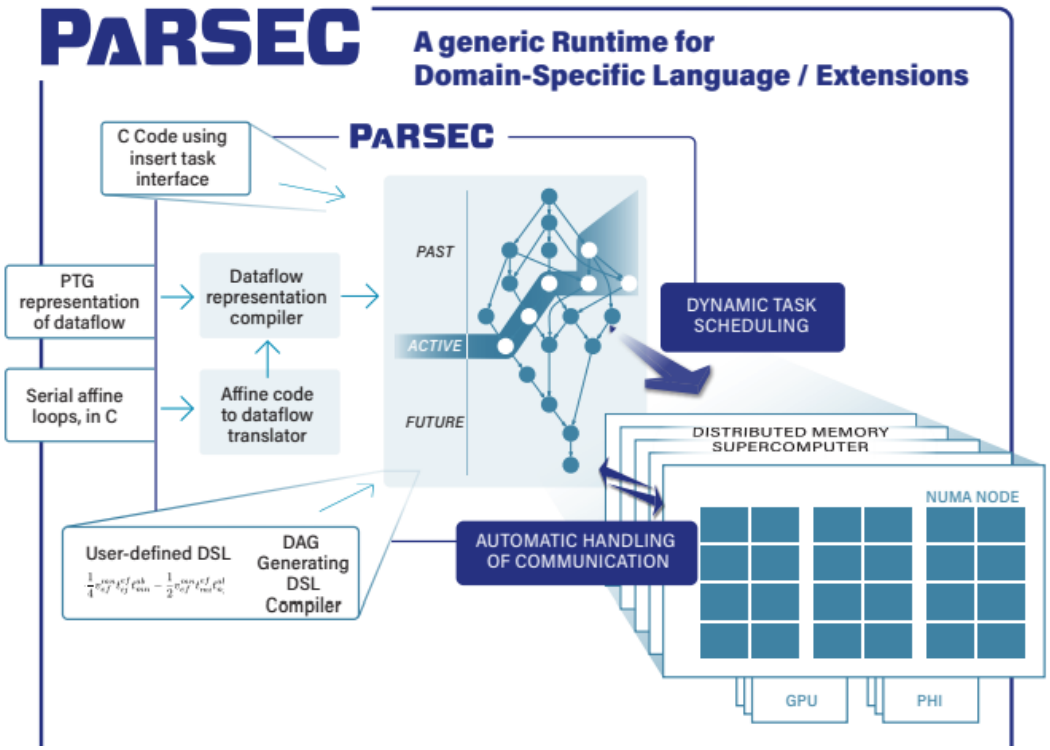


Figure 2.2: Diagram of the main components of PaRSEC runtime system.

To enhance the productivity of the application developers, DTD interface adopts the STF model and provides an API-based implicit parallelism model. Users can write sequential code and DTD will build the task graph internally. When inserting tasks, a function is provided as the code body of the task, and the data used with its usage information (Read, Write, Read-Write) are passed to DTD for analysis. The correctness is guaranteed by the sequentiality of the data accesses. In distributed-memory systems, all the processes will iterate over the same task discovery process to ensure consistency among the processes without communication and without a priori task graph specification. The scalability of this model is a known issue and several previous research efforts proposed partial solutions to remedy it [4] [50].

The newest interface—TTG—provides a C++ API and extends the idea of PTG by generalizing the notion of parameters to arbitrary types and enabling data-driven selection of task dependencies including conditional execution. This is a critical feature for algorithms with irregular data access patterns since many of them are dependent on the intermediate data. In a manner similar to PTG, in TTG the user also needs to specify all the possible task templates, and the connections between the task templates via data-communicating edges to the different terminals. But with task graph being data dependent, the program specifies a set of possible DAGs of tasks with the actual executed DAG being dependent on the data flowing through it.

2.2.2 Other Runtime Systems

In recent years, there were many new and now actively developed runtime systems. I will summarize the features of the representative ones for both shared memory and distributed memory systems.

Legion

Legion [25] introduces a concept of logical regions to virtually represent partitioned real data to infer task dependencies. For each task using any logical regions, users provide coherence and privilege information, and Legion extracts the parallelism inherent in the presented

regions and depending on the provided constraints. Communication among remote nodes is not required to be expressed explicitly as it is managed by Legion implicitly. It uses GASNet for inter-node communication and has support for heterogeneous architectures. Legion showed good performance on both benchmarks and applications [104] [24]. Legion provides a strong and flexible interface to decompose any data into logical regions to help in porting applications that exhibit different behavioral parallel patterns. It provides a DSL called Regent [98] to describe the task graph and at a lower level, and the runtime called Realm [106], which is in charge of scheduling the tasks.

StarPU

StarPU [7] is a high-level runtime for both shared and distributed system. There are two approaches for distributed system: (1) allow users to specify communication explicitly, and (2) infer communication from dataflow implicitly. StarPU has its own data interface to manage data movement and versioning. For inter-node communication it uses MPI and has support for heterogeneous architectures. Multiple applications in areas like dense and sparse linear algebra kernels have showed performance improvement when implemented with StarPU runtime.

It also adopts the STF programming model allowing users to insert tasks. The runtime infers the dependencies to order tasks' execution based on the information provided by the users regarding the data produced or consumed by each task. StarPU builds the task graph dynamically during runtime in a manner similar to PaRSEC's DTD. For distributed memory, a global view of the task graph is maintained to connect the local and remote tasks. The global view can be pruned but needs user input. There is no support for building a static task graph like PaRSEC PTG.

Charm++

Charm++ [82] is a parallel programming framework supported by an adaptive runtime system that builds on three main concepts: 1) over-decomposition, whereby the work and data are

decomposed to many more entities than the total number of physically available processing elements; 2) asynchronous message-driven execution, whereby a "process", or chare in the Charm++ parlance, never wastes the physical resources while waiting on communication's completion, by allowing other "processes" to take over the physical cores and continue the progress on their own work; and 3) migratability, whereby the data and work can move among the processing elements. Combining these three features provides the potential to dynamically balance the load and hide the communication overheads. It is a well established framework with success stores based on results in many different applications.

OmpSs and OmpSs-2

OmpSs [51] is a programming model composed of a set of directives and library routines that can be used in conjunction with a high-level programming language in order to develop parallel applications. It uses Nanos++ runtime to manage ordering of tasks. Their initial proposals for task-based directives were a primary driver for the inclusion of advanced concepts that allows task-based parallelism in OpenMP. OmpSs uses a different execution model from that of OpenMP and does not implement fork-join parallelism as OpenMP does. OmpSs supports heterogeneous memory systems through leveraging native kernel implementations provided by the user.

Taskflow

Taskflow [79] was motivated by the lack of advances in computer-aided design (CAD) tools with heterogeneous parallelism to achieve better performance and productivity. Unlike traditional loop-parallel scientific computing problems, many CAD algorithms exhibit irregular computational patterns and complex control flow that require strategic task graph decomposition to benefit from heterogeneous parallelism. This type of complex parallel algorithms are difficult to implement and execute efficiently. Taskflow provides an expressive task graph programming model by leveraging modern C++ lambda closures. It supports a new conditional tasking model that supports in-graph control flow beyond the capability of

traditional DAG models that prevail in existing systems. Conditional tasks enable developers to integrate control-flow decisions, such as conditional dependencies, cyclic execution, and non-deterministic flows into the task graph of end-to-end parallelism. For applications that frequently exhibit dynamic behavior, such as optimization with branch-and-bound methods, programmers can efficiently overlap tasks both inside and outside of the control flow to hide expensive control-flow costs. Their scheduling algorithm prevents the graph execution from underutilized threads that is harmful to performance, while avoiding excessive waste of thread resources when available tasks are scarce. This improves the overall system performance, including latency hiding, limiting energy usage, and increasing task throughput.

2.3 Numerical Linear Algebra

Numerical linear algebra algorithms are the computational foundations of many scientific computing applications. Improving the performance of these algorithms can reduce simulation time and advance the scientific discovery. A family of tiled matrix algorithms were developed specifically in response to the rising number of processing elements found in today's computer systems. Tiled algorithms are structured in such a way that concurrency is expressed at the algorithmic level. Therefore, a matrix $A \in \mathbf{R}^{n \times m}$ with $n \times m$ elements is divided into tiles of size $n_t \times m_t$ with operations being applied to the individual tiles as shown in Figure 2.3. At their core, many of these algorithms are structured such that, in each iteration, the tile on the matrix' diagonal is updated, followed by an update of a tile row or column and the update of the tiles in the trailing part of the matrix. By encapsulating the operations on the tiles in tasks and specifying the data-flow between them, a large amount of concurrency can be exposed, leading to minimized idle times in the threads executing the tasks. In many cases, the communication pattern is an irregular many-to-many exchange of tiles, which is evolving throughout the course of the execution of the algorithm. The required coordination of the execution of tasks at a global scale is challenging due to the sheer amount of tasks and the irregular communication pattern. These properties make this class of algorithms an interesting target for task-based programming models.

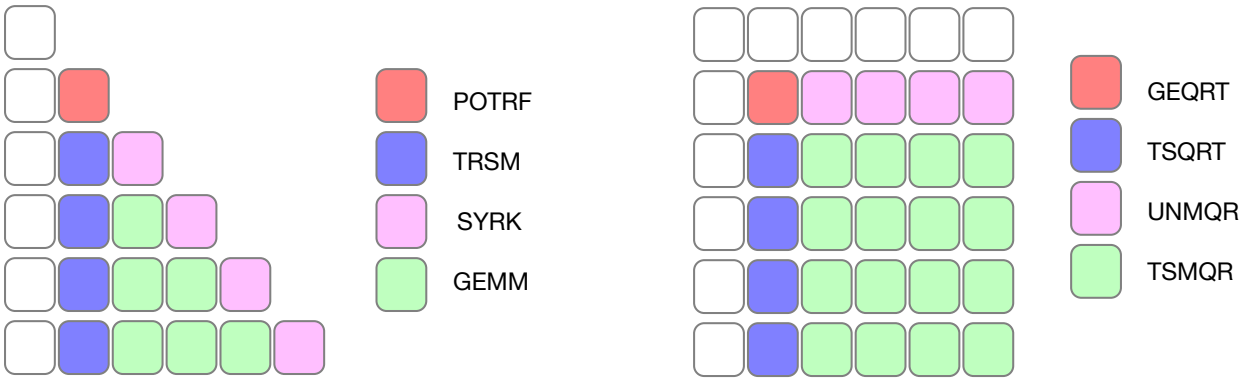


Figure 2.3: The four different kernels from Cholesky and QR respectively, during the 2nd iteration of kernel executions.

Numerous libraries provide dense linear algebra routines. Since its initial release nearly 30 years ago, LAPACK [15] has become the de facto standard library for dense linear algebra on a single node. It leverages vendor-optimized BLAS for node-level performance, including shared-memory parallelism. ScaLAPACK [27] was built upon LAPACK by extending its routines to distributed memory computing with message passing and by relying on both the Parallel BLAS (PBLAS) and explicit distributed-memory SPMD parallelism and synchronization. Some attempts have been made to adapt the ScaLAPACK library for accelerators, but these efforts have shown the need for a new framework. More recently, the DPLASMA [28] and Chameleon libraries [1] both build a task dependency graph and launch tasks as their dependencies are fulfilled. This eliminates the artificial synchronizations inherent in ScaLAPACK's design, and allows for overlap of communication and computation. DPLASMA relies on PaRSEC's PTG or DTD DSLs to specify and schedule tasks, while Chameleon [1] can use either StarPU or PaRSEC runtime. And they both support GPU-based task executions. SLATE [58] is a recent effort to implement the linear algebra routines in the distributed memory settings with the goal of fully replacing ScaLAPACK's functionality and adding new algorithms. It uses modern C++ framework and MPI+OpenMP model, with support for modern accelerated architectures.

Sparse solvers are entirely different from the dense ones, but they are also a critical part of many numerical simulations. Usually, no scalable sparse solvers can work for all applications, nor are there single implementations that work well for all problem sizes. As a result, the mostly widely used packages for iterative methods' solvers and optimization, including PETSc [20] and Trilinos [Trilinos Project Team], provide a wide variety of algorithms and implementations that can be further customized. In terms of direct solvers that were built on top of task-based runtime systems, both PaStiX [70] and MUMPS [5] are two of prominent efforts.

Chapter 3

Parallel Programming Models Evaluation

3.1 Overview

Scientific simulations from many domains utilize high-performance computers to run in parallel their workloads and speed up obtaining the results and thus contribute to knowledge discovery. Traditionally, these applications are implemented with the MPI only model coupled in the vast majority of cases with a static data distribution. A static mesh partitioning or domain decomposition methods could lead to imbalanced workloads, especially when the workload can change dynamically. Moreover, the explicit synchronization introduced in the MPI programming model invariably results in significant idle time under dynamically imbalanced workloads.

The computational and storage costs of the dense matrix operations can be reduced significantly using a low-rank format that exploits so called data sparsity that relies on low numerical rank of the off-diagonal submatrices that physically represent far-range interactions that tend to have favorable eigen-spectrum and admit reduced-size numerical approximations. More precisely, Block Low-Rank (BLR) partitions the matrix in 2-D blocks and compresses the off-diagonal blocks using their low-rank representations, leading to a

smaller need for storage space and a lower computational intensity. Thus, the use of a low-rank format can drastically shorten the factorization time, a highly desirable property for critical algorithms for as long as the error can be bound a priori. Solutions of a large-scale, diagonally dominant dense linear system of equations is needed for a number of scientific and engineering simulations, and BLR format enables simulations at larger scale, which would not have been practical using the classic dense storage format, either due to the storage or to the excessive computational costs.

One such application is the LU factorization of a dense matrix stored in the BLR format [11]. We observed that the geometry-based matrix partitioning compresses the matrix favorably, leading to many off-diagonal blocks with small or fast decaying numerical ranks, and therefore this translates into a lower computational cost. In a 2-D block-cyclic dense distribution, data is mostly evenly distributed across participating processes, leading to well-balanced—both in terms of memory and computation—factorizations [46]. However, the compressed format does not inherit the even balancing of the classic dense algorithm, leading to an algorithm that, while similar to the dense counterpart, is unbalanced and dynamic in memory needs, communication, and computation. An implementation of this algorithm using MPI only exacerbates this imbalance due to its tightly coupled nature, where an explicit synchronization is necessary at each factorization step. It also highlighted that the accumulated idle time due to the explicit synchronization at each step of factorization can be significantly greater than the load imbalance in the total local computation time among the processes.

Moreover, the dynamic nature of each block's rank during execution makes it difficult to statically distribute the blocks among the processes to reduce the load imbalance. Alternative, more dynamic, approaches are necessary to cope with the imbalance, and deliver efficient executions in distributed memory environments.

In this chapter, I explore the computer science aspects of this highly dynamic problem, and try to understand how different programming approaches compare when supporting such an imbalanced application. Looking simultaneously at the metric of programmability

and the more objective metric of performance. More precisely, I evaluate five different programming models for implementing the BLR LU factorization of a dense matrix, arising from the boundary element analysis of electrostatic field:

1. The **Flat MPI** (MPI only) model with blocking collective operations, which leads to synchronization at each step of factorization;

2. The Adaptive MPI (**AMPI**) model: an implementation of the MPI standard on top of Charm++ that supports over-decomposition and dynamic load balancing [3];

3. The **MPI+OpenMP** tasking model, where both the computational and communication tasks are dynamically scheduled in order to remove the synchronization points of our flat MPI implementation;

4. The Dynamic Task Discovery (**DTD**) model [75] where the algorithm is described sequentially as a series of tasks and the runtime build the data dependency graph dynamically; and

5. The Parametrized Task Graph (**PTG**) model where the algorithm has a dataflow description as a parameterized graph of tasks.

For the DTD and PTG cases, I use the PaRSEC distributed-memory runtime system [30], that can dynamically move data among processes to satisfy data dependencies and schedule the available tasks.

I evaluate the programmability of each model, commenting on the experience of transitioning from the original flat MPI implementation of BLR LU to task-based programming models. I then analyze in detail the performance, focusing on the effectiveness of each programming model to address the load imbalance, overlap of communication and computation, and, more globally, reduce the factorization time. This work is a guide for parallel application developers, to provide a path to avoid performance pitfalls with the MPI+X programming model, while describing a possible path to alternative programming models. Simultaneously, the data movement patterns and the exposed data

dependencies represent the backbone of a large class of algorithms, and can be used by parallel programming researchers when developing new features on their next-generation programming models.

3.2 Related Work

In addition to the BLR format, several other low-rank formats have been proposed, including $\mathcal{H}$-matrix [66] and Hierarchical Off-Diagonal Low-Rank (HODLR) [10] formats, and their nested variants $\mathcal{H}^2$-matrix [67] and HSS [39] formats. There are also multi-level low-rank formats with the lattice structures [12, 118]. Among those formats, the $\mathcal{H}$-matrix has the most general low-rank format, leading to the near-linear complexity of the factorization. However, its irregular hierarchical block structure poses a challenge when parallelizing the factorization on a distributed-memory computer. To simplify the parallelization and improve the scalability, BLR abandons the hierarchy, but comes with the price of higher storage and computational complexities , i.e., $\mathcal{O}(n^{1.5})$ for storage and $\mathcal{O}(n^2)$ computational complexities for the BLR factorization of a dense matrix of dimension n [13], respectively. This may be compared with $\mathcal{O}(n \log n)$ and $\mathcal{O}(n \log^2 n)$ complexities with the $\mathcal{H}$-matrix format [66], respectively. Nevertheless, for factorizing a small-scale matrix in practice, e.g., $n = \mathcal{O}(10^5)$, the BLR and $\mathcal{H}$-matrix formats often have similar costs of factorization in practice.

The BLR's simpler flat low-rank format brings the potential for higher computational performance. However, for solving a practical problem with an irregular partitioning of the matrix, the parallel scalability of the BLR factorization can be greatly limited by the load imbalance among the processes, even on a small number of processes (e.g., tens or hundreds of processes). It is then the responsibility of the programming paradigm and its model to provide developers with the means to efficiently handle such imbalance, either by shifting it around the participating processes or by overlapping multiple, possibly partially dependent, iterations.

The BLR format was used for distributed multi-frontal sparse factorization [14]. In the Hierarchical Computations on Manycore Architectures (HiCMA) library, the StarPU

runtime [16] was used to improve the performance of the distributed BLR Cholesky factorization [8]. And more recently, HiCMA has been ported to use PaRSEC runtime system, and extensions to include mixed-precisions and sparse tasks graph have been studied [36] [35] [2]. Previously, load balancing issues in generating and performing the matrix vector multiply with the $\mathcal{H}$-matrix have been studied [72]. Compared to matrix generation and multiplication, the factorization has more complex dataflow, and for matrix multiplication, the numerical ranks of the blocks do not change.

In terms of comparing programming models, [18] compared UPC++ with the Partitioned Global Address Space (PGAS) implementation of direct linear solvers for sparse symmetric matrices with two state-of-the-art ones and showed favorable results. Direct comparisons between several task-based runtime systems using a set of benchmarks to help application developers made informed decisions on the transition from MPI+X models [77]. However, all of these efforts dealt with regular and certainly less dynamic applications, and this study will complement their findings using a BLR factorization. More recently, a more comprehensive benchmarking suite comparing multiple parallel programming approaches was proposed [99]. With a unified framework for testing the scalability, imbalanced workload and runtime overheads, it provided many great insights for runtime optimizations.

3.3 Block Low-Rank Factorization Algorithm

To store the matrix in BLR format, our implementation uses a geometric-based partitioning algorithm [80] (to obtain high compression rate of the matrix) and tolerance-based recompression [83, 26] during the factorization, for all the low-rank off-diagonal blocks. For the application of interest, namely the LU factorization, when the matrix is properly ordered and partitioned, many of the off-diagonal blocks can be well approximated using small ranks. As a result, when n is the dimension of the coefficient matrix, BLR has the potential to reduce the storage and computational complexities of factorization to $\mathcal{O}(n^{1.5})$ and $\mathcal{O}(n^2)$ from $\mathcal{O}(n^2)$ and $\mathcal{O}(n^3)$, respectively, when using the dense matrix format [13]. All diagonal blocks are stored in the dense format and treated as dense with regard to computations.

At each step of factorization, BLR algorithm first computes the LU factorization of the leading dense diagonal block using the LAPACK subroutine dgetrf. Then, the off-diagonal blocks aligned with the leading block's row and column, commonly known as panels, are factorized using the BLAS triangular solve DTRSM. with the lower- and the upper-triangular factors of the diagonal block, respectively. These panel blocks are then used to update the trailing submatrix block by block. Figure 3.1(a) shows the resulting factorization algorithm.

In BLR format, the off-diagonal blocks can be either low-rank or dense. Thus, when updating the trailing blocks $B_{i,j}$ on Figure 3.1(a), each of the three blocks involved, $B_{i,j}$, $L_{i,k}$, and $U_{k,j}$ can be either dense or low-rank, giving eight potential configurations for the updating kernel. We update these blocks according to the approach that would minimize the floating-point operation (FLOP) count (see Figure 3.2 for an illustration). When a dense block $B_{i,j}$ is updated using two low-rank blocks, $L_{i,k} = V_{i,k}Y_{i,k}^T$ and $U_{k,j} = V_{k,j}Y_{k,j}^T$, we first compute the small matrix $T := Y_{i,k}^T V_{k,j}$. We then multiply T with either $V_{i,k}$ or $Y_{k,j}^T$, depending on the required FLOP counts. Finally, $B_{i,j}$ is updated with the low-rank matrix, i.e., $B_{i,j} := B_{i,j} - V_{i,k}(TY_{k,j}^T)$. Similarly, to update a dense block using a low-rank block and a dense block, we first merge the dense block into the low-rank block, i.e., $B_{i,j} := B_{i,j} - V_{i,k}(Y_{i,k}^T B_{k,j})$. On the other hand, if $B_{i,j}$ is a low-rank block, we can then directly merge the low-rank representation of the update with the original low-rank representation of $B_{i,j}$, i.e., $B_{i,j} := \widehat{V}_{i,j}\widehat{Y}_{i,j}^T$, where $\widehat{V}_{i,j} = [V_{i,j}, -\bar{V}_{i,j}]$ and $\widehat{Y}_{i,j} = [Y_{i,j}, \bar{Y}_{i,j}]$, and $V_{i,j}Y_{i,j}^T$ is the original low-rank representation of $B_{i,j}$ before the update, while $-\bar{V}_{i,j}\bar{Y}_{i,j}^T$ is its low-rank update to be applied. Compared with the dense-block update that requires $O(n_i n_j n_k)$ FLOPs, the low-rank update only requires $O(n_i n_j \min(r_{i,k}, r_{k,j}))$ FLOPs, where $r_{i,k}$ and $r_{k,j}$ are the respective numerical ranks of the blocks $L_{i,k}$ and $U_{k,j}$, and n_k is the dimension of the k-th diagonal block. As a result, when the blocks have small ranks, i.e., $r_{i,k}, r_{k,j} \ll n_k$, low-rank compression can significantly reduce the FLOP count.

To avoid the increase in the numerical rank while maintaining the user-specified accuracy, we use Adaptive Cross Approximation (ACA) [83, 26] to recompress the low-rank block after each update. As shown in Figure 3.1(b), at each step of ACA, we compute the pivot row (and

for $k = 1, 2, \ldots, n_t$ **do**
  //Factorize diagonal block
  $[P_k, L_{k,k}, U_{k,k}] := \mathrm{LU}(B_{k,k})$
  for $i = k+1, \ldots, n_t$ **do**
    //Compute blocks in panel column
    $L_{i,k} := B_{i,k} U_{k,k}^{-1}$
  end for
  for $j = k+1, \ldots, n_t$ **do**
    //Compute blocks in panel row
    $U_{k,j} := L_{k,k}^{-1} P_k B_{k,j}$
  end for
  for $i = k+1, \ldots, n_t$ **do**
    for $j = k+1, \ldots, n_t$ **do**
      //Update trailing block
      $B_{i,j} := B_{i,j} - L_{i,k} U_{k,j}$
    end for
  end for
end for

(a) LU factorization, where n_t is the numbers of the blocks in the matrix row or column.

$\Pi_{\text{row}} = \emptyset$, $\Pi_{\text{col}} := \emptyset$, $r := 0$, $\pi_1 := 1$
while not converged **do**
  // increment numerical rank
  $r := r + 1$
  // generate pivot row
  $\mathbf{y}_{:,r} := \mathbf{b}^T_{\pi_r,:} - \mathbf{y}_{:,1:r-1} \mathbf{v}^T_{\pi_r,1:r-1}$
  // pick pivot column
  $\pi_r := \arg\max_j (|y_{j,r}| : j \notin \Pi_{\text{col}})$
  $\Pi_{\text{col}} := \Pi_{\text{col}} \cup \{\pi_r\}$
  // generate pivot column
  $\mathbf{v}_{:,r} := \mathbf{b}_{:,\pi_r} - \mathbf{v}_{:,1:r-1} \mathbf{y}^T_{\pi_r,1:r-1}$
  // pick pivot row
  $\pi_r := \arg\max_i (|v_{i,r}| : i \notin \Pi_{\text{row}})$
  $\Pi_{\text{row}} := \Pi_{\text{row}} \cup \{\pi_r\}$
  // convergence check
  $\|E\| := \|V_{:,1:r}\| \|Y_{:,1:r}\|$
  if $r == 1$ **then** $\|A\| := \|E\|$
  if $\|E\| \leq \tau \|A\|$ **then** break;
end while

(b) ACA compression to compute a low-rank VY^T form of a block B.

Figure 3.1: Low-rank matrix factorization and compression algorithms.

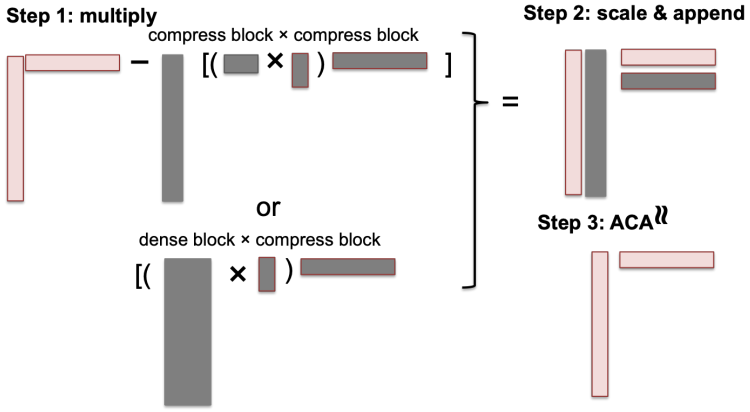


Figure 3.2: Illustration of algorithm updating a low-rank block.

column) by multiplying the corresponding row of $\widehat{Y}_{i,j}$ (and $\widehat{V}_{i,j}$) with $\widehat{V}_{i,j}$ (or $\widehat{Y}_{i,j}^T$). Thus, we do not explicitly form the dense representation of the whole low-rank block. The algorithm terminates when the user-specified accuracy of the approximation is obtained. As a result, the numerical rank of each block may change at each step of the factorization.

Our LU implementation seeks pivots only within the diagonal block, ignoring the potential pivots outside the diagonal blocks. This pivoting scheme (combined with the matrix balancing) was sufficient to maintain the numerical stability of the factorization for matrices arising from the applications we were interested in.

3.4 Required Features

I highlight some of the most critical features needed in order to implement an efficient BLR factorization algorithm. Most of these requirements are generic enough to be applied regardless of the programming model, but some are particular to task-based models.

1. Mitigate the load imbalance at each step due to variable task granularities (different sizes of blocks whose numerical ranks change dynamically).

2. Allow dynamic reallocation of the data that define the dependencies among the tasks (to store the low-rank block whose numerical rank changes, e.g., the numerical rank could increase).

3. Handle the dynamically changing size of the data to be sent or received (to send the low-rank block whose numerical rank is known only at run time).

4. Provide the means to overlap communication with computation (e.g., using a communication thread). A fork–join programming model (e.g., with MPI + OpenMP) without dedicated communication tasks or threads may not be sufficient.

5. Enable the ability to specifically highlight the critical path of the algorithm, and prioritize its execution.

6. Define task or data dependencies at runtime (e.g., depends on the input matrix due to empty blocks, though the dependencies are not changed during the factorization).

7. Support heterogeneous systems (e.g., the ability to offload work to GPUs) and manage devices' tasks automatically.

Most of the target programming models have some level of support for these features, even if in some instances the burden of handling concurrency (or potential parallelism) is on the developer. All MPI-based approaches (flat-MPI, MPI+OpenMP, and AMPI) claim support for asynchronous, or non-blocking, communication and for collective communication. In addition, AMPI supports load balancing via migration of computations to a less busy peer and communication-computation overlap by maintaing a highly oversubscribed state in their approach, but behaves in the same way as MPI for other features.

At the current stage, PaRSEC supports all but (3), where on the receiver a fixed size temporary buffer is used. In addition, I did not use feature (2) in the current PaRSEC implementations of BLR factorization (we only send the required data, but need a larger buffer). Thus, current PaRSEC implementation requires two additional parameters for specifying the maximum numerical rank of each block, and setting the size of the buffer (i.e., the minimum rank r_{min} and the ratio r_{rate} with respect to the block sizes such that the maximum rank for the (i, j)-th block is given by $\max(r_{\text{min}}, r_{\text{rate}} \cdot \min(n_i, n_j)))$. While these parameters could have been the target of an autotuning campaign, I selected in this study the default values based on the input data; they might not be optimal but they should be relatively close. In the experiments, the maximum rank is set such that it is larger than the ranks chosen during the factorization, leading to a larger memory requirements for the PaRSEC implementation compared with the other implementations.

3.5 Implementation with the Programming Models

In the following sections, I describe the design and distributed-memory implementations of an optimized version of the BLR LU algorithm using different programming models. First,

I explore an MPI-based version based on the Flat MPI model, and then extend it with the integration of OpenMP – the MPI+OpenMP model. To facilitate the handling of the load imbalance and minimize the waiting time, I also explored an oversubscribed resources model for the Flat MPI approach using Charm++ AMPI as the implementation layer.

3.5.1 Flat MPI Programming Model

To parallelize the BLR LU factorization on distributed-memory computers, my first implementation follows the ScaLAPACK LU implementation and is based on the Flat MPI programming model. I arrange the MPI processes on a 2-D grid of dimensions p-by-q and distribute the blocks in a Two-Dimensional Block-Cyclic (2DBC) fashion among the processes (each block is stored in a contiguous memory region called a tile). Then, to factorize the matrix, each process updates and factorizes only its local blocks.

To gather the non-local blocks that are needed to update the local blocks from another process, each process creates two MPI sub-communicators: one for the processes in the same column of the process grid and the other for the processes in the same row. Then, at each factorization step, the blocks in the current panel are broadcast using these two sub-communicators.

Since the numerical rank of a low-rank block can change after each update, the processes involved in the broadcast must be informed of the size of the data prior to the broadcast, so the communication of the low-rank block is divided into two messages: the first message propagates the current numerical rank, and the second message the low-rank block data.

The LU factorization with local pivoting is relatively simple to implement in the Flat MPI programming model especially with the 2DBC distribution. However, the collective communication required for the panel update that executes within the panel sub-communicators introduce a synchronization at each factorization step. When load imbalance exists among the processes at each factorization step (e.g., for the trailing submatrix update due to the different sizes and types of the blocks), many processes would idle while waiting for the slowest process at these synchronization points, leading to a significant performance

loss. My evaluation of the effects of standard techniques for algorithmic optimization (e.g., lookahead, accumulated update with multiple panels, balanced block sizes) showed no significant benefits.

3.5.2 Flat MPI with Charm++/AMPI

AMPI provides an MPI implementation that is built on top of the Charm++ framework. It uses user-level threads instead of OS processes to allow several MPI processes on a single physical core, providing the benefits mentioned above to the MPI code. It has been shown that AMPI can improve the performance of the Flat MPI implementation for many imbalanced applications and benchmarks [3] [19].

Porting the Flat MPI implementation to use AMPI requires minimal effort. We only need to change the name of the main routine to mpi_main, and to switch the compiler and linker to the ones required by AMPI. Setting the over-subscription factor could be challenging, but in our case the load imbalance was reproducible and relatively enough to allow us to tune the over-subscription parameter manually. The expectation was that the over-subscription would be highly beneficial, as the MPI processes are spending a significant amount of their execution time blocked on MPI_Bcast, and thus another process on the same node could then utilize the physical core for computations—thus reducing the idle time of the core.

3.5.3 OpenMP Task Programming Model

In order to manually remove the synchronization points, the second implementation relies on the OpenMP task programming model. Then, at run time, the OpenMP scheduler executes both computational and communication tasks of the factorization as their dependencies are resolved. In this implementation, I cannot use the memory pointers to the required data to track the data dependencies among tasks because the compressed blocks are dynamically freed and reallocated as their numerical ranks change after each update. Instead, I used a separate n_t-by-n_t integer array to keep track of the task dependencies.

With this implementation, the OpenMP runtime manages the dependency graph of only the local tasks, and does not form the global dependency graph of the factorization. Hence, the tasks are scheduled for the execution once all the local dependencies are resolved. However, when the task needs to communicate blocks with other processes, the thread will call MPI_Bcast either to send the local block (its current numerical rank and then the data) or to receive the non-local block. Thus, these tasks may block until the corresponding communication task is scheduled on other processes, contributing to the idle time of the local CPU core.

In order to reduce the number of tasks that are blocked due to the call to MPI_Bcast and are keeping the core idle, a nested parallelization was implemented. In this implementation, a single task updates all the blocks in one block column, but once it is scheduled to execute the update, it launches the child tasks, each of which updates one of the blocks in the column. To integrate nicely and maximize the performance of MPI in a multi-threaded environment, I applied a selection of the techniques described in a report about multithreaded MPI implementations [91]. I created a separate communicator for each thread (to minimize the cost of MPI's tag matching and the potential for message overtaking) and I used the communicators in a round-robin fashion on the block columns at each step of the factorization. I placed a higher priority on factorizing the panel column and updating the next panel column since all the tasks updating the trailing matrix blocks depend on these panel columns as shown in Figure 3.3. I used the depend clause to specify the data dependencies among the tasks, where A.getTile(k, k) returns the pointer to keep track of the (k, k)-th block. Line 10 is a blocking call that factors the diagonal and broadcast the data to panel row/column. The calls to lookaheadUpdateA() and remainingUpdateA() have similar structure, where we create an OpenMP task for updating the column. In that task, we solve the panel for obtaining the values in the U factor, broadcast it down the column, then create nested tasks to compute individual updates.

In order to factorize a large matrix, the MPI buffers used to store the non-local blocks needed to be deallocated once all the tasks that require the blocks have completed. Thus,

```
#pragma omp parallel
#pragma omp master
{
  // start pipeline (factor 1st panels)
  factorPanel(0, A);
  for (int k = 1; k < A.getNt(); k++) {
    lookaheadUpdateA(k-1, A);

    // factor next panel
    factorPanel(k, A);

    // update remaining submatrix
    // using current (k-1)th panel
    remainingUpdateA(k-1, A);
  }
}
```

(a) BLR factorization with OpenMP.

```
int *tileA = A.getTile(k, k);
int *tileB = k == 0 ? A.getTile(k, k) : \
                      A.getTile(k-1, k);

#pragma omp task priority(1) \
                 depend(in:tileB[0:1]) \
                 depend(inout:tileA[0:1])
{
 // factor diagonal
 if (A.isLocalRow(k) || A.isLocalCol(k)) {
     A.factorDiagBlock( k );
 }
 // compute off-diagonal L
 if (A.isLocalCol( k )) {
     for (int i = k+1; i < A.getMt(); i++) {
         if (A.isLocalRow( i )) {
             #pragma omp task priority(1)
             {
                 A.computeL(i, k);
             }
         }
     }
     #pragma omp taskwait

     if (!A.isLocal(k, k)) {
         A.freeBuffer(k, k);
     }
 }
 // broad cast tiles in panel along the rows
 A.iBcastL(k);
}
```

(b) Factor diagonal block and nested tasks for panel column update.

Figure 3.3: OpenMP task implementation of BLR factorization

I inserted the tasks that set and decrement the counter for each non-local block, and once the counter becomes zero, the task deallocates the block.

The BLR factorization has a relatively simple dependency graph, and the computational kernel, which each task executes, has been already separated into its own subroutine for the Flat MPI implementation. Thus, it did not present a significant challenge to integrate OpenMP tasks to the sequential code. Furthermore, since many of the application codes already use OpenMP, and my implementation can leverage that so it does not require any changes to compile the code. Overall, the tasking improved the performance of Flat MPI by removing the synchronization points and reducing the idling time of the cores due the load imbalance. However, correctly scheduling the communication tasks for optimal performance remained a challenge. As the process count increased, it became progressively more difficult to coordinate these communication tasks, and some of them might have been blocked waiting on the communication to finish, and thus they kept the CPU cores in idle state.

3.5.4 PaRSEC DTD

DTD allows the sequential task insertions into the PaRSEC runtime, hence providing a simpler to use API, capable of describing parallel distributed algorithms. In order to use the DTD API, the user must specify the distribution of the data that the tasks operate on, the dependencies among the tasks through their data usage, and the code that each task executes once all the required data become available. Recent study on StarPU [4] has demonstrated that by pruning the task graph it is possible to delay the task insertion bottleneck, allowing sequential task insertion model to scale to a larger number of processes, each storing its own pruned version of the global task graph. Although DTD could benefit from such optimization, we did not implement it in the current version.

In order to remove the need to manually move data among the processes and managing temporary buffers for the non-local data, I ported the OpenMP implementation to use the DTD interface in PaRSEC. Since DTD provides the sequential task insertion interface, as can be seen in Figure 3.4, the DTD implementation resembles the OpenMP implementation.

```
for(k = 0; k < NT; k++){
  // diagonal DGETRF
  insert_task(taskpool, parsec_dgetrf,
    1, "getrf",
    sizeof(int)  , &k                ,VALUE,
    PASSED_BY_REF, TILE_OF(A, k, k) ,INOUT | AFFINITY,
    PASSED_BY_REF, TILE_OF(IP, k, 0),OUTPUT,
    PARSEC_DTD_ARG_END);
  if(k < NT-1){
    for(int i = k+1; i < NT; i++){
      insert_task(taskpool, parsec_dtrsm_l,
                  ...);
      insert_task(taskpool, parsec_dtrsm_u,
                  ...);
    }
    data_flush(dtd_tp, TILE_OF(A, k, k));
    data_flush(dtd_tp, TILE_OF(IP, k, 0));

    for(int i = k+1; i < NT; i++){
      for(int j = k+1; j < NT; j++){
        insert_task(taskpool, parsec_dgemm,
                    ...);
      }
    }
  }
}

int parsec_dgemm(parsec_execution_stream_t *es,
                 parsec_task_t *this_task) {
  int k, i, j;
  double *A, *B, *C;
  parsec_dtd_unpack_args(this_task, &k, &i, &j,
                         &descA, &A, &B, &C);
  int rankA = (int)A[0]; // rank of non-local block A
  int rankB = (int)B[0]; // rank of non-local block B
  int mb = descA->super.nbi[i]; // # of rows in block C
  int nb = descA->super.nbi[j]; // # of cols in block C
  // perform update
  ...
  // update the output message size
  new_count = rank * (mb + nb) + 1;
  dtd_update_count_of_flow(this_task, 2, new_count);
}
```

Figure 3.4: PaRSEC DTD implementation of BLR factorization, including insertion of the tasks in sequential order with the data usage information provided.

Thus, it was straightforward to implement it and I show how the data usage is specified only for the dgetrf task: it executed the code parsec_dgetrf that takes three arguments k, A, and IP, where the diagonal block A and pivoting IP are passed in by reference. The INOUT flag indicates that the task reads data and by its completion time it would have written new data to the same place. AFFINITY flag indicates that this task will be executed on the process that owns the k-th diagonal block. PARSEC_DTD_ARG_END signals the end of parameters list. DTD provides an API call dtd_update_count_of_flow to update the size of the data to be sent in the task body.

At each step, first the diagonal factorization task dgetrf is inserted. Then computation proceeds with the off-diagonal blocks of the panel by inserting the triangular solve tasks dtrsm_l and dtrsm_u for each off-diagonal block in the lower and upper triangular factors, respectively. Finally, the implementation continues by inserting the tasks that update each block in the trailing sub-matrix. To recycle the temporary buffer for the non-local data, data_flush needs to be called when the non-local data is no longer needed.

In order to transition our MPI implementation to use the PaRSEC runtime, I needed to deal with the data distribution in the PaRSEC data descriptor format. Though the PaRSEC data collections can be more dynamic and support non-regular, non-2DBC distributions, I selected the PaRSEC data collection to be regular 2DBC distribution, which the MPI implementation uses, ensuring a uniform implementation across the models.

BLR factorization requires the runtime system to dynamically change the size of the data being sent or received since the numerical rank of the block changes during the factorization. DTD provides this capability by enabling selection of the size of the data in the task body. I use this feature such that our implementation sends only the required amount of data specified by the current numerical rank. Similar to Flat MPI, the dynamic size of the blocks imposes an increased communication load, as the size of the blocks must be propagated before sending the block data.

In order to maintain the minimum amount of the memory usage, I would like to reallocate the data as the low-rank block is recompressed. PaRSEC provides a flexible data descriptor

that supports irregular data sizes, which allows the reallocation of the data to accommodate rank changes. The current implementation does not use this functionality. Instead, I specify a maximum rank for each low-rank block to avoid the reallocation at the cost of higher memory consumption.

3.5.5 PaRSEC PTG

PaRSEC's PTG DSL uses a concise, parametrized task graph description known as Job Data Flow (JDF) to represent the data dependencies between tasks. As I will show subsequently in the later section, the developer must specify a few crucial pieces of information for each task class: 1) the data distribution, 2) the possible input parameter values, 3) the process that will execute the task based on the data distribution, 4) the data dependency between the task classes and 6) the actual code body of the task. Based on these pieces of information, PaRSEC can discover and then execute all the available tasks at runtime, moving the data between processes as the tasks are completed and it can be achieved without exploring let alone instantiating the whole task graph at once in memory: this may be considered an *implicit* task graph representation. Previous results have shown that PTG can deliver a significant percentage of the hardware peak performance on heterogeneous distributed machines [116].

In the dataflow description of the PTG DSL, each computational task is defined by a set of parameters and a number of input and output flows of data. Unlike in the DTD implementation, the PTG model requires the programmer to express the data dependencies between tasks as mathematical relationships between the tasks' parameters. These data dependencies, along with the shape and size of the data, must be specified and agreed upon by a pair of tasks that is sending and receiving the data.

Figure 3.5 shows a JDF specification of the diagonal factorization task, where the parameter `k` defines the task for factorizing the k-th diagonal block. In the figure, "`RW`" designation specifies that these diagonal factorization tasks both read and write the data (equivalent to "INOUT" label in DTD), while the left- and right-arrows show where the data

```
1  dgetrf(k)
2  k = 0 .. NT
3
4  : descA(k, k) //locality
5
6  RW A  <- (FIRST) ? descA(k,k)
7        <- (!FIRST) ? C dgemm(k_prev, DIAG, DIAG)
8
9        -> (END>=START) ? A dtrsm_l(k, START..END)
10       -> (END>=START) ? A dtrsm_u(k, START..END)
11
12 RW IP <- IP ipiv_in(k)  [type = PIVOT count = NB]
13       -> IP ipiv_out(k) [type = PIVOT count = NB]
14
15 /* Priority */
16 ;1
17
18 BODY
19 {
20   // Factorizing diagonal block (k, k)
21   int mb = descA.nbi[k];
22   double *dA = &(((double*)A)[1]);
23   iinfo = LAPACKE_dgetrf(LAPACK_COL_MAJOR,
24                          mb, mb, dA, mb, ipiv);
25 }
26
27 dgemm(k, i, j)
28 ...
29   RW C <- (k == 0) ? descA(i, j) : C dgemm(k-1, i, j)
30                           [count = COUNT_C]
31        -> (k == lastk && i == j) ? A dgetrf(m)
32                           [count = COUNT_C]
33 ...
34
35 /* Priority */
36 ;(j == k+1 ? 1 : 0)
37
38 BODY
39 {
40   ...
41   // update the output message size
42   this_task->locals.COUNT_C.value = 1 + ranks * (mb + nb);
43 }
```

Figure 3.5: PaRSEC PTG specification of the diagonal factorization tasks: defining the parameter space, data locality, and data dependencies, written in JDF.

is read from, and written to, respectively, at the completion of the task, e.g., for reading the data A, "descA(k,k)" indicates that the data is read from the memory at the initialization, while "C dgemm(k_prev, DIAG, DIAG)" indicates that the task dgemm(k_prev, DIAG, DIAG) will send the data C, which the diagonal factorization uses as A. The "type" combined with "count" indicates the temporary buffer size for sending and receiving the data. It is possible to change the size of the data to be sent (e.g., when the numerical rank changes after the recompression), by changing the local value passed to "count" in BODY.

The second line specifies the range of the parameter, showing that all integer values between 0 and the last diagonal index, NT, are legal for the parameter k. On the third line, the locality statement specifies that the k-th diagonal factorization task will be executed by the process that owns the specified data (i.e., the k-th diagonal block). Finally, the data dependencies for the tasks are defined (the data can be initialized by reading from the memory, written to the memory, or passed in or to another task).

For the computational task to be executed, once all the input flows are locally available, we can simply call (in the BODY) the computation kernels developed for the Flat MPI implementation.

Given this dataflow expression in JDF format, the PTG preprocessor generates the C/C++ code that encodes the symbolic task representation. Then, at run time, the PaRSEC runtime explores the task graph, moves the specified data between the tasks, and executes the tasks as all the required data become available—without the overhead of task discovery, which the DTD implementation has to endure.

Both PTG and DTD implementations use the same data distribution descriptor, allowing a smooth transition from the DTD to PTG implementation. From programmability perspective, PTG introduces a completely different parallelization philosophy, driven by data dependencies and not by control dependencies. For most of HPC users, converting their parallel applications (e.g., parallelized with MPI and OpenMP) might require substantial amount of effort. However, the description provides enough information to the runtime itself

to allow for automatic communication and computation overlap, as well as collective pattern description, providing a strong base for more scalable and more efficient implementations.

3.6 Performance Evaluation

In addition to evaluating the effort needed for each implementation qualitatively, in this section I compare the performance of the models quantitatively.

3.6.1 Experimental Setup

For experiments, I used a software package called ppohBEM [81] that numerically solves the integral equations for simulating the electrostatic field based on the boundary element method. In particular, I used the BLR matrices generated by the software package called HACApK [80], which uses the low-rank matrix format for solving dense linear systems of equations. To compute the appropriate matrix permutation and partition for generating the low-rank matrix, HACApK uses the geometrical information associated with the underlying physical problem such that the off-diagonal blocks of large dimensions become low-rank. Figure 3.7 shows the size information of the test matrices, and their initial numerical ranks are shown in Figure 3.6.

I compiled the entire software stack using Intel Parallel Studio XE 2019 suite and linked with the corresponding MPI and OpenMP library. I used PaRSEC library from the master branch as of June 2019 with DTD dtd_update_count_of_flow API in development branch, and the release version 6.9.0 of Charm++. The experiments were conducted on the Bridge cluster located at Pittsburgh Supercomputing Center (PSC). Each compute node has 2 Intel Haswell (E5-2695 v3) CPUs with 14 cores per CPU, running at 2.3–3.3 GHz, and are interconnected using Intel Omni-Path networking equipment.

Experiments were run using all 28 cores per node starting from one node all the way up to 16 nodes (the total of 448 cores), which were enough to show the overall performance trend. The results for the 1ms dataset start from 4 nodes due to memory constraint. For the Flat

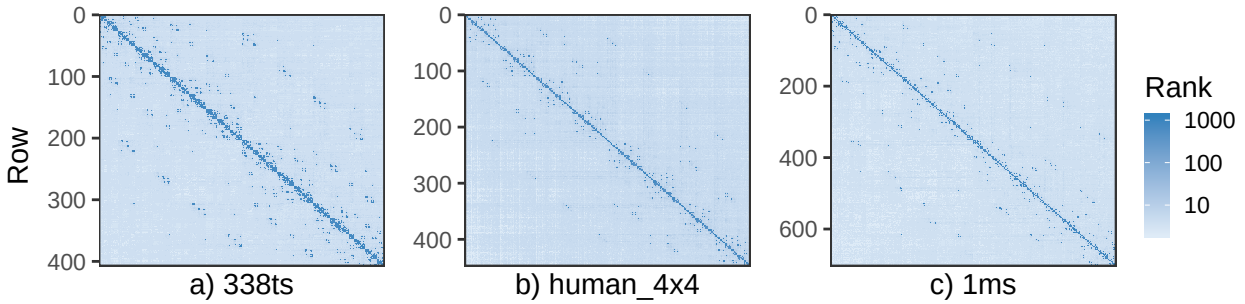
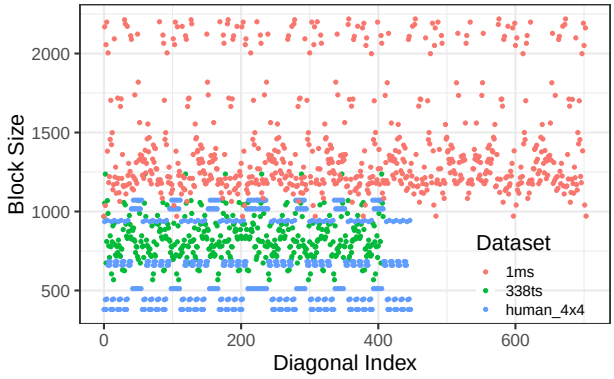


Figure 3.6: Initial block ranks for each test matrix, all have dense tiles near diagonal, but different off-diagonal low rank patterns



(a) Diagonal square blocks sizes, and the off-diagonal blocks will be rectangular

	matrix dimension	# of block columns
338ts	338000	408
human_4x4	314624	448
1ms	10004400	704

(b) Test matrices

Figure 3.7: Test matrices information

MPI model, each core has one process; for the MPI+OpenMP model, the best configuration I observed is with two processes per socket (with the socket's cores evenly divided between the processes), so the total of four MPI processes ran per node. For the AMPI model, I used the SMP mode, with two processes per node, each with 14 threads, and I set the virtual process number to be three times higher than the physical core count. Finally, for the PaRSEC implementations I had one process per node with one core dedicated to the communication thread, and the rest – as computational threads. To avoid the non-uniform memory access (NUMA) effects when accessing the main memory, PaRSEC data was initialized by all the threads to ensure uniform affinity of the virtual memory pages because first-touch policy was in effect.

3.6.2 Experiment Results

Flat MPI

In the experiments with the 2DBC data distribution of the blocks, the total computational load was well balanced among the participating processes. However, at each step of the factorization, the changing sizes and numerical ranks of the blocks created a significant load imbalance among the processes. Since the Flat MPI implementation introduces global synchronization, all the processes are forced to wait for the slowest processes by design, and the accumulated idle time due to the load imbalance can become a significant portion of the total factorization time. This observation had motivated us to explore alternative programming models besides Flat MPI.

Figure 3.8 illustrates these load imbalance issues for the three test matrices. To measure the imbalance, I put an artificial global barrier before each broadcast and accumulated this wait time for each process. As shown in the figure, the average idle time can be as high as 77% of the execution time, while the error bars indicate that the total computational load among the processes has a much smaller variation for most cases.

Thus, the existence of such a large imbalance effect opens up the possibility for opportunities favoring approaches that rely on node oversubscription to translate this wasted

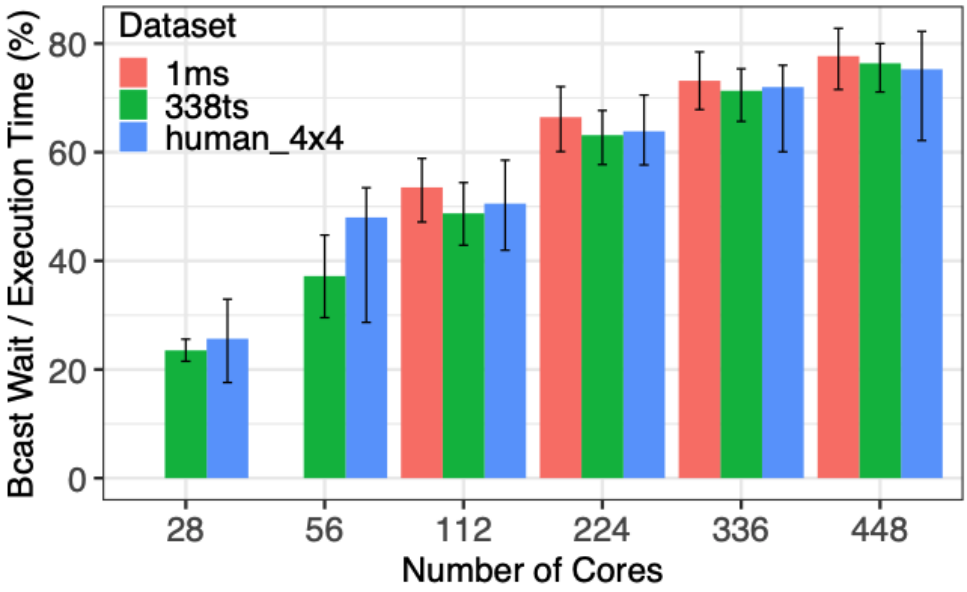


Figure 3.8: The average wait time of a MPI process in a collective call for the flat MPI model, shown as percentage of total execution time. Minimum and maximum shown as well

wait time into useful computation time for another thread residing on the same node. Moreover, in the case where over 50% of the time is wasted on average on all processes, it seems extremely plausible that oversubscription could drastically reduce the wasted time and therefore minimize the time-to-solution.

AMPI

In Figure 3.9, the green and the pink lines show the results comparing AMPI with Flat MPI. By oversubscribing the cores, I expect AMPI to be able to reduce the idle time, thus achieving better performance than the Flat MPI model. Unfortunately, the result contradicts my expectation. To investigate why the AMPI is taking more time to execute, I timed the different sections in the Flat MPI/AMPI implementation using a smaller test dataset on a single node (28 processes, 84 virtual processes). Figure 3.10 shows the trace for one process. FactorDiag includes dgetrf and the resulting broadcast to panel row and column. PanelUpdate computes the panel, BCastPanel broadcasts the panel blocks to the corresponding column or row processes. UpdateRemain is the computation of the update kernel on trailing submatrix.

Since I oversubscribe by 3:1, AMPI's UpdateRemain time is roughly 1/3 of Flat MPI's time. But the AMPI BCastPanel time is much larger and is the reason for the longer execution time. We varied the oversubscription factor from 1 to 5, and 3 was the best configuration.

MPI+OpenMP

In Figure 3.9, in addition to the factorization time, the black line shows the average total compute time for each process, as obtained from the Flat MPI model result. It also has ticks for the minimum and maximum among the processes as well. The line serves as an unattainable lower bound of the execution time, as it represents the most favorable scenario, one that only accounts for computational costs and completely disregards all costs related to data movement.

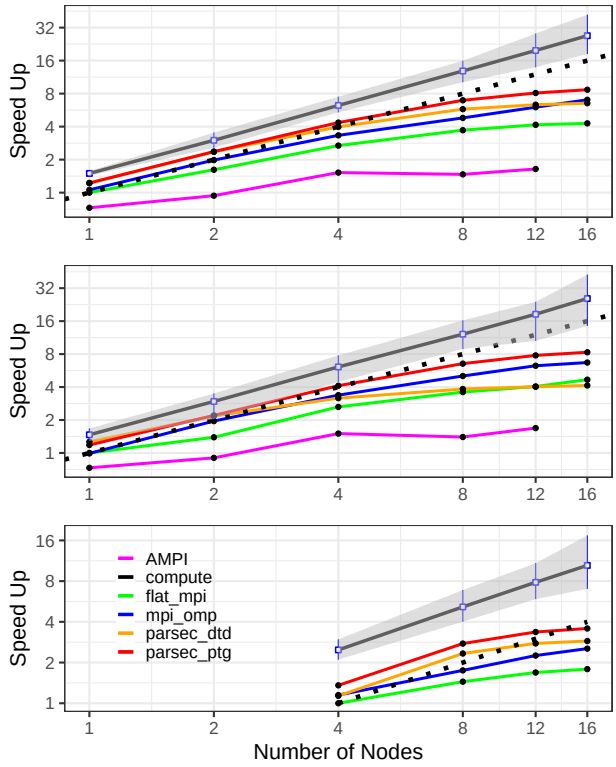


Figure 3.9: Execution time of each model on different datasets, top) 338ts, middle) human_4x4, bottom) 1ms. Flat MPI performances on 1 node (28 cores)/4 nodes are used as base to show speed up of the models. They are 317, 221 and 1050 seconds respectively. Both X- and Y-axis are plotted on log2 scale. PTG speed up over MPI+OMP are 1.23, 1.24 and 1.40 at 16 nodes

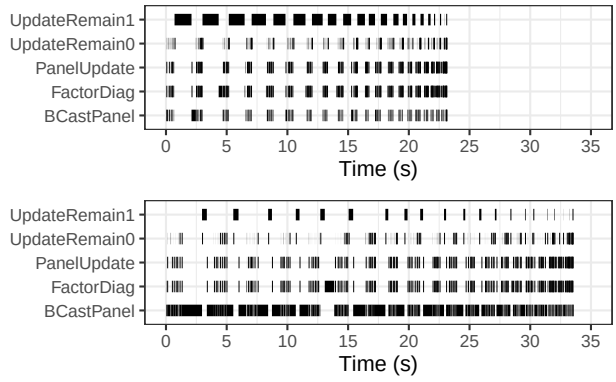


Figure 3.10: Execution stream of the different sections for one selected process, top) Flat MPI, bottom) AMPI. Most of the BCastPanel time are likely idle time

The first thing I noticed was that Flat MPI model performs the worst among the other programming models that I tested, while the remaining models performed better to some degree. This is expected as the strongest point of all the other approaches is to relax the strong synchronization inherent to the Flat MPI model to some extent. The second thing was that MPI+OpenMP scales well as the number of nodes increases, but there are still limitations to prevent it from obtaining better performance, as I will analyze later.

PaRSEC DTD

The DTD implementation performs better than MPI+OpenMP at the beginning, which can be attributed to its finer-grain dependency tracking, further removing the synchronization imposed on block columns. But PaRSEC DTD has its own issues, mainly with regard to the scalability of the sequential task insertion. As the node counts increase, the performance begins to deviate from that of the PaRSEC PTG version to finally become worse than the Flat MPI result for the human 4×4 dataset on 16 nodes. I believe that, as we strong-scale, the overhead of PaRSEC DTD task discovery starts to occupy a bigger portion of the execution time, and the discovery and insertion of local tasks being slower than the execution of already inserted tasks. The proposed DAG trimming technique [4] was likely to help in mitigating the problem but is not implemented in this case.

PaRSEC PTG

The PTG implementation performs consistently better than the other models. Not only it removes all global synchronization (replacing them with fine-grain synchronization points at the task level), but also creates more opportunity for communication-computation overlap. It also has the feature allowing us to specify higher priorities for diagonal tasks. Given the disparity between diagonal and off-diagonal computation loads, this capability ensures high levels of parallel workloads and cores' occupancy by directing the runtime to follow, even if loosely, the algorithmic critical path [36].

To understand the improvement from MPI+OpenMP to PaRSEC PTG, I profiled the executions on a single node. MPI+OpenMP was run with two processes, each on a socket. The top plot in Figure 3.11 shows a summary of each thread's occupancy information, defined as the summation of all the computation kernels' time on a thread divided by the total execution time. On a single node, both models achieve over 90% efficiency, and roughly speaking we can attribute the rest to runtime overheads.

But a closer look at the diagonal factorization kernel in MPI+OpenMP reveals that all the processes in the current panel will call this kernel in order to receive the actual diagonal factorization. Root process will thus complete the computation then block on the broadcast to panel row and column, while the receiving processes will directly block on the broadcast. The bottom plot in Figure 3.11 shows that if the kernel is doing only broadcast (and is not the root process), it takes as long as needed to complete the broadcast and exit. But for the root process, it will block on the broadcast for additional time after the computation.

On the other hand, PaRSEC delegates all the communication to a single thread dedicated to communication and uses non-blocking communication. This removed this synchronization point and likely provided the performance benefits we observe in Figure 3.9 as problem scale increases.

3.7 Conclusions

In this chapter, I presented implementations of BLR LU factorization as test cases for using five programming models: Flat MPI, MPI+OpenMP as well as alternative models AMPI/Charm++, PaRSEC DTD, and PaRSEC PTG. I summarized the experience implementing the algorithm using these models, and evaluated their respective performance. The results indicate the potential for the task-based approaches to address the load imbalance and outperform Flat MPI. Overall, PaRSEC PTG achieved the best execution time and scalability, with a certain cost on the programming effort. PaRSEC DTD provides a smoother transition to task-based runtime but faces scalability issues as the number of nodes grows.

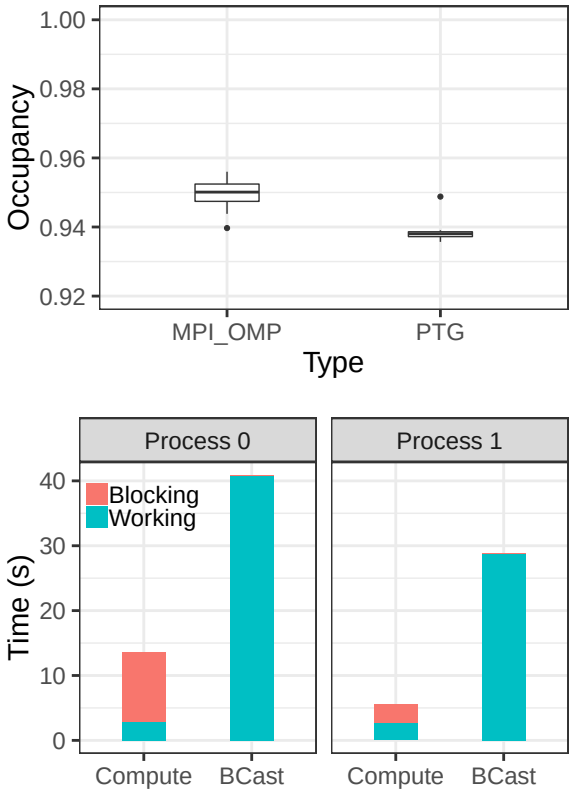


Figure 3.11: Top) Computation kernels occupancy summary of all the threads, Bottom) Detail breakdown of the diagonal factorization task for MPI+OpenMP model

MPI+OpenMP can obtain reasonable results and might be more familiar and easier to implement. AMPI's result for this test case is unexpected and warrants further investigation.

Several features needed for an efficient BLR factorization are highlighted, including necessary capabilities to address load imbalance, handle dynamic data sizes, reduce synchronization, and provide the ability to highlight the algorithm's critical path. I hope that this work can motivate future adoption of alternative programming models to tackle the irregular workloads arising from the system or the application, and improvements of features in runtime systems.

The current implementation is designed as a benchmark to compare different programming models. It is possible to further optimize some of the implementations. For example, instead of using the 2DBC distribution, we may evenly distributed the dense tiles close to the diagonal among the processes, which may greatly improve the load balance and improve performance [36]. Since PaRSEC handles all the data movement, the user just needs to define a new data distribution, making it easy to use a different distribution scheme. Other programming models like Task-aware MPI (TAMPI) [95] implements the interoperability services between MPI and OpenMP tasks, and can be further investigated. I also observed a higher memory consumption of PaRSEC-based approaches due to the temporary buffer used in the runtime, a quantitative evaluation of this aspect will also be interesting in the future. Finally, GPU kernels can also be added to offload work and speed up the computation.

Chapter 4

Sequential Task Flow Runtime Model Improvements and Limitations

4.1 Overview

Task-based runtime systems were developed to manage the challenges of programming at extreme scale and successfully adapted to heterogeneous hardware. In this programming model, the runtime is in charge of scheduling the computational tasks on parallel computing resources, as well as the communication between nodes. The user needs to decompose the algorithm into tasks with explicitly specified data dependencies among them, forming a Direct Acyclic Graph (DAG). This programming paradigm was adopted in recent years by many different software systems, among them StarPU [17], Legion [25], and PaRSEC [31]. Even within task-based runtime systems, there are many different ways to express the algorithm as a task-graph and subsequently to analyze and then to schedule the resulting DAG onto HPC systems.

PaRSEC PTG interface [43] provides a domain specific language (DSL) to describe the algorithm by specifying the individual tasks, as well as the data dependencies between those tasks. Given this compact representation of the graph, each node can process the tasks that will be executed on that node, and can react to the data received from remote

nodes without the overhead of building the global knowledge during execution. And for StarPU and PaRSEC DTD interface [76], the Sequential Task Flow (STF) model provides an easy-to-use API for algorithm formulation. A single thread of execution is responsible for inserting the tasks following the sequential execution order of the algorithm, and the data usage information is provided (either READ, WRITE, or READ-WRITE) so that internally-independent tasks can be scheduled in parallel, and dependent tasks will follow the correct read-after-write orders, and data usage across nodes will trigger the corresponding data transfers and have them inserted in to the scheduling flow.

Although such an interface is easy to use, but the granularity of the tasks needs to be sufficiently high to overlap with the dependency analysis, and this was demonstrated in the previous chapter with the BLR LU implementation, and elsewhere [76, 99]. This analysis overhead increases equally with the problem size on all the processes. This means that STF model will face significant scalability issues especially in the exascale era, where the number of compute nodes can be in the range of hundreds of thousands. Still, when making the transition to programming using a task-based runtime system, the STF model is a very attractive target for the new adopters and they can obtain immediate performance improvement over the more commonly used MPI+X model when running on smaller scale [92].

Also, unlike in the PaRSEC PTG model where parallelism is unleashed eagerly, and can lead to wrong scheduling decisions: certain control flows are needed to enforce task execution priority. STF model usually will have a parameter specifying the size of the window into the global graph of tasks. It is included with the primary goal of limiting the memory usage resulting from graph exploration (the main thread will keep inserting tasks up-to the window size, then join the computation threads for task execution and when the number of tasks decreases below a prescribed threshold the main thread would go back to the task insertion mode). This window size is a tunable parameter and has the side benefit of acting similarly to the lookahead technique that is common in matrix factorization implementations, and

it allows the task execution to follow the critical path of the algorithms, ensuring optimal scheduling for the user.

Given the benefits of the STF model, in this chapter, I would like to push the limits of STF model to achieve better scalability and performance, while balancing it with the ease-of-use of the model. Previous work has tested similar ideas for Cholesky factorization [4] [50]. But I would argue that the dependency graph is relatively simple in the previous studies. As a result, I also implemented and evaluated trimming and broadcast's impact on QR factorization, which has a tighter dependency graph.

The contributions from this work are as follows:

- Create the sender/receiver key internally so that user can trim the DAG during task insertion.
- Adopt a two-stage approach for data broadcast operation for the PaRSEC DTD interface.
- Evaluated empirically the changes in writing the algorithms in order to trim the DAG, and the usability of such an interface for more complicated algorithms.
- Evaluated the impacts of graph trimming and broadcast operation on performances of Cholesky and QR factorizations on two HPC systems at scale.

4.2 User Graph Trimming and Broadcast Operations

4.2.1 DTD Model

PaRSEC as a task-based runtime supports multiple interfaces, the PTG interface requires the users to specify the body of the tasks, and the dependencies between the tasks via the Job Data Flow (JDF) DSL. On the other hand, DTD interface allows users to write sequential-looking code, including conditionals, for-loops, and code blocks to insert tasks using PaRSEC's API without using a custom DSL. Both methods of task graph definition

share the same runtime scheduler, data representation, and communication engine. There are three main concepts that enable expression of a task graph in PaRSEC using DTD: a task, dependency, and data item. A task is any kind of computation that will not block due to communication, data items are regions of main memory used by the computations that will be accessed or modified, and, finally, dependencies are the ordering relationships between tasks in the graph. To insert a task with any of PaRSEC's API options, users must indicate the data and the mode of operation that will be performed on that data by the task (either read, write or read-write). Dependencies between tasks are created based on the operation-type on the data: a task performing a write before a task performing a read on the same data will create a read-after-write (RAW) dependency between the writing task and reading task, such that the reading task will only execute after the writing task is completed. The properly sequenced expression guarantees the correct ordering of tasks regardless of the parallel execution and any data concurrency interaction.

In distributed memory systems all the participating processes need to have a consistent view of the DAG for DTD to maintain the correct sequential order of tasks and requiring the whole DAG to be discovered by all the processes is one solution. This means that many tasks that are not related to a given node will still need to be inserted and inspected, creating a growing overhead as more nodes are involved. With this kind of implementation guaranteeing the insertion of the entire graph, it allows for creating a *unique key* and thus a consistent naming of each task on all the nodes without involving extra communication. This approach allows for simple message matching across nodes, based on this naming scheme and its unique keys. This is a sufficient solution, but is a stronger requirement than what is needed for the STF model.

4.2.2 PaRSEC DTD Tasks and Communications Tracking

PaRSEC communication engine is exposed to the rest of the runtime only through a well-defined interface. The DSLs encapsulate the information of a communication via an object called remote_deps (the circle in Figure 4.1) for remote dependencies that is passed into the

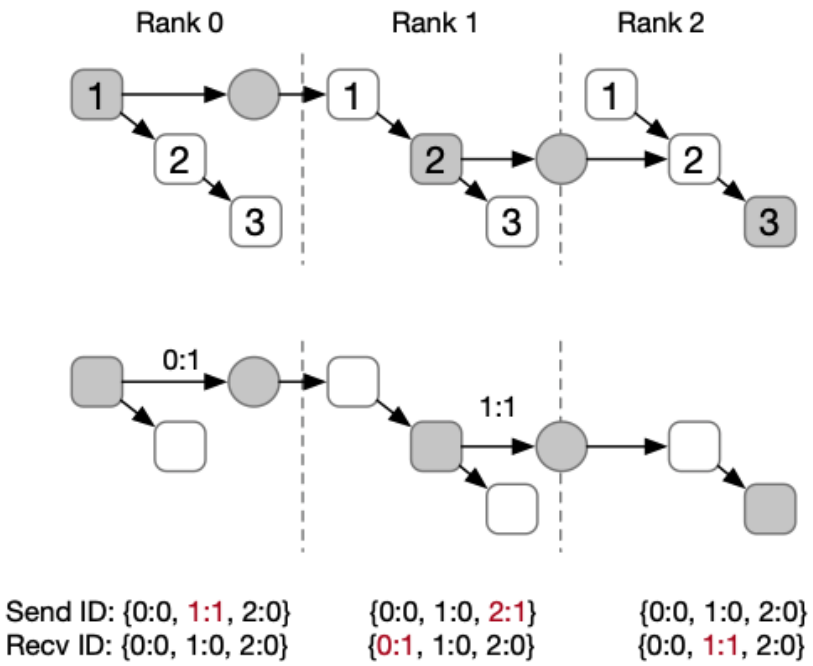


Figure 4.1: Top: original DTD, each task has a unique key Bottom: send/recv level key. Grey square represents local task, white square represents remote task. Circle represents the remote_deps structure. In the new scheme, data flow ID is a combination of sender rank and sequence number to uniquely label each data transfer. As long as both the sender and the receiver has the dependent tasks inserted, the data ID will be assigned correctly for the two sides to match the data transferred.

engine. This abstraction allows PaRSEC to adopt different underlying libraries (right now it uses MPI two-sided) for communication. When we are inserting the tasks, DTD keeps track of the remote parent task or the received remote_deps object in a local hash table with the unique key for a given task. Since each task in the entire DAG has a unique key, the communicated data represented in the remote_deps can be matched with the remote task object and continue the task graph execution.

4.2.3 Graph Trimming

This unique key generated independently on each node is the link between the task management level and the underlying communication engine. The correct message carrying the data will be provided as the input data to the corresponding task via the key generated independently on each node. By observing that for each send-receive pair of exchanging data between two participating nodes, they only need to keep track of the order of the previous communication instances between the two, then they can correctly generate the next key for the point-to-point transfer between the two. So to remove this artificially stronger requirement of inserting all tasks and labeling them uniquely, thus permitting user level graph trimming, each node keeps internal arrays that will track the sends and receives with respect to other ranks instead. With this approach, users can trim the task graph at the user-level transparently, reducing the overhead of the runtime scheduling and improving performance (an example of the arrays is shown in Figure 4.1). This change does not affect the existing code that inserts all tasks on each node, since the irrelevant tasks that get trimmed will never have the data IDs assigned to them and thus will not affect the correct ID assignment for retained tasks.

4.2.4 Broadcast Operation

Collective operations are a critical part of message delivery optimization, especially for large-scale distributed systems. In a typical MPI-based program, collective operations are done via a predefined communicator, and as a result all the callers know the participant ranks.

In a sequential task insertion interface like PaRSEC DTD, tasks are inserted sequentially and the group of nodes/processes/ranks participating in a collective operation are not known beforehand. Previous work [50] implemented implicit broadcast, assuming all the participants are discovered when the data is ready to be send (i.e. the broadcast will cover all the descendant ranks or most of the ranks). The benefit of this approach is that it is transparent to the application writer, your original STF code will benefit without any changes. But the assumption that the task discovery progresses faster than the kernel execution turns out to be a strong one, and risks the possibility of lacking ability to identify collective operations and falling back to doing point-to-point communication.

I proposed an explicit broadcast API, whereby with the knowledge of the algorithm writer, the root of a broadcast call can specify all the participating ranks (the dependent tasks that will use the data), and the participants don't need to know each other. This is the same kind of information that is needed when the user trims the task graph with remote read tasks not knowing each other but will have the same writer task in the root inserted. Also, with an explicit collective API, many similar collective calls can be implemented (reduction/allreduce, gather/allgather etc).

PaRSEC PTG and PaRSEC DTD models share the same underlying communication engine, and a version of broadcast has already been implemented for PTG. It is built on top of the MPI point-to-point operations. Since for PTG, the entire graph's information is represented locally, a descendant can replay the task schedule as the root in order to rediscover the participant ranks and the propagation path, thus can continue the data broadcast downstream. There are two different typologies supported, namely: chain and binomial trees. The mechanism to check for direct descendants is based on a bit array representing participant ranks and, as a result, the route will be fixed given a topology and a set of participant ranks.

Since both DSL variants share the same communication engine, the idea then is to adopt a two-stage approach (Figure 4.2) and reuse many of the same implementations. In this scheme, I first prepare a message containing a global ID, local data keys (the P2P keys

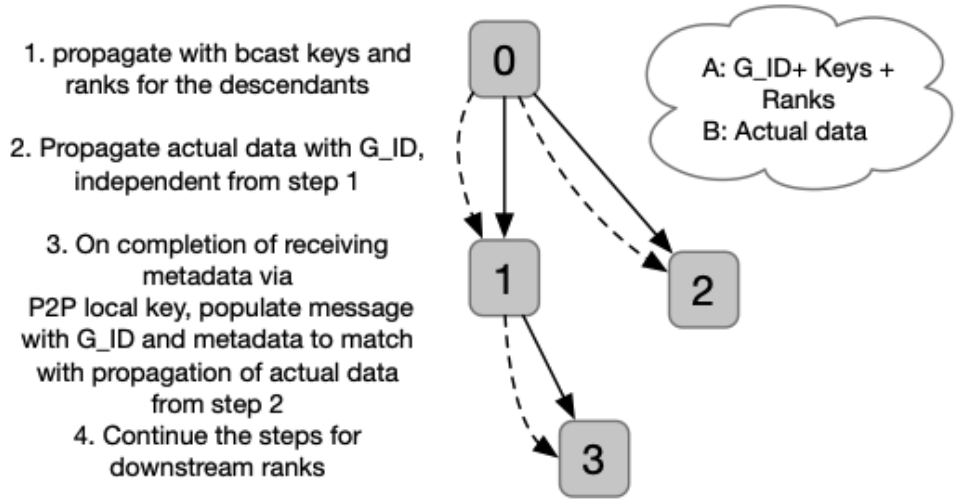


Figure 4.2: Two-step broadcast with meta-data transfer as the first, and data payload transfer as the second. They propagate as two separate flows but data reception call can only be matched when the meta-data is received and global ID is known.

between the root and each children) and participating ranks as the first step. This is the propagation of the metadata information representing the broadcast. In the PTG case, I can query the parameterized graph information to obtain this knowledge, but in the STF model, the parent needs to inform the descendants of the global knowledge coming from the root. This metadata is matched via the point-to-point data keys between the root and each of the descendants. For an intermediate node, once it has received the metadata, it can act as the root to continue the propagation of metadata. The actual data broadcast will use the global ID to progress as an independent second step. This is possible because after the first step completes and the global ID is known, the communication engine can match the data received using this ID. By populating the metadata received into the outgoing message, DTD broadcast can reuse PaRSEC collective implementation to continue message propagation using the selected topology.

4.3 Evaluation with the Cholesky and QR Factorizations

Cholesky and QR factorizations are classic linear algebra algorithms that are widely used for solving linear systems of the form $Ax = b$ with A having special numerical properties benefiting special algorithmic choices. For square matrix, their corresponding floating point operation (FLOP) counts are $\frac{n^3}{3}$ and $4\frac{n^3}{3}$. Their corresponding tile-based algorithms are listed in Algorithm 1 and 2, respectively. They both use four computational kernels that are successively applied on the trailing sub-matrix at each step, as illustrated in Figure 4.3 for matrices of 6×6 tiles at iteration $k = 2$. In practice, the implementation of these kernels relies on a BLAS library, such as MKL on Intel's x86 CPUs or SSL2 on Fujitsu A64FX CPUs.

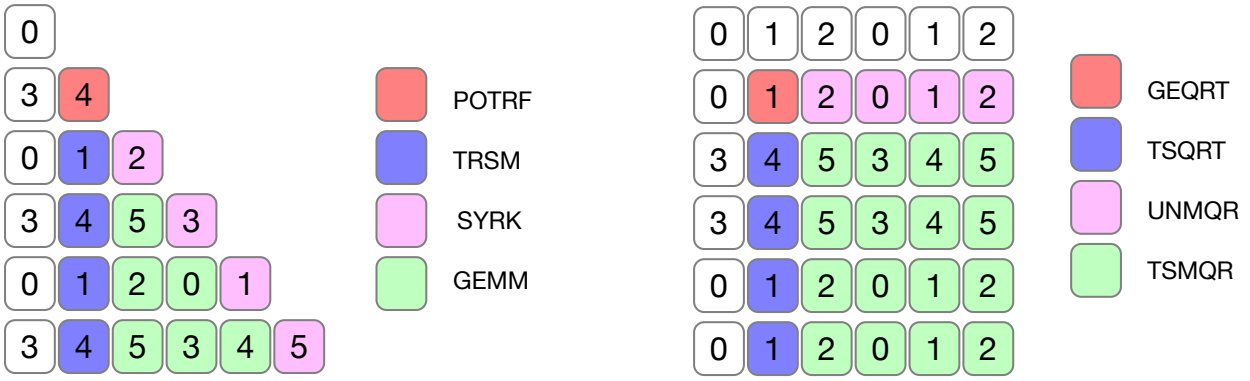


Figure 4.3: The four different kernels from Cholesky and QR respectively. Both runs on a 2X3 compute grid with 2-D block cyclic distribution. For QR, a super-tiling of 2 is used on the grid row to reduce cross node P2P communication.

Algorithm 1: Pseudo-code of Cholesky Factorization.

```
1 for k = 0 to NT − 1 /* Panel Factorization (PF) */
2     POTRF(C_kk^RW)
3     for m = k + 1 to NT − 1
4         TRSM(C_kk^R, C_mk^RW)
5     for m = k + 1 to NT − 1
6         SYRK(C_mk^R, C_mm^RW)
7     for m = k + 2 to NT − 1 /* Trailing Submatrix Update */
8         for n = k + 1 to m − 1
9             GEMM(C_mk^R, C_nk^R, C_mn^RW)
```

Algorithm 2: Pseudo-code of QR Factorization.

```
1 for k = 0 to NT − 1
2     GEQRT(C_kk^RW, T_kk^W)
3     for n = k + 1 to NT − 1
4         UNMQR(C_kk^R, T_kk^R, C_kn^RW)
5     for m = k + 1 to MT − 1
6         TSQRT(C_kk^RW, C_mk^RW, T_mk^W)
7         for n = k + 1 to NT − 1
8             /* Trailing Submatrix Update */
9             TSMQR(C_kn^RW, C_mk^R, T_mk^R, C_mn^RW)
```

4.3.1 Modifications to the user code

STF model provides a simple-to-use programming interface, but as demonstrated before and later in this study, the task graph overhead will significantly increase as we increase the problem size because the number of tasks is proportional to the problem size when the tile size remains fixed. As a result, the graph trimming is a required step to include in order to achieve good performance at large system and problem scales. Based on the tiled algorithm

of Cholesky and QR, here I describe how to both trim the task graphs and to incorporate explicit broadcast operations into the algorithms.

To ensure the correctness of the algorithm, the sender side needs to insert all the remote descendant tasks and on the receiver side, the remote data provider task needs to be inserted as well. For the Cholesky factorization without broadcast, this means that all TRSM tasks need to be inserted on the POTRF task node, and each of the nodes in the current panel need to insert the remote POTRF task in order to receive input data. On the receiving nodes, this means that other remote TRSM tasks can be trimmed (Figure 4.4, Left). Similarly, for the connections between TRSM and GEMM, each TRSM needs to insert all the GEMMs that are in the same row, as well as the GEMM tasks in the reflective column. On the receiver side, all the GEMM tasks will need to insert the two TRSM tasks from the given row/column. With an explicit API call to a user-level broadcast added, the expression of the program is changed. The destination ranks are iterated to create the metadata, and, as a result, the broadcast operation itself (yellow tasks in Figure 4.4, Right) can serve as the connection between the sender task and receiver tasks and we don't need to insert the tasks on the other side of communication exchange, thus simplifying the trimming code.

For the QR algorithm, it has a tighter set of data dependencies between the tasks, where each row has data dependency on the previous row. As a result, for a trailing task TSMQR, we need to discover the TSQRT on that row as well as the UNMQR task or the previous rows' TSMQR task in order to correctly obtain the input data. In the case of 2-dimensional block cyclic data distribution with $P \times Q$ number of nodes (usually with super-tiling on P to reduce row level communication frequency), we only need to insert $(P+Q)/(P \times Q)$ number of the original TSMQR tasks. For broadcast operations, the opportunities are limited in the QR algorithm, as the row-by-row updates naturally translate to point-to-point operations. The only possible broadcasts are the propagation of panel data across a given row of Q processes, either for GEQRT to UNMQR, or TSQRT to TSMQR (Figure 4.5).

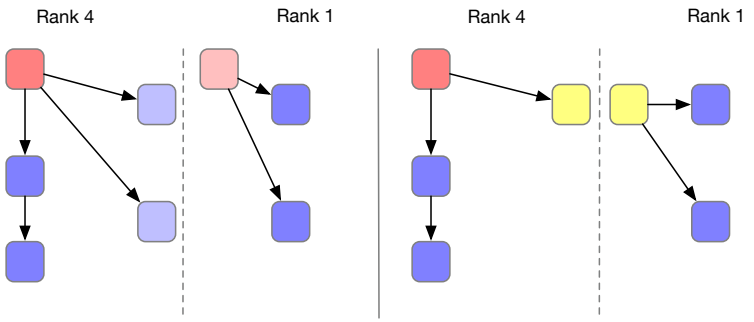


Figure 4.4: Left, trimmed task graph without broadcast call; Right, explicit broadcast call to propagate POTRF data. Color scheme and data distribution follows that from Figure 4.3. Lighter red and purple represent remote tasks, yellow represents broadcast task. Data dependency between TRSM and GEMM omitted.

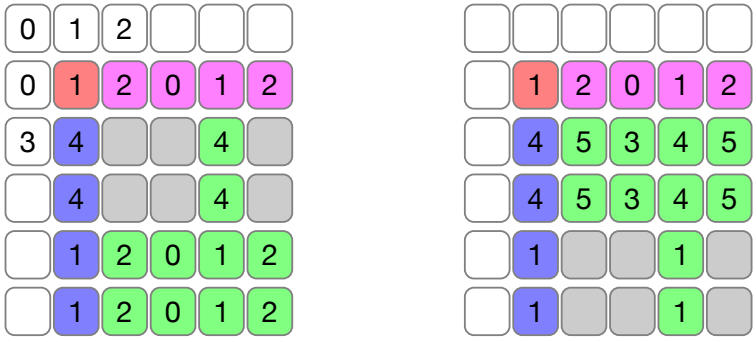


Figure 4.5: Since only the TSMQR tasks are of order $O(N^3)$, we can insert all the other tasks in all the nodes while inserting TSMQR only on ranks that are in the same row or column of the current panel tasks. Figure on the left, shows the situation for tasks inserted on rank 1, while figure on the right is for tasks on rank 4.

4.3.2 Qualitative Analysis

The major appealing factor for the STF model is that it is easy to use. Indeed, one can simply write the two algorithms following the pseudo-code with PaRSEC DTD and it will work out-of-the-box. The issue is that in order to obtain good performance and to avoid the overhead of traversing the entire task graph, the user needs to include many conditionals in the user code to evaluate whether we should insert a given task. Here, I will argue that this modification at the user level is not insignificant, rendering the STF model complicated to use (in some ways similar to the SPMD model). This is in contrast to previously brought up suggestions that this modification is easy and can be hidden. For data users, it can insert all the relevant remote tasks that will produce this data, but for the data writer tasks, the algorithm writer needs to be aware of the users of output data tasks, and will need to insert those reader tasks, correspondingly. In the case of Cholesky and QR factorizations, it is tractable, but when the algorithm becomes more complicated instead of trivially nested for-loops, we can imagine that trimming can produce very error prone codes.

This goes back to some of the difficulties in writing algorithms using PaRSEC PTG. One is that you need to write in a domain specific language, but more importantly, the user needs to think of the algorithm in terms of the DAG and to specify the data dependencies between the tasks explicitly. This includes all the data' input and data' output links of each of the tasks. But to trim the graph correctly, the algorithm writer is essentially expressing the same information as with PaRSEC PTG. As a result, I view the trimming optimization as trying to express the same information on these two interfaces, and they only differ as to when and where the users supply additional information about the relationships between tasks.

4.4 Performance Results and Analysis

4.4.1 Description of HPC systems

I implemented the new features in PaRSEC based on the branch from Nov, 2020. All the results presented in this paper use the IEEE 754 double precision variants DPOTRF and DGEQRF for Cholesky and QR factorizations, respectively. In this paper, I ran the experiments on two systems:

- **Shaheen II**, a Cray XC40 supercomputer with 6,174 nodes composed of two-socket 16-core Intel Haswell (AVX2) processor and 128GB of main memory, using the Cray Aries network interconnect. I used Cray MPI and Intel programming environment (MKL).
- **Fugaku system**, a Fujitsu ARM (SVE) system with A64FX nodes composed of four 12-core core memory groups (CMGs) and 32GB of main memory, connected through the TofuD interconnect. I used Fujitsu MPI and SSL2 libraries.

4.4.2 Broadcast Benchmark Performance

I measured the message transfer time of one broadcast operation and compared it with the scenario of using the default DTD point-to-point (P2P) to evaluate the benefit from doing the broadcast. I varied the number of nodes, as well as the size of the data I was sending to match the amount with the square tile from the Cholesky and QR factorizations. The two machines have different networks: Shaheen II uses Cray Aries Interconnect with Dragonfly topology with bandwidth of around 10 GB/s. Fugaku uses TofuD interconnect, a 6D torus topology, from Fujitsu with bandwidth of around 40 GB/s.

The results are shown in Figure 4.6. On Fugaku machine, the result followed our expectations, namely the broadcast can propagate the data equally or faster than P2P where the root sending the data to each of the descendant. Also, as the message sizes became bigger, it took longer to complete the entire data transfer.

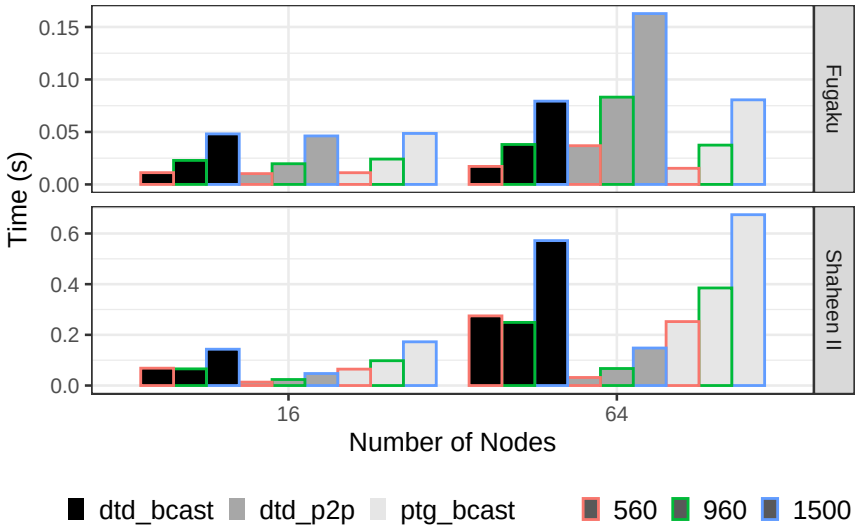


Figure 4.6: Benchmark of a broadcast operation for sending a square tile of double precision floating point values. I tested on two sets of nodes, and varied the message data size. For comparison, I have the default DTD P2P, the proposed DTD broadcast and finally the broadcast utilized in PTG (the two shared the same mechanism).

But on Shaheen II, the point-to-point version could finish faster than the collective version for all message sizes. The reason for this is not known, but my hypothesis is that the difference in network topology alleviated the bottleneck of the P2P from the root node. But in real applications, the situation can be complicated and network state could change. For example, the computation threads can create memory contention and reduce network performance [49], and when employing broadcast, the operation can share the network usage across the nodes, instead of relying on a single root node for data transfers, potentially saturating a single node's outflow bandwidth.

4.4.3 Experiment performances

As the baseline to compare our achieved performance, I also ran the ScaLAPACK version of Cholesky and QR factorizations provided by the math libraries on the respective system (MKL from Intel on Shaheen II and SSL2 from Fujitsu on Fugaku). ScaLAPACK is a widely used library that provides distributed version of common linear algebra operations and its optimized versions are provided by vendors.

Based on the previous descriptions, I implemented different versions of Cholesky, with graph trimming, broadcast operation, or a combination of both. I compared the performance of the different flavors of these algorithms with the original DTD as well as PTG implementations from DPLASMA. And for the QR factorization, I have the trimmed-only version as well as trimming with broadcast version (since the broadcast-only version shows no improvement, it is not shown here). I obtained results for matrices varying in size from 100K up to size of 600K, using two different tile sizes. Finally, I show the scalability of the implementations by running on 256 and 512 nodes.

Shaheen II Results

The results from Shaheen II for the Cholesky and QR factorizations are shown in Figure 4.7 and Figure 4.8 for 256 and 512 nodes, respectively. The black lines are for the results from ScaLAPACK with one MPI rank per core, block size of 64. I tested two different tile sizes,

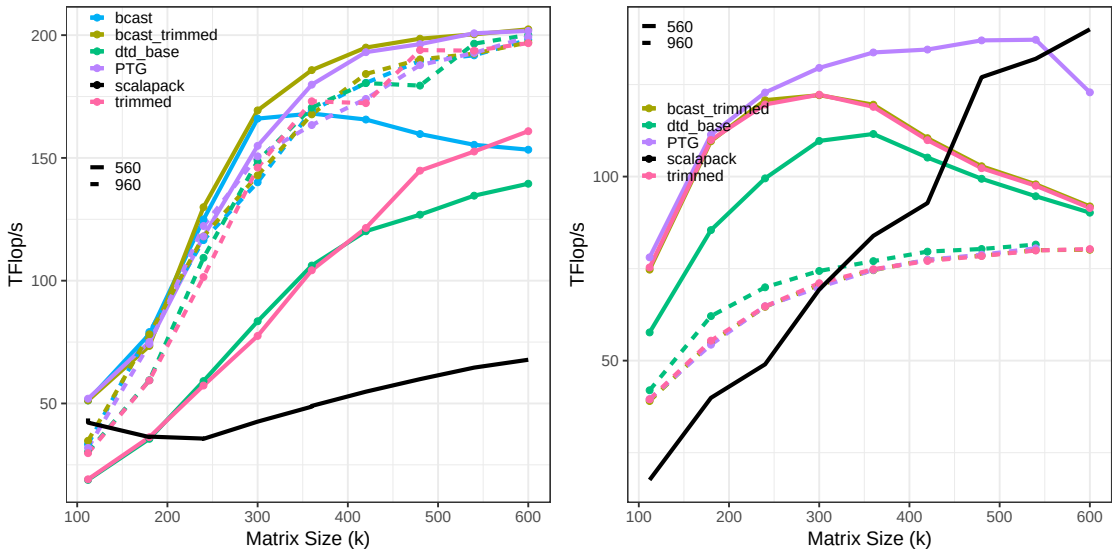


Figure 4.7: Performance on Shaheen II, 256 nodes. Left: Cholesky, Right: QR

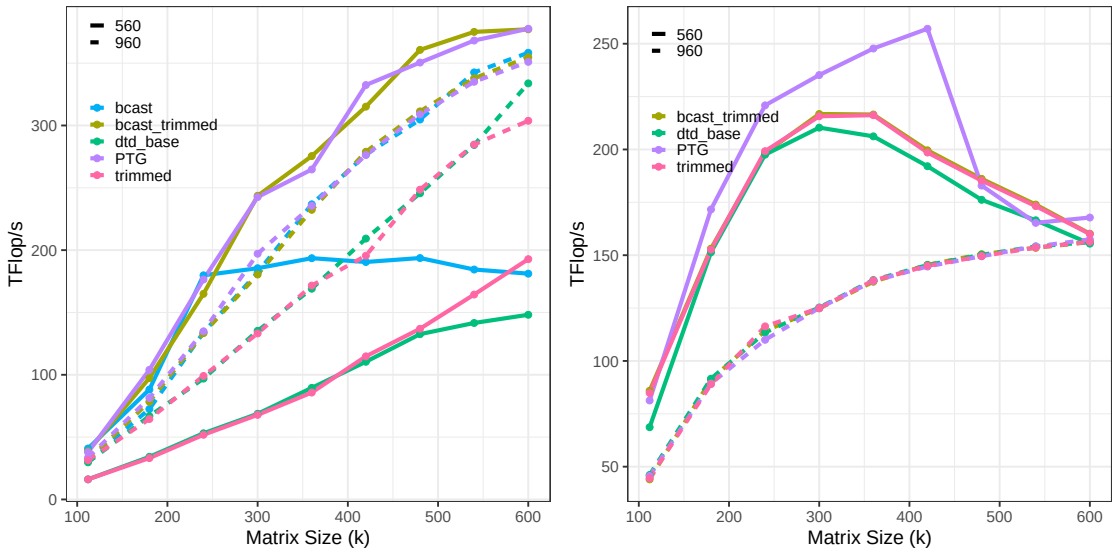


Figure 4.8: Performance on Shaheen II, 512 nodes. Left: Cholesky, Right: QR

which affects the number of available tasks as well as the degree of parallelism. First thing I would like to point out is that overall, the performance from the runtime system-based implementations were better than the ones from ScaLAPACK, this is especially true for the Cholesky factorization. Performance on a single node for DPOTRF is around 860 GFLOP/s, meaning that assuming the perfect scaling, we would have reached 220 TFLOP/s with 256 nodes. But in terms of the maximum performance achieved, the Cholesky factorization could reach a higher efficiency than QR, which was likely due to a larger degree of parallelism. ScaLAPACK QR performs very well as the problem size increases.

Second thing to note is that for the tile-based algorithms, the tile size needs to be tuned in order to obtain good performance. And the optimal tile size depends on the interface we used and the balance between computation, communication, and runtime overheads. For most cases, the tile size of 560 is better than 960, but for the QR implementation this is dependent on the problem size, as the matrix sizes increase, using tile size 560 we observe performance degradation instead of stabilization, while a larger tile size of 960 shows performance improvements. I suspect this came from the overhead of task insertion and management. This explains why the trimmed version of QR is faster than the base DTD version when tile size was 560, with the reduction of task analysis overhead.

The two user-level features, that I added, provided various degree of performance improvements. When the tile size was 960 instead of 560, since the number of tasks is cubic with the number of tiles, we could have had 5-fold reduction in the number of tasks. As a result, the graph trimming is not providing as much of an impact as with the case of tile size 560 (trimming can reduce the number of inspected tasks by an order of magnitude). Adding broadcast for the Cholesky factorization provided a good performance gain in the case of tile size 560, but when combining the two, we could get the biggest boost in performance, even better than the PTG version of Cholesky implementation. This two features also changed the optimal tile size for Cholesky from 960 to 560. Tile size of 560 is big enough to obtain good performance from Intel's MKL and the bigger the tile size was the more likely it was to compensate for the higher overhead from base DTD overheads.

The interesting thing is that in Figure 4.6, the P2P is faster than broadcast but results from Figure 4.7 and Figure 4.8 showed improved performance for Cholesky factorization. Other authors indicated [49] that computation can reduce network bandwidth due to memory contention, I think that the actual P2P bandwidth during Cholesky factorization is less than the benchmark measurement. By spreading the message propagation across the participating nodes via broadcast, it could remedy the network degradation and improve overall performance.

Fugaku Results

Similarly, the results from Fugaku are in Figure 4.9 and Figure 4.10 for 256 and 512 nodes, respectively. I observed generally the same trends as in the result from Shaheen II, with good scalability on both 256 and 512 nodes. On one node of Fugaku, I could obtain DPOTRF results of around 1700 GFLOP/s, meaning the result from 256 nodes would have had a ceiling of 435 TFLOP/s. The single node base is lower than other SSL2 results due to an issue calling SSL2 math library from multiple threads, and I had to disable the sector cache optimization to complete the runs. One difference is that the base DTD Cholesky was performing much worse relative to the ones from Shaheen II. And correspondingly, a much smaller effect was observed from just adding broadcast. With 48 cores instead of 32 from Shaheen II, insertion efficiency might have had a larger factor in order to saturate all the cores. And the trimming in this case provided a larger degree of relieve to this bottleneck. With the two features combined, I actually obtained significantly better result for the Cholesky implementation in comparison with the PTG version from 512 nodes.

For the QR implementation, the broadcast-only version showed a minimum improvement effect, since the dependencies are tighter than for the Cholesky one. Although trimming can provide a small performance boost, I still need to increase the tile size to further reduce the overhead, this in turn diminishes the effect of trimming. In summary, further profiling is needed to understand the exact reason for the performance drop and where the limiting factors were coming from for DTD version.

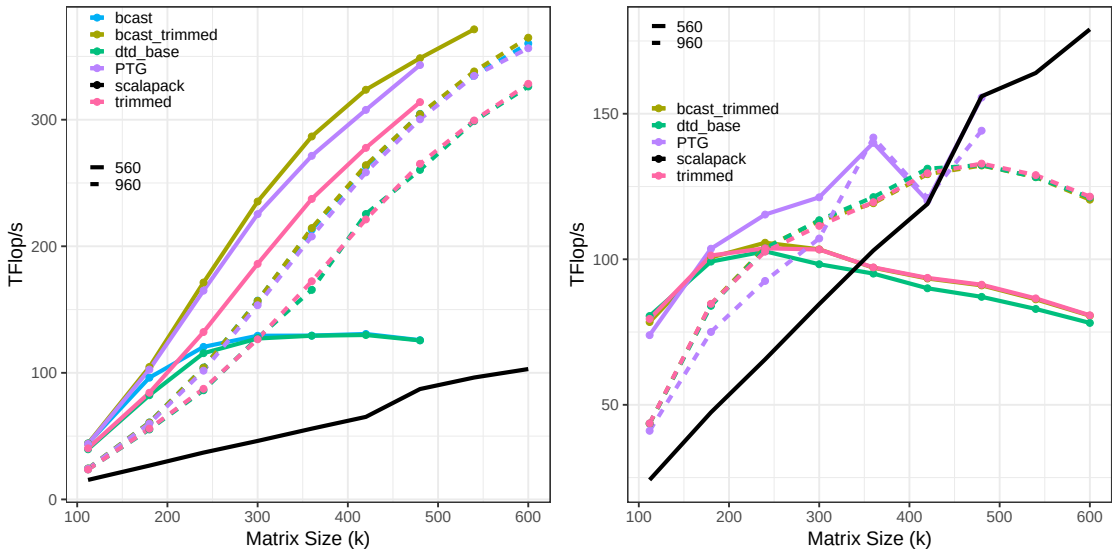


Figure 4.9: Performance on Fugaku, 256 nodes. Left: Cholesky, Right: QR

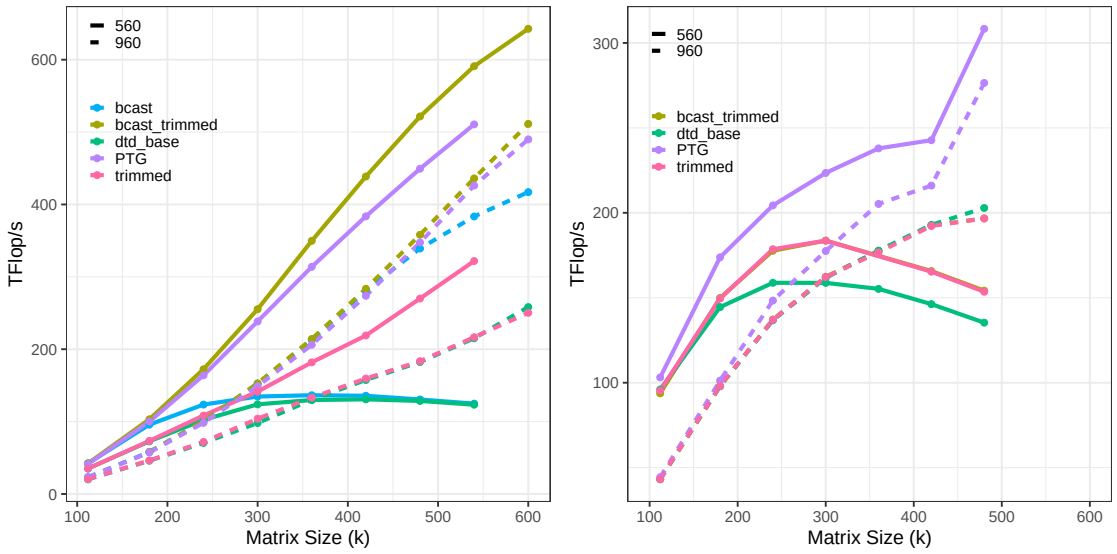


Figure 4.10: Performance on Fugaku, 512 nodes. Left: Cholesky, Right: QR

4.5 Conclusions

In this work, I introduced two new features to PaRSEC DTD interface, namely user-level graph trimming and broadcast operations. I demonstrated the user-level changes needed to use these two features with the Cholesky and QR factorizations. This experience indicated that although it is relatively easy to trim the graph when the algorithm is simple, it can be complicated when the user needs to know the exact dependencies among the tasks and will lead to messy and error-prone user codes. I also showed that with these two added features, I can significantly improve the performance of Cholesky, while providing only modest improvement for QR. From both the usability and performance perspective, I showed that the STF interface still has a lot of opportunities for improvement, but it will likely limit the usability of the original interface and create difficult to maintain and debug user code.

Exploring the reason for QR performance drop is beyond the scope of this programming model study, but further investigation is needed to understand the bottleneck in those cases and the likely overheads still in the runtime implementation, and the applicability of STF interface to a wider range of scientific applications.

Chapter 5

Extension to PTG - Testcase with Communication Avoiding 2D Stencils

5.1 Overview

Stencil computations are a common pattern in a variety of scientific and engineering simulations based on discretizations of partial differential equations (PDE), and they constitute a key component of many canonical algorithms such as stationary iterative methods involving sparse linear algebra operations, for example Jacobi iteration [61], as well as non-stationary and projection methods employing geometric multigrid [Hackbusch, 109] and Krylov solvers [105, pp. 241-313]. They are routinely used to solve problems that arise from the discretizations of PDEs [60]. Stencil codes can be characterized as having high regularity in terms of the data structures and the data dependency pattern. However, they also exhibit low arithmetic intensity and, as a consequence, the available memory bandwidth required for data movement is the limiting factor to their performance. To exacerbate these issues, the recent trends in the hardware architecture design have been skewed towards ever-increasing number of cores, widening data parallelism, heterogeneous accelerators, and a decreasing amount of per-core memory bandwidth [114]. The prior work on optimizing the stencil computations has mostly focused on techniques to improve the kernel performance

within a particular domain such as cache oblivious algorithms, time skewing, wave-front optimizations, and overlapped tiling [23]. On modern systems, these algorithmic classes must be recast to overcome the geometrically growing gap between processor speed and memory/network parameters, in particular, CPU/GPU speeds have been improving at 59% per year while the main memory bandwidth at only 23%, and the main memory latency decreased at a mere 5.5% [63]. Given the widening gap between computation speed and network bandwidth, a systematic study of performance of stencils on the distributed memory machines is still relevant. Especially the optimization of communication is lacking.

In the recent years, a number of runtime systems and new programming models have been developed to facilitate application development by separating the domain science and the tuning of the performance, leveraging the respective strengths of domain scientist and runtimes. The runtime abstraction layer invariably comes with a certain amount of overhead that can be overlapped with enough task granularity [99]. But with the low arithmetic intensity of most of these types of kernels, a viable runtime solution needs to be able to maintain efficiency with fine grained tasks. STF model with dynamic task graph analysis as a result is not suitable, while from our results in Chapter 3, PTG is a good platform for the development of a set of stencil-like operations.

As a result, in this pilot project, I adopted PaRSEC PTG to abstract away the MPI communication across nodes, and experimented with communication-avoiding (CA) techniques to further reduce the communication overhead that is the limiting factor in stencil computation. I use the 2D five-point stencil as the test case, and compared the performance of three implementations: PETSc, base-PaRSEC and CA-PaRSEC. I investigated extensively the interplay between memory bandwidth, computation speed, and network latency/bandwidth on stencil code performance. I demonstrated that under some reasonable assumptions on workload and system configurations, performance improvements between 33% and 57% were obtained on the two tested machines when communication avoiding scheme is adopted into PaRSEC runtime.

5.2 Related Work

Stencil codes research has mostly focused on optimizing the kernels [122] or domain specific systems that can generate efficient kernels automatically [114] and generating code that can utilize GPUs efficiently [121]. The authors not only optimized the kernel [23], but also implemented the communication avoiding technique directly within their compiler framework. Here, instead of combining everything within one compiler system, I investigate delegating the internode communication to runtime system instead of combining both communication hiding and communication avoiding at the runtime system's level. Communication avoiding (CA) methods (or *s-steps methods*) themselves represent a mature concept [47] and many Krylov subspace solvers were built with this idea [73] [119]. The numerical properties of such approaches, including monomial basis and matrix powers kernels (MPK), are out of scope of this paper and we mainly focus on the feasibility and benefits of having CA ability implemented within a runtime infrastructure. Applications of communication avoiding techniques to numerical linear algebra algorithms have also been studied and performance improvement demonstrated [101] [59]. In particular, authors also studied the interaction between communication computation overlap with communication avoiding technique programmed with Unified Parallel C (UPC) [59], a partitioned global address space (PGAS) model.

In this study, I adopt the runtime-based approach and study extensively the benefits of runtime system's benefits to provide better communication-computation overlap, and the opportunity for further improvement via communication avoiding scheme for stencil computations and for Sparse Matrix-Vector multiplication (SpMV) in general [7] [117]. To the best of my knowledge, this is the first time the combined approach addressed distributed systems in a comprehensive manner. My goal was to demonstrate the feasibility of such a software infrastructure for a broad range of numerical algorithms.

5.3 Background

5.3.1 Stencil Problem Description

Scientific simulations in diverse areas such as diffusion, electromagnetics, and fluid dynamics use PDE solvers as the main computational component. These applications commonly employ discretization schemes such as finite-difference or finite-element techniques. During the solve, they sweep over a spatial grid, performing computations involving nearest-neighbor grid points. Such compute patterns are called stencils. In these operations, each of the regular grid points is updated with weighted contributions from a small subset of neighboring points in both time and space. The weights represent the coefficients of the PDE discretization for each data element. Depending on the solver, these coefficients may be the same across the entire grid or differ at each grid point. The former is a constant-coefficient stencil while the latter – a variable-coefficient stencil. The range of solvers that often employ stencil operations includes simple Jacobi iterations [61] to complex multigrid [109] and adaptive mesh refinement (AMR) methods [44].

Stencils can operate in different dimensions of the domain, having different iterations and coefficient types. In this work, I use the classic Jacobi iteration to solve the Laplace's equation, which means that I have one input grid $X^{\ell-1}$ (for reading) and one output grid X^{ℓ} (for writing), and the update between the two is in the form of:

$$\begin{aligned} x^{\ell}_{i,j} =& w_{0,0} \cdot x^{\ell-1}_{i,j} \\ &+ w_{0,-1} \cdot x^{\ell-1}_{i,j-1} + w_{0,1} \cdot x^{\ell-1}_{i,j+1} \\ &+ w_{-1,0} \cdot x^{\ell-1}_{i-1,j} + w_{1,0} \cdot x^{\ell-1}_{i+1,j} \end{aligned} \tag{5.1}$$

I used the more general form of weights which will give me the consistent FLOP/s count of $9n^2$ for all implementations (5 multiplications and 4 additions). A diagram of the Jacobi iteration scheme is shown in Figure 5.1.

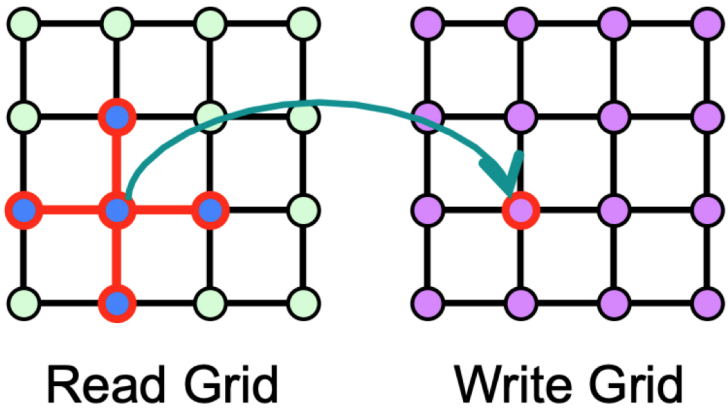


Figure 5.1: Common illustration of the Jacobi update scheme [44].

5.3.2 Communication Avoiding Approach

With the increasingly widening gap between computation and communication, modern algorithms should try to minimize communication both within a local memory hierarchy and between processor or nodes. And this is especially true for SpMV, stencil operations that are bound by the speed of the memory system and network interconnect. The key idea from Demmel et al [47] is to perform some redundant work locally that would relieve the bottleneck of communication latency. Two new algorithms were introduced, PA1 (depicted in Figure 5.2) and PA2 as they described in the paper, where PA1 is the naïve version while PA2 minimizes the redundant work but might limit the amount of available overlap between computation and communication. My implementation follows the PA1 algorithm.

As an example shown in Figure 5.2, ghost region of 3-layers are used to store remote data. This allows the local grid to perform Jacobi updates up to three time steps on the local data (white points) with replication of work from remote points (outer red points used to update inner red points). By performing redundant work, I reduce the frequency of communication thus the effect of network latency.

PaRSEC runtime system by design allows computation and communication overlap, by incorporating the communication avoiding scheme into the task-based implementation of stencil operations, I believe such an infrastructure can further improve its performance.

5.4 Implementations

5.4.1 Standard Implementation with PETSc

PETSc is a suite of data structures and routines for the scalable (parallel) solution of scientific applications modeled by partial differential equations [21, 20, 22]. It provides many of the mechanisms needed within parallel application codes, such as simple parallel matrix and vector assembly routines that allow the overlap of communication and computation. In addition, PETSc includes support for parallel distributed arrays useful for finite difference methods.

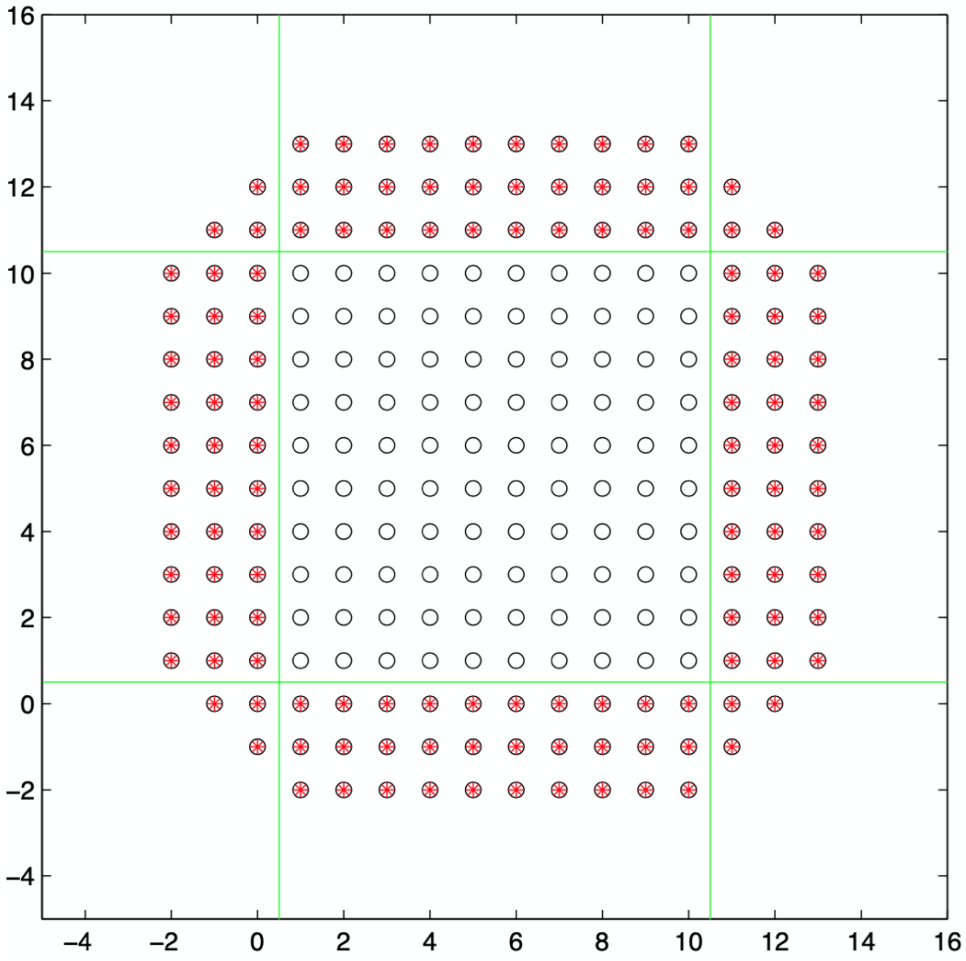


Figure 5.2: The 2D five-point stencil operation using PA1 algorithm on a 10-by-10 grid, having a step size of 3 as illustrated in the original report [Demmel et al.]. For a single processor with the projected view. Red asterisks indicate remote values that need to be communicated.

In order to mplement Jacobi iteration in PETSc, I simply expand the 2D compute grid points into 1D solution vector, and the corresponding 5 points stencil update is expressed as a sparse matrix multiplication by the vector of unknowns. By default, PETSc partitions the sparse matrix by rows with each process assigned a continuous block of matrix rows. To perform the updates, I keep two solution vectors that are swapped within the main for-loop up to a specified iteration count. Since PETSc is a mature and widely used package, the result will serve as the baseline to compare against our PaRSEC performance.

5.4.2 Task-based Implementation in PaRSEC

Baseline PaRSEC Version

The first version follows the formulation of the Jacobi iteration, with data partitioned into 2D blocks over the 2D computation grids. Then the data on each node are divided into tiles that each task will operate on. By providing this extra level of decomposition, only the tasks that have neighbor tiles on a remote node will incur communication, while the inner tasks can still be processed with the remaining workers. Each tile will have an extra ghost region used for data exchange between tasks.

Figure 5.3(a) provides the diagram that depicts the implementation of the baseline stencil code. The dashed line indicates the node boundary, the 2D blocked data distribution ensures that the surface to volume ratio is minimized and we have minimal remote communication. Each tile had the same size and also each one had an extra ghost region for copying neighbor tiles' value. Since the implementation involves a 5-point stencil, the figure indicates that there are three possible dependencies cases, for the interior tasks, all the neighbors are local to the task and can simply copy the memory into the ghost region. For the tiles on the boundaries or corners, one or two remote data transfers are needed. Otherwise, the computation kernel itself is very straightforward as it simply loops over the grid elements within a tile and apply the updates during the iteration.

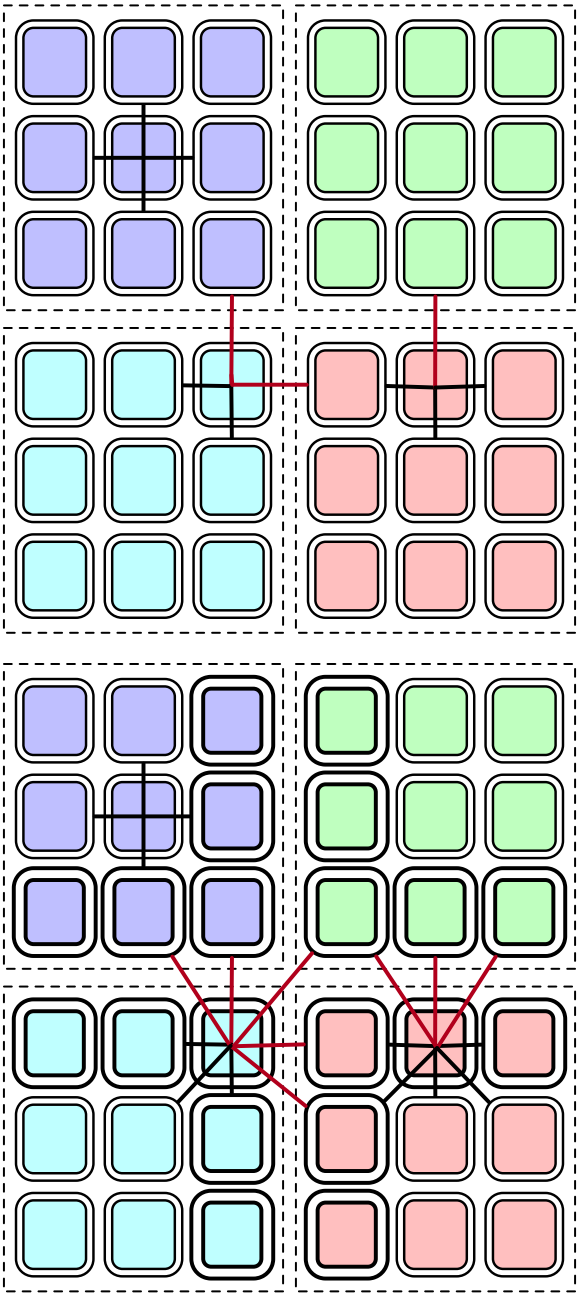


Figure 5.3: Top) Diagram of the baseline version of the PaRSEC implementation. Three possible task locations and their data dependencies are shown. Black line indicates within node data copy while red line indicates remote communication., Bottom) Diagram of the communication avoiding version PaRSEC implementation. Three possible task locations and their data dependencies are shown. Black line indicates within node data copy while red line indicates remote communication. The boundary tiles will have a bigger ghost region to accommodate the extra layers of remote data.

Communication Avoiding PaRSEC Version

The second version I adopted used the communication avoiding scheme whereby I trade extra computation for less frequent communication exchanges. Figure 5.3(b) has the overall structure very similar to the baseline version, and the interior tasks have the same task dependency as in the baseline version. But for boundary tiles, in addition to the four neighbors, I need to buffer additional data from the four corner neighbors due to the additional steps of remote computation that need to replicate locally. Since I still don't have the support at the runtime level, I implemented the logic directly as a proof of concept, specifying the dependencies directly at the PTG level. Conditions are provided to test whether the task is operating on a boundary tile, and whether I need to communicate at a given iteration. And the corresponding logic in the body of the task to decide on the data I need to copy in and out, and the kernel it should call. Similar to the baseline version, the tiles that have all its neighbors local to the node would still have one layer ghost region for data exchanges since they do not need remote communication. But the boundary tiles have a ghost region of *steps*-layers as specified for the extra amount of data exchanged, and as a result, this version uses slightly more memory.

5.5 Experiments Results

5.5.1 Experimental Setup

To evaluate the benefits of implementing stencil operations with a runtime system and additionally the benefits of incorporating communication avoiding scheme, I consider the following properties of the problem and the characteristics of the machines:

1. Number of arithmetic operations and memory accesses per task;
2. The maximum achievable network bandwidth of the cluster's nodes and the memory bandwidth of a compute node;

3. Number of floating-point numbers communicated per processor, and the number of messages sent per processor.

Since I formulate the problem in the more generic version which performs 9 floating-point operations per grid point update and need to transfer 16 to 24 Bytes (read and write of double floating point numbers) of data depending on the size of tiles, therefore I will use the range of 0.37 to 0.56 as the arithmetic intensity. To measure the peak network bandwidth performance, I used the NetPIPE benchmark [110] and for the memory bandwidth performance, I used the STREAM benchmark [86].

As mentioned before, there are three versions of my stencil code implementation, one in PETSc and two in PaRSEC, with normal communication pattern and CA scheme respectively. I first compare the strong scaling performance of the three versions using PETSc as the baseline in order to have a better understanding of PaRSEC versions' performance. Then, I move on to adjust the step sizes and tune the execution time of the kernel (to simulate memory bandwidth utilization rate) to investigate the interplay between memory bandwidth and network communication on the overall performance. As the computer architectures continue to evolve, the results here should provide a guidance for future performance improvements that can be expected on stencil operations.

The experiments were run on two systems. First is an in-house cluster called NaCl that had a total of 64 nodes, each with two Intel Xeon X5660 (Westmere) CPUs, with the total of 12 cores spread across two sockets of each node and 23 GB of memory per node. The network switch and network cards are Infiniband QDR with a peak network rate of 32 Gb/s. The second system was Stampede2 system located at TACC: each node is equipped with two Intel Xeon Platinum 8160 (Skylake) CPUs with a total of 48 cores across two sockets of a node, and 192 GB of on-node memory. The interconnect was a 100 Gb/sec Intel Omni-Path network.

I used PaRSEC master branch from commit faf0872052, and PETSc release version 3.12. On NaCl, we compiled the code with GCC 8.3.0 C compiler and used Intel MPI 2019.3.199. On Stampede2, we compiled our code with Intel C compiler 18.0.2 and MVAPICH2 version

2.3.1. PETSc was compiled with all the optimizations enabled and used 64-bit integers for indexing. PETSc runs had one MPI process per core. For PaRSEC runs, I configured the system to have one process per node, with one thread dedicated for communication while the remaining ones were assigned to computational tasks. The nodes during the runs were arranged into square compute grid and the data tiles were allocated in a 2D block fashion to exploit the surface-to-volume effect.

5.5.2 Network and Memory Bandwidth Benchmark

STREAM benchmark was run on both systems utilizing all the cores on a compute node since, as the results show, a single core cannot saturate the memory interface. The results are shown in Table 5.1. The different STREAM modes vary in their arithmetic intensity: bytes transferred per FLOP computed. For simplicity, in the following I use the results from COPY operation as the achieved memory bandwidth.

The achieved bandwidth on NaCl and Stampede2 were 39.1 GB/s and 172.5 GB/s, respectively. The estimated arithmetic intensity is between 0.37 to 0.56 depending on data availability in cache. I expect the effective peak performance between 14.5 to 21.9 GFLOP/s and 63.8 to 96.6 GFLOP/s for the memory-bound stencil kernels under the assumptions of the roofline model [115].

I tested the network interconnect on both systems with the NetPIPE benchmark and obtained the performance results plotted in Figure 5.4). The effective peak network bandwidth on NaCl is about 27 Gb/s while on Stampede2 I could achieve up to 86 Gb/s. Given the size of our stencil tiles, it is unlikely to reach that peak bandwidth shown in Figure 5.4. The latency of the network was around 1 microseconds.

5.5.3 Tuning of Tile Size for PaRSEC Performance

Next, we measure the actual performance results of the base implementation on top of PaRSEC that runs on a single node (no network communication) with different tile sizes across all available cores. The results allow us to select a reasonable tile size for local

Table 5.1: STREAM Benchmark Results (MB/s) for NaCl and Stampede2.

System	Scale	COPY	SCALE	ADD	TRIAD
NaCl	1-core	9814.2	10080.3	10289.3	10271.6
NaCl	1-node	40091.3	26335.8	28992.0	28547.2
Stampede2	1-core	10632.6	10772.0	13427.1	13440.0
Stampede2	1-node	176701.1	178718.7	192560.3	193216.3

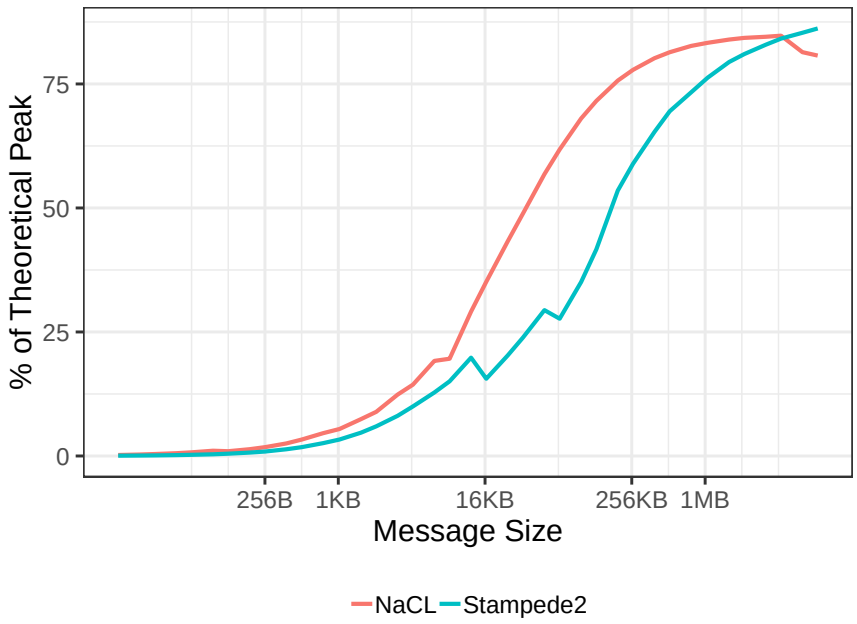


Figure 5.4: Network Performance from NetPIPE on NaCl and Stampede2 with theoretical peak of 32Gb/s and 100 Gb/s, respectively.

computation. They also tell us the gap between the performance of the native kernel and the peak memory bandwidth performance to provide us with a reference point for distributed runs.

In Figure 5.5 there is a certain range of tile sizes that allows us to obtain reasonable performance levels. For the NaCl system, the tile sizes of 200 to 300 will result in 11 GFLOP/s while on Stampede2 the tile sizes 400 to 2000 will yield close to 43.5 GFLOP/s. Given the fact that I did not optimize the kernel, the obtained result is acceptable for the circumstances but is still not close to the peak memory bandwidth level indicated in the previous section. Therefore, in the following experiments, I will run PaRSEC versions with the tile sizes in the optimal range obtained from the local-only runs.

5.5.4 Comparing Strong Scaling Performance

Figure 5.6 shows the strong scaling speed up of the three stencil implementations when using optimal single node performance as baseline. All three maintain good scalability levels, and the PaRSEC versions can achieve twice the performance of PETSc. This performance advantage can be partially explained by the specific SpMV formulation used by PETSc, since instead of having the weight matrix be represented with only 5 numbers, the update will involve both sparse matrix indices and the corresponding values – other versions represent the indices implicitly as small-value constants in the array indexing code. This, at the very least, doubles the number of memory loads (64-bit integers) that are needed for the same amount of floating point operations (64-bit floating-point.) Finally, I notice that the two PaRSEC versions are almost indistinguishable from each other, indicating that the communication avoiding approach is not very helpful for 5-point 2D stencils on the tested machines as long as the kernel is bound by the local memory bandwidth instead of being sensitive to the network bandwidth.

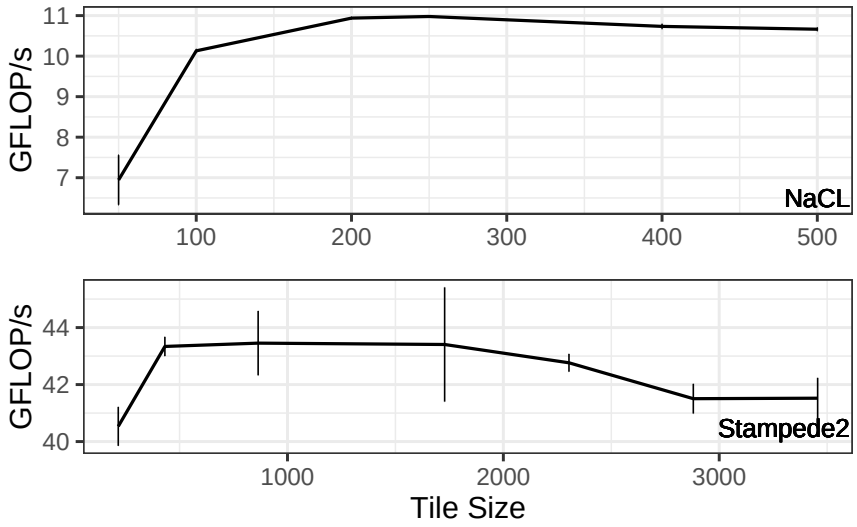


Figure 5.5: Shared memory PaRSEC base version performance for a given tile size; (top) NaCl with problem size 20K, (bottom) Stampede2 with problem size 27K.

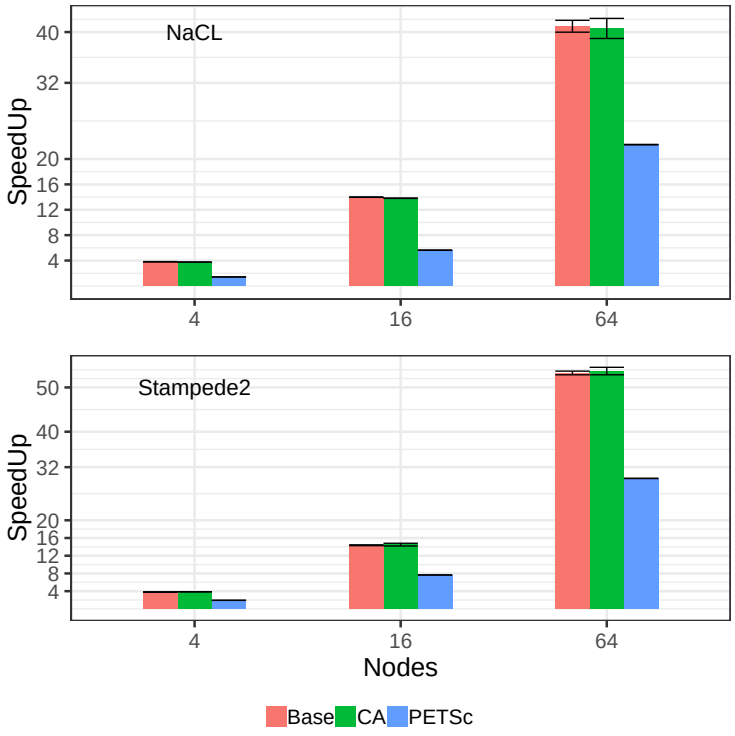


Figure 5.6: Strong scaling speed up over single node baseline PaRSEC; (top) NaCl result with problem size 23k, tile size 288; (bottom) Stampede2 result with problem size 55k, tile size 864, running for 100 iterations. Steps size of 15 is used for CA version.

5.5.5 Tuning of Kernel Time and Performance Impact of Communication Avoiding Scheme

To further investigate the potential benefits of communication avoiding schemes on a distributed problem, I tested the case where the memory system is much faster or the computational kernel was optimized to better utilize the memory bandwidth (for example, the case of a local communication avoiding scheme that reduces slow memory accesses). To simulate this, I set a ratio parameter r, so that only $(r \times m_b) \times (r \times n_b)$ portion of the tile gets updated, which effectively reduces the memory access thus speedup the kernel execution by an adjustable ratio r (m_b and n_b are the rows and columns number of a tile, respectively). Figure 5.7 shows that in such case, communication avoiding can provide a decent amount of improvements, for example the NaCl 16 nodes case we can see a 57% improvement if the kernel time is small. While on 16 Stampede2 nodes, a moderate 18% improvement can be observed in that case. The fast kernel times I assume here is quite realistic as well. Based on STREAM memory bandwidth test result, 0.6 ratio kernel performance is similar to reaching around 80% of STREAM bound. According to recent study [122], it is an efficiency level achieved with optimized kernel.

The step size affects how often the boundary tiles communicate with each other, the size of the message and the amount of available tasks can be enabled in this interval. Although in my implementation, it had no impact on the boundary tasks' execution time since I simulated the kernel time without the extra computation. The interplay between step size and kernel execution time is complicated, but the optimal step size can be searched via experimental runs. Figure 5.8 indicates that if communication avoiding scheme can improve performance over the base version, the step size needs to be tuned to get the best possible speedup on the tested system.

5.5.6 PaRSEC Profiling of the Two Versions

To validate that the communication avoiding versions indeed reduce the network latency thus reducing the cores idling time, I used PaRSEC's profiling system to record the execution trace

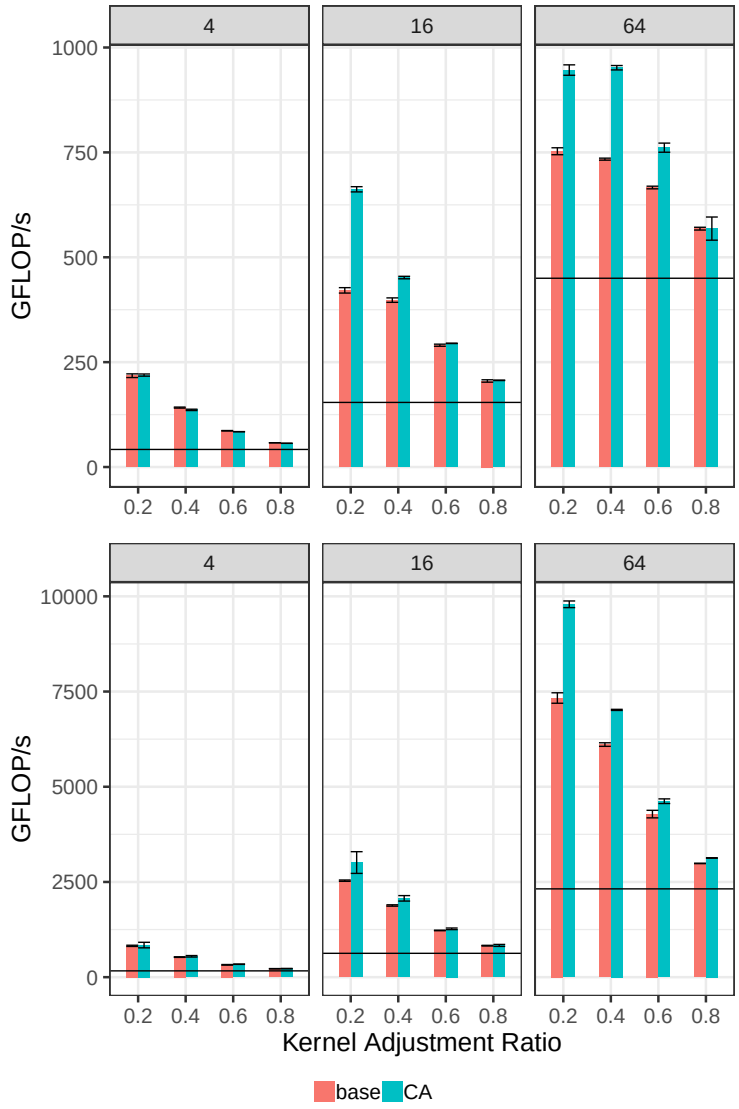


Figure 5.7: Tuned kernel performance: (top) NaCl result with problem size 23k, tile size 288; (bottom) Stampede2 result with problem size 55k, tile size 864, running for 100 iterations. Steps size of 15 is used for CA version. Running on 4, 16 and 64 nodes with squared compute grid. The ratio r indicates the ratio of m_b and n_b of tile being operated on, namely r^2 of the original number of points in a tile. Black lines indicate the base PaRSEC with original kernels' result.

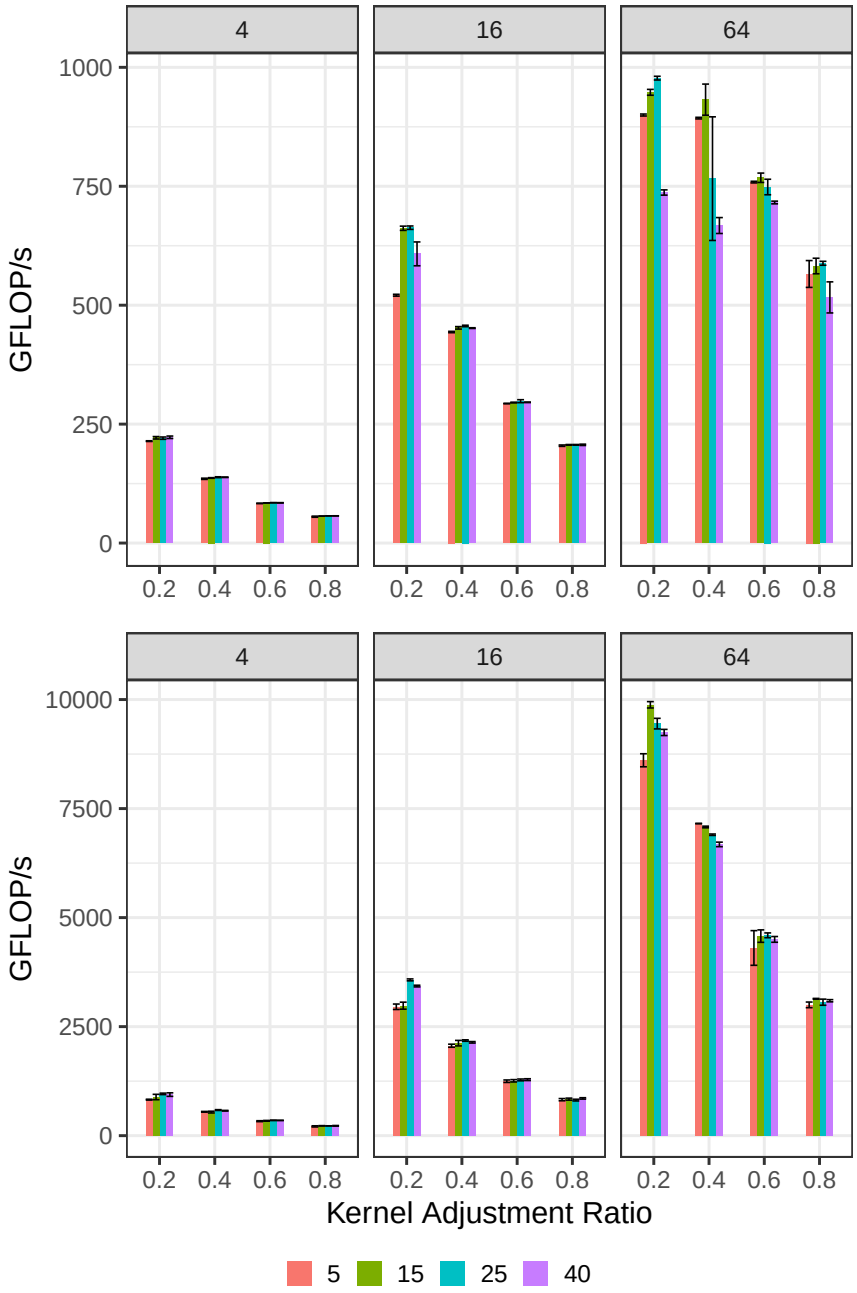


Figure 5.8: Tuned step size performance: (top) NaCl results with problem size 23k, tile size 288; (bottom) Stampede2 results with problem size 55k, tile size 864, running for 100 iterations. Step sizes of 5, 15, 25 and 40 are used.

of the tasks to generate the content of Figure 5.9. The result from Figure 5.7 shows that for tuned ratio of 0.4 running on 16 nodes on NaCl, I get a 14% performance improvement. The execution trace indicates that indeed with the help of the CA approach, more tasks are enabled for execution while network messages are exchanged and it generally results in higher CPU occupancy. And the faster execution is achieved despite the fact that the base version has median kernel time of 136 milliseconds while CA version has median kernel time of 153 milliseconds due to the extra copies in the body.

5.6 Conclusions

In this chapter, I described and analyzed implementations of a 2D stencil code and its communication-avoiding (CA) variant on top of the PaRSEC runtime system. In particular, I proposed three implementations of a 5-point stencil as our test cases. I showed performance results on two distinct systems: NaCl and Stampede2; and compared three versions: PETSc, baseline PaRSEC and CA PaRSEC. The approaches based on a tasking runtime show good performance results, with minimal distinction between the two approaches in all compute-intensive scenarios. By artificially reducing the kernel execution time, I highlighted the case where the CA variant on top of PaRSEC is able to outperform the others in the strong scaling regime with up to 57% and 33% improvements on both the NaCl and Stampede2 systems.

On the current state-of-the-art high performance computing system such as the Department of Energy's Summit at Oak Ridge National Laboratory, each node has 6 GPUs and in excess of 900 GB/s HBM2 memory bandwidth per GPU and network latency of only about 1 microsecond [Vazhkudai et al.]. The emerging new exascale systems feature even higher memory bandwidth improvements with HBM2e, but the improvement of the network latency remains modest – a well established trend of growing compute-communication gap [63]. Thus, if the workload on each node can efficiently utilize the full memory bandwidth then, in all likelihood, it would become network-bound and the implementation variant based on communication-avoiding approach shows a distinct advantage. However, other changes

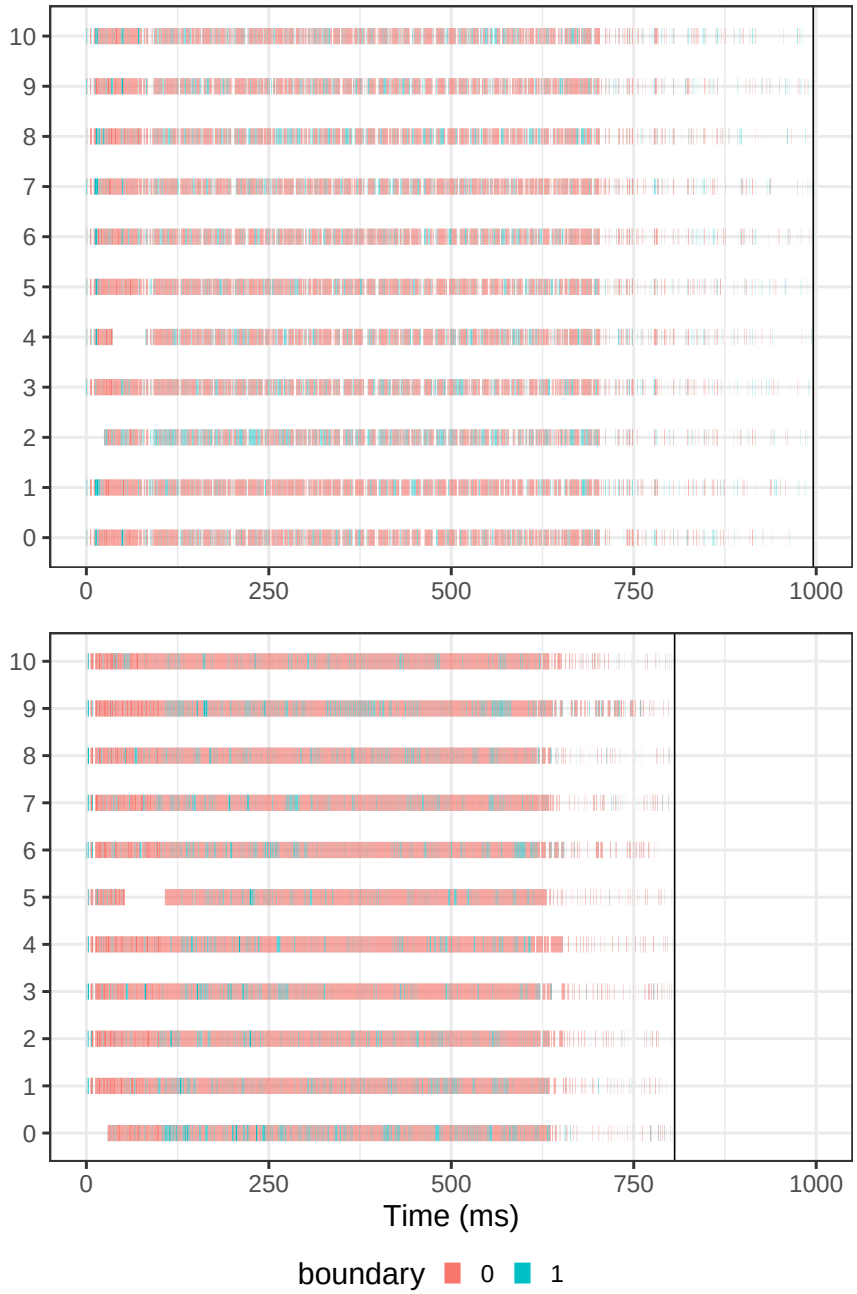


Figure 5.9: One node's profiling result, running on NaCl with 16 nodes, tuned ratio of 0.4, 11 computation threads on a node. (top) baseline PaRSEC; (bottom) CA PaRSEC. The boundary indicates the tiles that need to exchange data with remote nodes.

orthogonal to this study like increasing the arithmetic intensity of the algorithms by using higher-order discretizations and accompanying stencils, or increasing workload on each node could also provide effective ways to mitigate the network inefficiencies.

Another way to look at the benefits of the communication avoiding approach is to note how it aggregates the data across several iteration steps. This reduces the communication frequency to counteract the latency overhead and thus transforming a latency-bound algorithm into a bandwidth-bound one. This also allows us to more efficiently use the network due to communicating larger messages that allow increased bandwidth efficiency from 20% percent to 70% of peak network bandwidth as shown in the NetPIPE results in Figure 5.4. By performing redundant computations, we delayed the network latency penalty and achieved strong-scaling to larger node counts as a result.

Chapter 6

Profiling Analysis for Performance Tuning

6.1 Overview

It has been shown that a task-based approach is extremely efficient for load balancing and using intelligently all the resources' computational power in heterogeneous platforms for many scientific computing fields—including application libraries built on top of the usual dense [6, 28] and sparse [84] linear algebra solvers with either arithmetic- or memory-intense computational tasks. In a task-based programming environment, a large amount of parallelism is exposed by representing the algorithm as a large set of fine-grained tasks. Then, the runtime system is responsible for scheduling these tasks while satisfying the data dependencies between them for correctness. Such a runtime must adapt to the changes in the amount of parallelism available in applications and map tasks to underlying hardware resources under dynamic and unpredictable system conditions. PaRSEC [29] is one of the leading runtime systems that were actively developed and integrated into scientific application codes.

Although a task-based runtime system is convenient and efficient, from users' perspective a well-designed profiling system is needed to inspect the execution—especially when it suffers

from subtle performance problems that tend to be tedious to diagnose. The profiling system needs to integrate well with the runtime and be able to extract information that allows reasoning about task costs, scheduling quality, memory usage, and information regarding messages and data transferred in the network. In this chapter, I describe the profiling system of PaRSEC: the mechanisms embedded in the runtime system to extract critical information and produce a trace of the execution, and the tools allowing users to manage this collection of events. Based on this profiling system, I demonstrate how a performance issue was pinpointed for the Tile Low-Rank (TLR) Cholesky implementation, leading to the lookahead control flow optimization in Section 5 of [36].

6.2 Related Work

In this section, I focus on the profiling and performance instrumentation systems available for task-based runtime systems under the most active development.

Legion includes a performance profiling tool called Legion Prof, that generates at runtime a log file in an internal data format of the task graph execution [107, 85]. The logs can be converted to a set of dynamic HTML pages using a tool provided with the Legion distribution that shows utilization graphs of the processors and their memory during the run. These webpages are dynamic, and more detailed information can be obtained from them, including common pieces of information like showing what tasks executed on what resource at what time, and showing part of the directed acyclic graph (DAG) connecting these tasks.

StarPU provides multiple approaches for performance analysis. On the one hand, the analysis can happen online [103]: dynamic hooks are available for the application developer to connect to task- and communication-related events and write their own tracing or performance analysis mechanisms, or general statistics on the process status can be read at runtime (e.g., amount of compute time per core, time spent in the runtime system per core, etc.). On the other hand, the performance analysis can be conducted offline after creating a trace of the execution [102]. StarPU uses the Fast User/Kernel Tracing (FxT) [94] library to create traces that can be converted into GraphViz' DOT graph representations or the

Pajé trace format [45]. The latter can then be visualized with the ViTE tool [41] as an annotated Gantt chart. StarPU provides additional tools to create text files describing the execution of each task in a key/value format for integration with external tools, and allows the user to build application-specific analysis. It is possible to use a combination of the trace formats (Pajé, DOT, enriched text files), and with support of ad hoc conversion scripts to build a CSV database of the execution and analyze it in R or other statistical tools, using application-specific methods [57]. Lastly, a set of internal tools are also available in StarPU to measure the efficiency of the performance models built by the runtime system for its scheduling, and to check the accuracy of the simulations, if they are conducted with the runtime system.

To the best of my knowledge, QUARK does not provide any profiling or tracing tools within the runtime system to help the performance analysis aside tracing wrappers around scheduling calls that integrate with the Pajé-ViTE trace tooling duo. Initially, instrumentation was accomplished with manual intervention into each task's invocation in order to collect timing information and build Gantt diagrams and other performance analysis [69].

OmpSs includes a set of instrumentation plug-ins [88], that can be selected at run time, in order to dynamically call functions defined in these plugins when specific events occur. The set of events that trigger a call is controlled at compile time by a variety of options. Available plugins include an Ayudame plugin for the Temanejo graphical debugger [33], a module to compute and output the DAG of tasks. Another one, provides an experimental support for traces from execution of a task system simulator. Also, a module is available to provide traces suitable for us by Paraver [93] that can potentially embed Performance Application Programming Interface (PAPI) information obtained directly from hardware counters. Traces of parallel runs can be visualized as Gantt charts using Paraver from a variety of perspectives (e.g., from a task view or a thread perspective, showing the achieved rate of instructions per cycle in different thread contexts, or the TLB miss ratio, etc.) [96].

The HPX Performance Counter Framework [64] defines an API to access internal counters exposed by the HPX runtime. These counters include information about the hardware, but also about the runtime system's status. The counter values are identified by their names, following a fixed naming scheme. The runtime provides rudimentary tools to regularly read a set of hardware counters and display them on the standard output or send them to an output file, but this approach is only time-periodic and not event-driven and thus does not allow for creating a trace of the execution. The preferred approach is to embed the user analysis directly within the HPX program, or to write user's own tracing layer within the application for more sophisticated offline analysis.

As I describe below, the approach in PaRSEC differs from the other approaches in that a detailed trace of the execution is created and converted into an open format, which encourages the development of small and application-specific analysis tools in simple scripting languages. Below, I will illustrate how this approach allows users to take an application written with PaRSEC and collect a trace of execution with fine enough details to allow a programmer with a good understanding of the application itself to identify the bottlenecks and successfully eliminate them.

6.3 Background

This section briefly provides background information on the Tile Low Rank (TLR) Cholesky factorization. More detailed information is provided in Sections 4 and 5 of [36].

6.3.1 TLR Cholesky Factorization Basics

In the standard dense Cholesky factorization, the matrix data is stored in the form of an underlying tile layout and is usually executed by four computational kernels: POTRF (single-tile local Cholesky factorization), TRSM (triangular solve with a matrix), SYRK (symmetric rank-k update), and GEMM (general matrix-matrix multiply) on either the lower or upper part of the symmetric positive-definite matrix. The entire factorization translates into a

DAG with nodes corresponding to tasks and edges representing data dependencies, with a serial and incompressible critical path of $(n_t - 1)\times$ (POTRF + TRSM + SYRK) + POTRF, where n_t is the number of row or column tiles.

The DAG of tasks (and thus its critical path) is the same for both TLR and classic dense tile-based Cholesky factorization as shown in Figure 6.1, but there are two critical differences:

1. **data format**: all tiles are dense with size of $n_b \times n_b$ in the dense Cholesky factorization, where n_b is the tile size; while in TLR Cholesky factorization, only tiles on the main diagonal are dense, and off-diagonaltiles are numerically compressed using the application-dependent accuracy threshold by using a variant of the Singular Value Decomposition (SVD) with size of $n_b \times$ rank with rank $\ll n_b$ for tiles further away from the diagonal tiles;

2. **computational kernels, as well as arithmetic complexity**: to work on the compressed data layout of the off-diagonal tiles, TLR Cholesky requires an implementation of new low-rank kernels variants: LR_SYRK and LR_GEMM, which introduce decompression and compression phases, respectively, as introduced in the prior work [9]; the arithmetic complexity is $2 \times n_b^2 \times \text{rank} + 4 \times n_b \times \text{rank}^2$ for LR_SYRK and $36 \times n_b \times \text{rank}^2$ for LR_GEMM, instead of n_b^3 for SYRK and $2 \times n_b^3$ for GEMM. And for TRSM, the arithmetic complexity is reduced to $n_b^2 \times \text{rank}$ from n_b^3 as well.

6.4 Performance Tools

PaRSEC features a rich development environment including tools to debug programs written in the different DSLs, and to profile the performance of system's task execution. In this section, I present in further detail the performance profiling and instrumentation capabilities of PaRSEC.

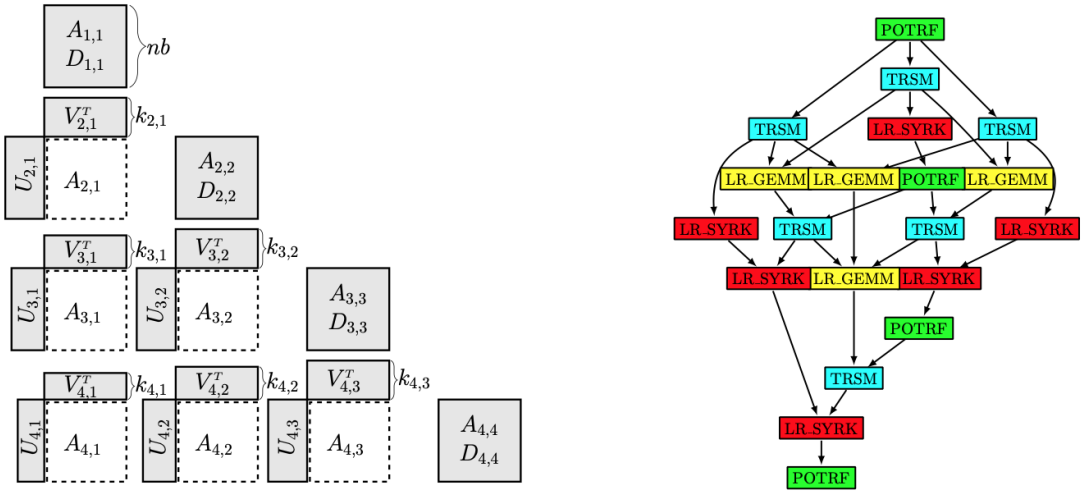


Figure 6.1: Left, TLR format for matrix A having 4-by-4 tiles of size nb-by-nb. Diagonal tiles are stored as dense. Off-diagonal tiles, are compressed to have U and V blocks, each has its own rank, k. Right, the corresponding DAG for TLR POTRF of the matrix.

6.4.1 Trace Collection Framework

The Trace Collection Framework may be considered at the root of the performance profiling system. It is an integral part of the PaRSEC runtime system and can be enabled through a compile-time option. The framework consists of a runtime support thread and library that provides a generic API to define and store events that occur during the execution. The user program (typically the PaRSEC runtime and the different PaRSEC DSLs) defines events as individually identified entities that are executed on a given thread at a given time, and they are bound with a contiguous structure of arbitrary size that holds information pertaining to the event. For example, for each task of the PTG DSL, the DSL defines task-start and task-end events that store the task class, task identifier, and parameters of the task.

These events are stored in a set of binary files, one per every process of the application. In each file, events are grouped in buffers of fixed size, each buffer belonging to a given thread of the process. Buffers are linked one to another, creating as many linked lists of buffers as there were threads in the process during the execution.

The library is designed to be highly scalable for many-thread environments and to incur a minimal overhead when logging events. Logging an event consists of reading a timer, and copying the information related to the event (which has a size from a dozen to a few hundreds bytes, depending on the event type) in a buffer of memory that is memory-mapped onto the backend file that stores the binary trace. At runtime, each PaRSEC thread owns an independent buffer to log its events, in order to avoid sharing and atomicity issues. When a buffer is filled, the PaRSEC thread that is logging an event atomically swaps its current logging buffer with a fresh one. The helping thread that is part of the Trace Collection Framework continuously expands the backend file on which these buffers are mapped, and prepares in advance new buffers for the PaRSEC threads to acquire when needed. The only thread-synchronizing operations occur when requesting a new buffer and releasing the current one, and different PaRSEC threads never interact with each other's tracing structures during the computation.

This approach relies on the availability of a few buffers of memory: one buffer per PaRSEC thread for the current buffer, and a few more that are allocated in advance to overlap I/O operations with logging operations. If the PaRSEC threads generate events faster than the operating system can complete the cycle of truncating the backend file, mapping the new area, and unmapping the completed areas, the system will throttle the incoming event data stream by slowing down the logging operations in order to complete the preceding ones. This is usually avoided entirely when the backend file is stored on a scalable or local I/O systems. The Tracing Framework helping thread is usually left without affinity to a particular CPU core, in order to allow "stealing" of idle cycles from computing threads, as all its time is spent waiting on incoming event data or being blocked on I/O operations.

In an effort to improve portability and to enable interoperability with existing performance analysis tools for parallel applications, the tracing interface can also be configured at compile time to produce its output log of the execution in the OTF2 trace format [54]. In this work, I focused on the binary trace collection and the conversion methods described below allowed me to build my own ad-hoc analysis tools.

6.4.2 PINS: PaRSEC INStrumentation

The Trace Collection Framework is used within the PaRSEC runtime through the PaRSEC INStrumentation (PINS) interface: different modules can register callbacks that typically log events that form the trace, and are called when the execution reaches critical points in the code. PINS modules are exposed to the final user through the Modular Component Architecture (MCA) [56], and can be selected at run time to decide the type of information logged in the binary profile files.

Typically, PINS registers callbacks for all the important steps of a task's life cycle: when it is created, when it becomes ready to run, when it is selected for execution, when it is assigned to an accelerator (if eligible and available), when it starts and ends its execution, and when it enables one of its successor tasks. There are also callbacks available that pertain to the

state of the PaRSEC runtime: when it allocates or frees system resources, when network events are triggered, etc.

The MCA design exposes different logging policies, available for the user: for example, the `pins_papi` module allows logging PAPI's hardware counters of user's choice. This is in addition to basic tracing that by default records the time and thread that generated each event. This enables augmenting the trace with information on the state of the hardware at the time of the event.

6.4.3 Dependency Analysis

The events instrumentation allows us to measure the status of the system at critical moments of task scheduling. In order to enable full analysis of the program's behavior, it is often necessary to connect this information with the actual DAG of tasks that was executed. In order to achieve this, the events' trace is completed with another file representing the dependencies as they are expressed to the runtime system in another set of files following the DOT syntax defined by the GraphViz software collection [53] for portability.

For all deterministic problems (typically when the DAG of tasks is not data-dependent, but is entirely defined by the parameters that instantiate the DAG as is the case with the PTG DSL), the assembly of the DAG can be done offline and does not have to be performed during the timing of the operation itself, thus completely avoiding the risk of impacting the execution's timing. One DOT file per process is produced, as for the tracing mechanism, and all PaRSEC DSLs provide a unique naming of tasks that enables an internal tool to stitch the different DOT files to produce a single one that represents the entire distributed DAG of tasks.

6.4.4 Trace Conversion Tools

It is important to note from the outset, that this is not the case for the binary trace: once a trace is generated, the user has access to a set of binary files, one per process in the application. The format of these files is not exposed to the user, as information in them is

kept as close as possible to the architecture, in order to avoid conversion costs to produce a portable trace format during the execution. Timing information, for example, is architecture- and operating system–dependent; each architecture defines its own time reading routine. All information logged by the user (typically integers of various size to store the parameters, PAPI counters, etc.) is also kept in the architecture-specific storage.

As is often the case with tracing systems, a conversion step is necessary to obtain a portable and exploitable file format of the trace. During this step, the generated binary files are merged in a single file by appending the rank of the process that produced the initial binary file as an identifier for each record. For portability and ease of use, PaRSEC chose to export the portable file format in Hierarchical Data Format (HDF5), following the structure required by the popular the Pandas library [87] to describe Data Frames in HDF5.

HDF5 [55] is an open format, self-describing, and efficient in representing large data sets. The self-describing property of HDF5 enables exposing a large variety of data collections with only minimal external documentation. Pandas [87] is a popular Python library providing high-performance, easy-to-use data structures and data analysis tools for the Python programming language. The goal behind these choices is to simplify writing ad hoc analysis tools tailored to their application, as is done in the following techniques I present below. The PaRSEC programming environment also provides tools to take the generated trace and convert it into a full Gantt chart, or simply compute basic summary statistics of the entire trace.

The HDF5 file contains a few of Pandas so called DataFrames or Series: a Series describes what event types have been registered with the application, and associates an identifier to them; another one collects all the architectural information, at the application level, or per process and per thread basis. The largest DataFrame is a relational array that stores all the events logged during the execution (one per row), and provides a tabular view of each record, where the columns define the fields of the events. Some fields are common to all events (e.g., timing of the event's start and end, resource identifier that produced the event,

etc.), and other fields that correspond to the information logged by the PINS module and are specific to some event types.

In order to simplify the development of ad hoc analysis tools in Python with Pandas, the PaRSEC environment also provides a library to read the generated DOT files into a NetworkX [68] representation that understands the naming scheme of the specific DSL, and connects the tasks in the graph object with the records in the Pandas DataFrame. The user can then easily select an event, find the task that relates to it, explore its successors or predecessors and find events relating to those in the DataFrame. The case study in this chapter makes use of this particular feature.

6.5 TLR Cholesky Case Analysis

The exploratory analysis of the data can proceed along many different directions, but there are several intuitive concepts that lead me to investigate the connections and delays between message transfers and task execution periods. First, the critical path of the Cholesky factorization is important for my analysis because it is executed as if it was sequential. Also, the tasks from the critical path are responsible for enabling all the parallel kernels, that update off-diagonal tiles, and that are essential for keeping the compute threads occupied with useful work. Second, the low-rank kernels were meant to reduce the computational intensity, and as a result, rendered the execution less tolerant to deviation and jitter occurring on the critical path, which may easily lead to work starvation and show up as under-utilization of the compute threads. And finally, based on my understanding of the PaRSEC runtime, which eagerly enables tasks for maximum parallelism, the scheduler can get overwhelmed with the amount of tasks eligible for execution and during scheduling, the less critical tasks could be selected, even with the scheduling hints available in the form of task priorities.

In order for the output files to be manageable, I performed a limited experiment on a 3×3 compute grid, with tile size of 2700 and a total of 100×100 matrix tiles. The tested kernel was a synthetic 2D application kernel where the kernels for off-diagonal work had average

numerical rank of 13 after compression (consult Figure 2 in [37] for further details). I profiled the resulting execution to ensure that as soon as the data is ready, PaRSEC enables the critical tasks first according to the priority information. To be able to compute the average time it takes for data to be produced on one node and consumed on another, I had to connect multiple distributed events: the task termination, network activation, payload emission, and remote task execution. This information was provided by the PaRSEC's profiling system through a combination of the trace information and the DOT files.

The generated DOT files are in the GraphViz DOT format, and can be used to graph the DAG of tasks. They also can be viewed directly with a graphical editor and a typical file entry is shown in Figure 6.2: it contains both the nodes as well as edges. For example, the text `tpid=4, tcid=0, tid=0` represents `potrf_dpotrf` task when value of `k` is 0. And the second entry indicates that there's an edge between the `potrf(0)` task and the `trsm(0, 2)` task. Tasks in DOT files can be uniquely mapped to the events corresponding to data payload sending and the payload receiving in the HDF5 profiling files through the following association: `tpid -> taskpool_id`, `tcid -> tcid`, and `tid -> id`. My Python script defines the ParsecDAG class, which uses NetworkX package to convert the information in DOT files into a complete DAG. The HDF5 file contains the information of the recorded time of MPI's data send on the sender rank, the MPI's data receive time on the receiver rank. It also contains the start and end times of each of the executed tasks as shown in Figure 6.3.

By combining the information from these sources, I could identify the times at which the diagonal tasks finished and the times when the following off-diagonal triangular updates start executing. Figure 6.4 shows the time interval between receiving the diagonal data for the POTRF task and the start of TRSM tasks in the matrix panels. In the default case, the tasks operating on data closest to the diagonal experience the largest delay, and this clearly is not optimal for the execution of the TLR Cholesky factorization, as the tasks on the critical path, operating on the data close to the diagonal, should have the highest priority to ensure that more off-diagonal updates are enabled for increased level of parallelism. Thus by using

```
potrf_dpotrf_4_0 [pencolor="...",label="<0/0/0>potrf_dpotrf(0)<512>",
    tooltip="tpid=4:tcid=0:tcname=potrf_dpotrf:tid=0"];

potrf_dpotrf_4_0 -> potrf_dtrsm_4_0_2 [label="T=>T"];
```

Figure 6.2: Example DOT file entries.

```
>>> import pandas as pd
>>> t = pd.HDFStore('dpotrf.h5')
>>> t.events[ t.events.type == t.event_types['MPI_DATA_PLD_RCV'] ]
      begin       end  src  dst  tpid   tcid  tid
2  54829493  55167839  1.0  0.0   2.0  10.0  4.0
3  54863972  55167839  1.0  0.0   2.0  10.0  1.0
6  45964530  46325381  0.0  1.0   2.0  10.0  0.0
7  46007558  46325381  0.0  1.0   2.0  10.0  6.0
8  46515282  46671406  0.0  1.0   2.0  10.0  3.0
9  57474746  57530395  0.0  1.0   2.0  10.0  2.0
>>> t.event_types
ACTIVATE_CB          6
Device delegate      1
MPI_ACTIVATE         2
MPI_DATA_CTL         3
MPI_DATA_PLD_RCV     5
MPI_DATA_PLD_SND     4
PUT_CB               7
TASK_MEMORY          0
potrf_dgemm          8
potrf_dpotrf        11
potrf_dsyrk          9
potrf_dtrsm         10
dtype: int64
```

Figure 6.3: Example HDF5 file entries.

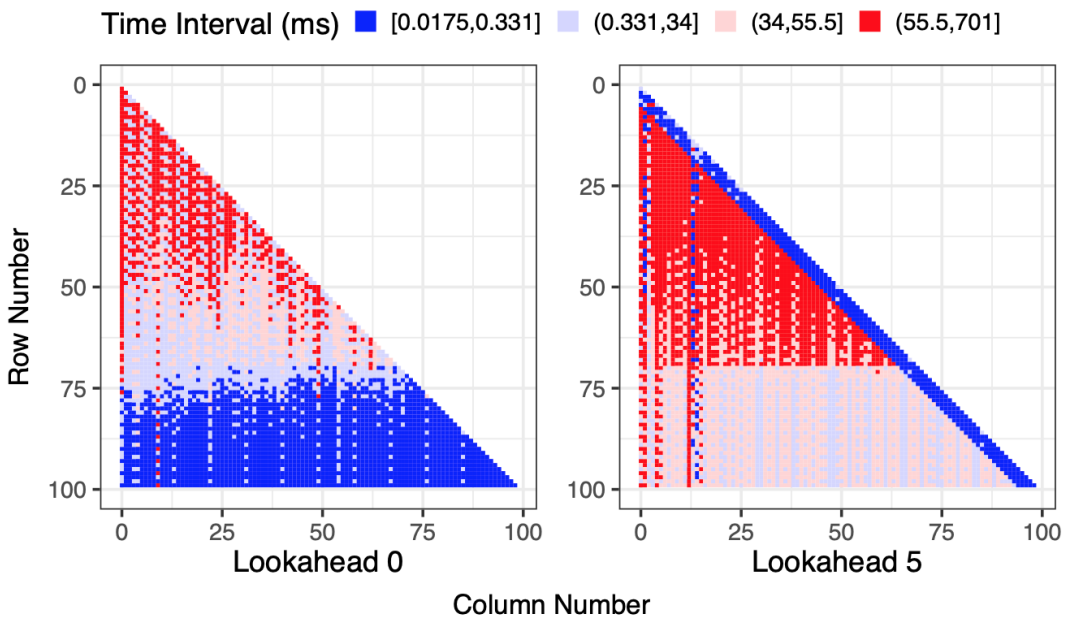


Figure 6.4: Time between data is ready and TRSM starts for st-2D-sqexp synthetic kernel data. Left, without lookahead; right, with lookahead of 5; each point represents one TRSM; matrix has 100×100 tiles.

my profiling tools for analysis I identified this inefficient scheduling effect and proceeded to addressing the resulting performance issue. I did it by introducing a new lookahead mechanism in the control flow between the LR_SYRK and TRSM tasks. This allowed the scheduling of the TRSM tasks, which worked on data further away from the diagonal, to only run after the critical updates near the diagonal have fully completed. With my new customized lookahead, the updated profiling visualization shows that indeed the intended scheduling change occurred and a performance improvement was achieved.

6.6 Conclusions

In this chapter, I presented the profiling system available in PaRSEC: the mechanisms embedded in the runtime system to extract critical timing information and produce a rich trace of the execution, and the tools allowing users to manage this collection of events. Using the information provided by this profiling system, I demonstrated the performance analysis to show how the optimization footprint of the TLR Cholesky factorization could impose a stricter control on execution sequence resulting in faster completion time. This perspective, which appropriately highlights the benefits of PaRSEC's instrumentation tools, provided insights into PaRSEC's scheduling process and system's details during execution thus enabling a comprehensive understanding of the behavior of the entire application, which in turn was crucial in identifying potential performance bottlenecks and regressions.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

The constant evolution of the HPC systems with more and more heterogeneous components poses a serious complexity challenge to the programming models in the software space. It is expected that in the near future, MPI standard will continue to be the dominant model of choice for applications to handle inter-node communication, while the landscape of viable choices for shared memory systems' programming (e.g. OpenMP, CUDA, Kokkos) will proliferate and thus will continue the trend of strong reliance on the MPI+X programming approach. At the same time, the ideal model should achieve a good balance between the usability on the one hand and on the other it should not get in the way of ability to extract a sizable portion of the peak performance from multiple generations of the HPC systems. In this regard, task-based programming model and their modern implementations has become a serious candidate for users' codes, including scientific applications, despite the initial effort required to adapt the existing applications to this promising paradigm.

My contributions to task-based programming model in general and to PaRSEC, one of its prominent runtimes, in particular, include the following:

Programming Models Evaluation: This work compared the dominant MPI and MPI+X programming models with two runtime systems for task-based model using the BLR LU

factorization as the scientific test program. Due to the load imbalance of this workload, not uncommon among scientific codes, the inherent ability to handle such use case via communication and computation overlap inside a runtime system was a major outcome of my work that resulted in better node performance and scalability results. Within the two task-based runtime systems, STF model is simpler from user perspective and consequently saw some level of adoption, but the performance is limited by the dynamic task graph analysis and the associated overheads. I showed that PaRSEC PTG is the most scalable approach, but it uses its own DSL with custom syntax and as a result is harder to adopt. Distributed task-based runtimes represent a shift away from the MPI+X approach and require benchmarking efforts and active demonstrations to encourage wider adoptions. The limitations exposed during my efforts (e.g. STF overhead) and the capabilities highlighted for efficient implementation (e.g. prioritizing tasks on the critical path for the scheduler to to follow, handling dynamic data sizes that change throughout the line) help drive runtime optimizations further.

STF Optimizations: Following the discoveries from the first study and other results from the literature, I proposed and implemented two optimizations in PaRSEC's DTD which follows closely the STF model, namely user-level graph trimming and a new API for dynamically constructed data broadcast operations. My performance results from Cholesky and QR factorizations indicated that they both can provide a certain degree of improvement, but it can be limited due to inherent design choices because we can not avoid the STF overheads under all circumstances, but rather we may be able to postpone the issues within certain scalability regimes. Additionally, implementing graph trimming at the user level requires non-trivial amount of user involvement: not only the local writer tasks need to specify all the remote tasks that use the updated data, the data user tasks need to insert the remote data writer tasks to provide with the latest copy. In essence, the algorithm writer needs to provide the same amount of information to trim the graph as is required to write the algorithm with the PTG

approach, thus breaking the ease-of-use benefit of the STF model and producing more complicated and thus error-prone code.

Extension of PTG to Support Communication Avoiding (CA): The recent hardware trends suggest that for sparse iterative solvers, although it is beneficial to adopt CA to speed up the SpMV calculation, however, the limited options for preconditioner have led to the shift away from this approach. Still, there are many stencil-like simulations derived from solving PDEs that can benefit from the combined benefits of runtime systems and CA. As the gap between improvements in the computational power and memory/network bandwidth continues to widen, this combination promises to remain relevant going forward. My results from the simple five-point 2D stencil code demonstrated these conditions with a positive impact on total running time.

Flexible Profiling Analysis: In order to increase application performance at the software level, we need to either innovate at the algorithm level or improve the adopted software system. Profiling tools help us identify the performance bottlenecks and facilitate performance optimization as we iterate this process as necessary until goal metrics are reached. For task-based runtime system with dynamic execution behavior, the profiling results contain task scheduling information, message transfer events, kernel execution time, runtime overhead, and many other pieces of information. In my work, I demonstrated the ability of PaRSEC's profiling system to collect this information, and implemented the flexible data analysis tools, that I subsequently used to synthesize them together to identify performance issues. With the TLR Cholesky factorization as test case, I was able to connect network events with task scheduling events to pinpoint an issue with the use of PaRSEC scheduling. And this lead me to the subsequent addition of control sequences to better guide the execution along critical path.

7.2 Future Work

This dissertation explored several of the key programming models with focused set of algorithms for scientific applications. Exploring their performance on more applications' domains would be of great interest, since the different characteristics of the computational workload can further showcase the variety of benefits of the task-based runtime approach, and shed light on new areas of possible improvements. Consistent benchmarking efforts is important for robust performance engineering and reliable hardware evaluation and software systems' analysis, thus expanding my baseline set of workloads to include other emerging programming models would be beneficial to further inform the application writers on the essential performance-productivity-portability trade-offs and would be worthy extension of my my work as well.

There are several opportunities for performance improvements of my DTD broadcast implementation. The first one is to add the ability to select a different broadcast topology for each of the subsequent broadcast collectives, since the best topology is usually dependents on the message size and node counts of the broadcast tree. In my work, only one topology layout is used for all calls. The second optimization is related to the PaRSEC's implementation of collective communication. Currently, the participating process ranks are encoded as a set by an implementation of a bit array to reduce memory usage, resulting in a loss of process ordering. As a result, the selection of the rank of the descendant process does not take into account task's priority, but only follows the ranks' numbering order. Instead, we could use an ordered set of communicating process ranks and use an array to represent the ranks, giving us more detailed ordering information at the price of the extra storage space: this could encode both ordering and priority information in a single array and be used during scheduling of message exchanges. And finally, we could cache the metadata associated with each collective operation on each of the participating nodes, where information similar to that of an MPI communicator is stored for even better scheduling decisions. Note that for any future collective exchanges, we could avoid the transfer of the metadata array, and use only a key to obtain a location of the cached group information.

My study on combining the communication avoiding and runtime system was just the beginning with additional new directions still possible. A more representative set of applications needs to be studied to better understand whether PaRSEC's PTG is a suitable DSL to form a foundation for efficient implementations. And for these new applications, a more generic communication avoiding framework could be created and also could be built directly into the runtime system, which I believe can further improve performance. This approach could include automatic data replication across the stencil's grid neighbors, i.e., the grid-owning nodes that share a frontier region with each other. Under such a design, the generation process and the scheduling of the redundant tasks becomes fully transparent to the user.

Finally, although all the evaluations are with CPU-only systems, PaRSEC supports execution of kernels on the available hardware accelerators because it features a device management engine. Evaluating many of the past design proposals would be critical for improvements since many of the leading systems obtain most of their computational power from their attached accelerators, and we would like to demonstrate PaRSEC's capabilities in that area. Also, this would alter the dynamic balance of the system as a whole, where it has less nodes in total but each node has more computational power. How well would the runtime systems perform on this new system configuration? And, what adaptations would be needed to make it work more efficiently under these new circumstances? These questions would be invaluable knowledge to both runtime developers and domain scientists.

Bibliography

[1] (January 2017). The chameleon project. https://gitlab.inria.fr/solverstack/chameleon. 18

[2] Abdulah, S., Cao, Q., Pei, Y., Bosilca, G., Dongarra, J., Genton, M. G., Keyes, D. E., Ltaief, H., and Sun, Y. (2021). Accelerating geostatistical modeling and prediction with mixed-precision computations: A high-productivity approach with parsec. *IEEE Transactions on Parallel and Distributed Systems*, 33(4):964–976. 23

[3] Acun, B., Gupta, A., Jain, N., Langer, A., Menon, H., Mikida, E., Ni, X., Robson, M., Sun, Y., Totoni, E., Wesolowski, L., and Kale, L. (2014). Parallel programming with migratable objects: Charm++ in practice. In *SC'14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 647–658. 21, 29

[4] Agullo, E., Aumage, O., Faverge, M., Furmento, N., Pruvost, F., Sergent, M., and Thibault, S. (2017). Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model. *TPDS - IEEE Transactions on Parallel and Distributed Systems*. 13, 32, 44, 50

[5] Agullo, E., Buttari, A., Guermouche, A., and Lopez, F. (2013a). Multifrontal qr factorization for multicore architectures over runtime systems. In *European Conference on Parallel Processing*, pages 521–532. Springer. 18

[6] Agullo, E., Demmel, J., Dongarra, J., Hadri, B., Kurzak, J., Langou, J., Ltaief, H., Luszczek, P., and Tomov, S. (2009). Numerical Linear Algebra on Emerging Architectures: The PLASMA and MAGMA Projects. *Journal of Physics: Conference Series*, 180. 91

[7] Agullo, E., Giraud, L., Guermouche, A., Nakov, S., and Roman, J. (2013b). Pipelining the CG Solver Over a Runtime System. In *GPU Technology Conference*, San Jose, United States. NVIIDA. 14, 72

[8] Akbudak, K., Ltaief, H., Mikhalev, A., Charara, A., Esposito, A., and Keyes, D. (2018). Exploiting data sparsity for large-scale matrix computations. In Aldinucci, M., Padovani,

L., and Torquati, M., editors, *Euro-Par 2018: Parallel Processing*, pages 721–734, Cham. Springer International Publishing. 23

[9] Akbudak, K., Ltaief, H., Mikhalev, A., and Keyes, D. (2017). Tile low rank Cholesky factorization for climate/weather modeling applications on manycore architectures. In Kunkel, J. M., Yokota, R., Balaji, P., and Keyes, D. E., editors, *High Performance Computing - 32nd International Conference, ISC High Performance 2017, Frankfurt, Germany, June 18-22, 2017, Proceedings*, volume 10266, pages 22–40. 95

[10] Ambikasaran, S. and Darve, E. (2013). An O(N log N) fast direct solver for partial hierarchically semi-separable matrices. *Journal of Scientific Computing*, 57:477–501. 22

[11] Amestoy, P., Ashcraft, C., Boiteau, O., Buttari, A., L'Excellent, J.-Y., and Weisbecker, C. (2015). Improving multifrontal methods by means of block low-rank representations. *SIAM Journal on Scientific Computing*, 37:A1451–A1474. 20

[12] Amestoy, P., Buttari, A., L 'excellent, J.-Y., and Mary, T. (2018). Bridging the gap between flat and hierarchical low-rank matrix formats: the multilevel BLR format. Technical Report hal-01774642, University of Manchester. 22

[13] Amestoy, P., Buttari, A., L'Excellent, J., and Mary, T. (2017). On the complexity of the block low-rank multifrontal factorization. *SIAM Journal on Scientific Computing*, 39(4):A1710–A1740. 22, 23

[14] Amestoy, P. R., Buttari, A., L'Excellent, J.-Y., and Mary, T. (2019). Performance and scalability of the block low-rank multifrontal factorization on multicore architectures. *ACM Trans. Math. Softw.*, 45(1):2:1–2:26. 22

[15] Anderson, E., Bai, Z., Bischof, C. H., Blackford, L. S., Demmel, J. W., Dongarra, J. J., Croz, J. J. D., Greenbaum, A., Hammarling, S., McKenney, A., and Sorensen, D. C. (1999). *LAPACK User's Guide*. SIAM, Philadelphia, 3rd edition. 18

[16] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011a). StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198. 23

[17] Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011b). StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198. 48

[18] Bachan, J., Bonachea, D., Hargrove, P. H., Hofmeyr, S., Jacquelin, M., Kamil, A., van Straalen, B., and Baden, S. B. (2017). The UPC++ PGAS Library for Exascale Computing. In *Proceedings of the Second Annual PGAS Applications Workshop*, PAW17, pages 7:1–7:4, New York, NY, USA. ACM. 23

[19] Bak, S., Menon, H., White, S., Diener, M., and Kalé, L. V. (2018). Multi-level load balancing with an integrated runtime approach. In *18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2018, Washington, DC, USA, May 1-4, 2018*, pages 31–40. 29

[20] Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Dener, A., Eijkhout, V., Gropp, W. D., Karpeyev, D., Kaushik, D., Knepley, M. G., May, D. A., McInnes, L. C., Mills, R. T., Munson, T., Rupp, K., Sanan, P., Smith, B. F., Zampini, S., Zhang, H., and Zhang, H. (2019a). PETSc users manual. Technical Report ANL-95/11 - Revision 3.12, Argonne National Laboratory. 18, 75

[21] Balay, S., Abhyankar, S., Adams, M. F., Brown, J., Brune, P., Buschelman, K., Dalcin, L., Dener, A., Eijkhout, V., Gropp, W. D., Karpeyev, D., Kaushik, D., Knepley, M. G., May, D. A., McInnes, L. C., Mills, R. T., Munson, T., Rupp, K., Sanan, P., Smith, B. F., Zampini, S., Zhang, H., and Zhang, H. (2019b). PETSc Web page. 75

[22] Balay, S., Gropp, W. D., McInnes, L. C., and Smith, B. F. (1997). Efficient management of parallelism in object oriented numerical software libraries. In Arge, E., Bruaset, A. M.,

and Langtangen, H. P., editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press. 75

[23] Basu, P., Venkat, A., Hall, M., Williams, S., Van Straalen, B., and Oliker, L. (2013). Compiler generation and autotuning of communication-avoiding operators for geometric multigrid. In *20th Annual International Conference on High Performance Computing*, pages 452–461. 71, 72

[24] Bauer, M., Lee, W., Slaughter, E., Jia, Z., Di Renzo, M., Papadakis, M., Shipman, G., McCormick, P., Garland, M., and Aiken, A. (2021). Scaling implicit parallelism via dynamic control replication. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 105–118. 14

[25] Bauer, M., Treichler, S., Slaughter, E., and Aiken, A. (2012). Legion: Expressing locality and independence with logical regions. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC*, pages 1–11. IEEE. 13, 48

[26] Bebendorf, M. and Rjasanow, S. (2003). Adaptive low-rank approximation of collocation matrices. *Computing*, 70:1–24. 23, 24

[27] Blackford, L. S., Choi, J., Cleary, A., D'Azevedo, E., Demmel, J., Dhillon, I., Dongarra, J., Hammarling, S., Henry, G., Petitet, A., et al. (1997). *ScaLAPACK users' guide*, volume 4. SIAM. 18

[28] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Haidar, A., Herault, T., Kurzak, J., Langou, J., Lemarinier, P., Ltaief, H., Luszczek, P., YarKhan, A., and Dongarra, J. (2011). Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1432–1441. 18, 91

[29] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Herault, T., and Dongarra, J. (2013). PaRSEC: A Programming Paradigm Exploiting Heterogeneity for Enhancing Scalability. *Computing in Science and Engineering*, 99:1. 2, 91

[30] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Herault, T., and Dongarra, J. J. (2013). PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science Engineering*, 15(6):36–45. 11, 21

[31] Bosilca, G., Bouteiller, A., Danalis, A., Faverge, M., Hérault, T., and Dongarra, J. J. (2013). PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science & Engineering*, 15(6):36–45. 48

[32] Bosilca, G., Harrison, R. J., Hérault, T., Javanmard, M. M., Nookala, P., and Valeev, E. F. (2020). The template task graph (ttg)-an emerging practical dataflow programming paradigm for scientific simulation at extreme scale. In *2020 IEEE/ACM Fifth International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, pages 1–7. IEEE. 11

[33] Brinkmann, S., Gracia, J., and Niethammer, C. (2013). Task debugging with temanejo. In *Tools for High Performance Computing 2012*, pages 13–21, Berlin, Heidelberg. Springer. 93

[34] Cao, Q., Bosilca, G., Wu, W., Zhong, D., Bouteiller, A., and Dongarra, J. (2020a). Flexible data redistribution in a task-based runtime system. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 221–225. IEEE. 11

[35] Cao, Q., Pei, Y., Akbudak, K., Bosilca, G., Ltaief, H., Keyes, D., and Dongarra, J. (2021). Leveraging parsec runtime support to tackle challenging 3d data-sparse matrix problems. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 79–89. IEEE. 11, 23

[36] Cao, Q., Pei, Y., Akbudak, K., Mikhalev, A., Bosilca, G., Ltaief, H., Keyes, D., and Dongarra, J. (2020b). Extreme-scale Task-based Cholesky Factorization Toward Climate and Weather Prediction Applications. In *Proceedings of the Platform for Advanced Scientific Computing Conference (PASC)*, pages 1–11. 11, 23, 44, 47, 92, 94

[37] Cao, Q., Pei, Y., Herault, T., Akbudak, K., Mikhalev, A., Bosilca, G., Ltaief, H., Keyes, D., and Dongarra, J. (2019). Performance analysis of tile low-rank cholesky factorization using parsec instrumentation tools. In *2019 IEEE/ACM International Workshop on Programming and Performance Visualization Tools (ProTools)*, pages 25–32. IEEE. 102

[38] Chamberlain, B. L., Callahan, D., and Zima, H. P. (2007). Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312. 8

[39] Chandrasekaran, S., Gu, M., and Pals, T. (2006). A fast ULV decomposition solver for hierarchically semiseparable representations. *SIAM Journal on Matrix Analysis and Applications*, 28(3):603–622. 22

[40] Chapman, B., Curtis, T., Pophale, S., Poole, S., Kuehn, J., Koelbel, C., and Smith, L. (2010). Introducing openshmem: Shmem for the pgas community. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, pages 1–3. 8

[41] Coulomb, K., Faverge, M., Jazeix, J., Lagrasse, O., Marcoueille, J., Noisette, P., Redondy, A., and Vuchener, C. (2009). Visual trace explorer (ViTE). Technical report, Technical report. 93

[42] Danalis, A., Bosilca, G., Bouteiller, A., Herault, T., and Dongarra, J. (2014). Ptg: An abstraction for unhindered parallelism. In *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pages 21–30. 2, 11

[43] Danalis, A., Bosilca, G., Bouteiller, A., Herault, T., and Dongarra, J. (2014). PTG: an Abstraction for Unhindered Parallelism. In *2014 Fourth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing*, pages 21–30. IEEE. 48

[44] Datta, K. (2009). *Auto-Tuning Stencil Codes for Cache-Based Multicore Platforms*. PhD thesis, USA. xv, 73, 74

[45] de Kergommeaux, J. C., Stein, B., and Bernard, P. (2000). Pajé, an interactive visualization tool for tuning multi-threaded parallel applications. *Parallel Computing*, 26(10):1253 – 1274. 93

[46] Demmel, J., Grigori, L., Hoemmen, M., and Langou, J. (2012). Communication-optimal parallel and sequential QR and LU factorizations. *SIAM J. Sci. Comput.*, 34(1):206–239. 20

[47] Demmel, J., Hoemmen, M., Mohiyuddin, M., and Yelick, K. (2008). Avoiding communication in sparse matrix computations. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12. 72, 75

[Demmel et al.] Demmel, J., Hoemmen, M. F., Mohiyuddin, M., and Yelick, K. A. Avoiding communication in computing Krylov subspaces. Technical Report UCB/EECS-2007-123, EECS Department, University of California, Berkeley. xv, 76

[49] Denis, A., Jeannot, E., and Swartvagher, P. (2021). Interferences between communications and computations in distributed hpc systems. In *50th International Conference on Parallel Processing*, pages 1–11. 64, 67

[50] Denis, A., Jeannot, E., Swartvagher, P., and Thibault, S. (2020). Using Dynamic Broadcasts to improve Task-Based Runtime Performances. In *Euro-Par - 26th International European Conference on Parallel and Distributed Computing*, Warsaw, Poland. Rzadca and Malawski, Springer. 13, 50, 54

[51] Duran, A., Ferrer, R., Ayguade, E., Badia, R. M., and Labarta, J. (2009). A proposal to extend the OpenMP tasking model with dependent tasks. *Intl. Journal of Parallel Programming*, 37(3):292–305. 15

[52] Edwards, H. C., Trott, C. R., and Sunderland, D. (2014). Kokkos: Enabling manycore performance portability through polymorphic memory access patterns. *Journal of parallel and distributed computing*, 74(12):3202–3216. 1, 8

[53] Ellson, J., Gansner, E., Koutsofios, L., North, S. C., and Woodhull, G. (2001). Graphviz—open source graph drawing tools. In *International Symposium on Graph Drawing*, pages 483–484. Springer. 99

[54] Eschweiler, D., Wagner, M., Geimer, M., Knüpfer, A., Nagel, W. E., and Wolf, F. (2011). Open trace format 2: The next generation of scalable trace formats and support libraries. In *PARCO*, volume 22, pages 481–490. 98

[55] Folk, M., Heber, G., Koziol, Q., Pourmal, E., and Robinson, D. (2011). An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47. ACM. 100

[56] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., et al. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pages 97–104. Springer. 98

[57] Garcia Pinto, V., Schnorr, L. M., Stanisic, L., Legrand, A., Thibault, S., and Danjean, V. (2018). A Visual Performance Analysis Framework for Task-based Parallel Applications running on Hybrid Clusters. *CCPE*, 30. 93

[58] Gates, M., Kurzak, J., Charara, A., YarKhan, A., and Dongarra, J. (2019). Slate: design of a modern distributed and accelerated linear algebra library. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–18. 18

[59] Georganas, E., Gonzalez-Dominguez, J., Solomonik, E., Zheng, Y., Tourino, J., and Yelick, K. (2012). Communication avoiding and overlapping for numerical linear algebra.

In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. 72

[60] Ghysels, P. and Vanroose, W. (2015). Modeling the performance of geometric multigrid stencils on multicore computer architectures. *SIAM Journal on Scientific Computing*, 37(2):C194–C216. 70

[61] Golub, G. and Ortega, J. (1993). *Scientific Computing, an introduction with Parallel Computing*. Academic Press. 70, 73

[62] Graham, R. L., Woodall, T. S., and Squyres, J. M. (2006a). Open mpi: A flexible high performance mpi. In Wyrzykowski, R., Dongarra, J., Meyer, N., and Waśniewski, J., editors, *Parallel Processing and Applied Mathematics*, pages 228–239, Berlin, Heidelberg. Springer Berlin Heidelberg. 8

[63] Graham, S. L., Snir, M., and Patterson, C. A. (2006b). Getting up to speed, the future of supercomputing. *The National Academies Press*. 71, 88

[64] Grubel, P., Kaiser, H., Huck, K., and Cook, J. (2016). Using intrinsic performance counters to assess efficiency in task-based parallel applications. In *IPDPS Workshops*, pages 1692–1701. 94

[Hackbusch] Hackbusch, W. *Multigrid Methods and Applications*. Springer Series in Computational Mathematics Vol. 4, Springer-Verlag, Berlin, 1985. 70

[66] Hackbusch, W. (1999). A sparse matrix arithmetic based on $\mathcal{H}$-matrices, part I: Introduction to $\mathcal{H}$-matrices. *Computing*, 62:89–108. 22

[67] Hackbusch, W., Khoromskij, B., and Sauter, S. A. (2000). On $\mathcal{H}^2$-matrices. In Bungartz, H., Hoppe, R., and Zenger, C., editors, *Lectures on Applied Mathematics*. Springer Berlin Heidelberg. 22

[68] Hagberg, A., Swart, P., and S Chult, D. (2008). Exploring network structure, dynamics, and function using networkx. Technical report, Los Alamos National Lab.(LANL), Los Alamos, NM (United States). 101

[69] Haugen, B., Richmond, S., Kurzak, J., Steed, C. A., and Dongarra, J. (2015). Visualizing execution traces with task dependencies. In *Proceedings of VPA'15*, pages 2:1–2:8. 93

[70] Hénon, P., Ramet, P., and Roman, J. (2002). Pastix: a high-performance parallel direct solver for sparse symmetric positive definite systems. *Parallel Computing*, 28(2):301–321. 18

[71] Heroux, M. A., Thakur, R., McInnes, L., Vetter, J. S., Li, X. S., Aherns, J., Munson, T., and Mohror, K. (2020). Ecp software technology capability assessment report. Technical report, Oak Ridge National Lab.(ORNL), Oak Ridge, TN (United States). 1

[72] Hiraishi, T., Munakata, K., Bai, S., and Ida, A. (2018). Dynamic load balancing for construction and arithmetic of hierarchical matrices. Presented at SIAM Conference on Parallel Processing for Scientific Computing. 23

[73] Hoemmen, M. (2010). *Communication-Avoiding Krylov Subspace Methods*. PhD thesis, USA. AAI3413388. 72

[74] Hoque, R., Herault, T., Bosilca, G., and Dongarra, J. (2017a). Dynamic Task Discovery in PaRSEC: A Data-flow Task-based Runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '17, pages 6:1–6:8. 2, 11

[75] Hoque, R., Herault, T., Bosilca, G., and Dongarra, J. (2017b). Dynamic Task Discovery in PaRSEC: A Data-flow Task-based Runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '17, pages 6:1–6:8, New York, NY, USA. ACM. 21

[76] Hoque, R., Herault, T., Bosilca, G., and Dongarra, J. (2017c). Dynamic Task Discovery in PaRSEC: A Data-flow Task-based Runtime. In *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, ScalA '17. 49

[77] Hoque, R. and Shamis, P. (2018). Distributed Task-Based Runtime Systems - Current State and Micro-Benchmark Performance. In *2018 IEEE 20th International Conference on High Performance Computing and Communications; IEEE 16th International Conference on Smart City; IEEE 4th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*, pages 934–941. 23

[78] Hornung, R. D. and Keasler, J. A. (2014). The raja portability layer: overview and status. 2, 8

[79] Huang, T.-W., Lin, D.-L., Lin, C.-X., and Lin, Y. (2021). Taskflow: A lightweight parallel and heterogeneous task graph computing system. *IEEE Transactions on Parallel and Distributed Systems*, 33(6):1303–1320. 15

[80] Ida, A., Iwashita, T., Mifune, T., and Takahashi, Y. (2014). Parallel hierarchical matrices with adaptive cross approximation on symmetric multiprocessing clusters. *Journal of Information Processing*, 22:642–650. 23, 38

[81] Iwashita, T., Ida, A., Mifune, T., and Takahashi, Y. (2017). Software framework for parallel BEM analyses with $\mathcal{H}$-matrices using MPI and OpenMP. In *Proceedings of the International Conference on Computational Science*, pages 12–14. 38

[82] Kale, L. V. and Krishnan, S. (1993). 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*, OOPSLA '93, page 91–108, New York, NY, USA. Association for Computing Machinery. 14

[83] Kurtz, S., Rain, O., and Rjasanow, S. (2002). The adaptive cross-approximation technique for the 3-D boundary-element method. *IEEE Trans. Magn.*, 38:421–424. 23, 24

[84] Lacoste, X., Faverge, M., Bosilca, G., Ramet, P., and Thibault, S. (2014). Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. In *IEEE International Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, pages 29–38. 91

[85] Legion Team (2019). Legion: Performance profiling and tuning. https://legion.stanford.edu/profiling/. 92

[86] McCalpin, J. D. (1991-2007). STREAM: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia. A continually updated technical report. http://www.cs.virginia.edu/stream/. 80

[87] McKinney, W. (2011). pandas: a foundational python library for data analysis and statistics. *Python for High Performance and Scientific Computing*, 14. 100

[88] OmpSs Team (2020). OmpSs: Instrumentation modules. https://pm.bsc.es/ftp/ompss/doc/user-guide/run-programs-plugin-instrument.html. 93

[89] OpenMP (2015). OpenMP 4.5 Complete Specifications. 1, 8

[90] Patinyasakdikul, T., Eberius, D., Bosilca, G., and Hjelm, N. (2019a). Give mpi threading a fair chance: A study of multithreaded mpi designs. In *2019 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 1–11. IEEE. 10

[91] Patinyasakdikul, T., Eberius, D., Bosilca, G., and Hjelm, N. (2019b). Give MPI Threading a Fair Chance: A Study of Multithreaded MPI Designs. In *2019 IEEE International Conference on Cluster Computing*, pages 1–11. 30

[92] Pei, Y., Bosilca, G., Yamazaki, I., Ida, A., and Dongarra, J. (2019). Evaluation of programming models to address load imbalance on distributed multi-core cpus: A case study with block low-rank factorization. In *2019 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI (PAW-ATM)*, pages 25–36. IEEE. 49

[93] Pillet, V., Pillet, V., Labarta, J., Cortes, T., Cortes, T., Girona, S., and Girona, S. (1995). PARAVER: A tool to visualize and analyze parallel code. Technical report, CEPBA/UPC Report No RR-95/03 February 1995. 93

[94] Russel, B., Danjean, V., and Thibault, S. (2020). Fast user/kernel tracing. https://savannah.nongnu.org/projects/fkt/. 92

[95] Sala, K., Teruel, X., Perez, J. M., Peña, A. J., Beltran, V., and Labarta, J. (2019). Integrating blocking and non-blocking mpi primitives with task-based programming models. *Parallel Computing*, 85:153 – 166. 47

[96] Servat, H., Teruel, X., Llort, G., Duran, A., Giménez, J., Martorell, X., Ayguadé, E., and Labarta, J. (2013). On the instrumentation of OpenMP and OmpSs tasking constructs. In *Euro-Par'12 Wksh*, pages 414–428. 93

[97] Siegel, A., Evans, T., Draeger, E., Deslippe, J., Francois, M., Germann, T. C., Martin, D. F., and Hart, W. (2021). Map applications to target exascale architecture with machine-specific performance analysis, including challenges and projections. 10

[98] Slaughter, E., Lee, W., Treichler, S., Bauer, M., and Aiken, A. (2015). 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. 14

[99] Slaughter, E., Wu, W., Fu, Y., Garcia, N., Kautz, W., Marx, E., Morris, K. S., Cao, Q., Bosilca, G., Mirchandaney, S., et al. (2020). Task bench: A parameterized benchmark for evaluating parallel runtime performance. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE. 23, 49, 71

[100] Snir, M., Gropp, W., Otto, S., Huss-Lederman, S., Dongarra, J., and Walker, D. (1998). *MPI–the Complete Reference: the MPI core*, volume 1. MIT press. 1, 7

[101] Solomonik, E. and Demmel, J. (2011). Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms. In Jeannot, E., Namyst, R., and Roman, J., editors, *Euro-Par 2011 Parallel Processing*, pages 90–109, Berlin, Heidelberg. Springer Berlin Heidelberg. 72

[102] StarPU team (2019a). StarPU: Offline performance tools. http://starpu.gforge.inria.fr/doc/html/OfflinePerformanceTools.html. 92

[103] StarPU team (2019b). StarPU: Online performance tools. http://starpu.gforge.inria.fr/doc/html/OnlinePerformanceTools.html. 92

[104] Torres, H., Papadakis, M., and Jofre Cruanyes, L. (2019). 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. 14

[105] Trefethen, L. N. and Bau, D. (1997). *Numerical Linear Algebra*. SIAM, Philadelphia, PA. 70

[106] Treichler, S., Bauer, M., and Aiken, A. (2014). Realm: An event-based low-level runtime for distributed memory architectures. In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pages 263–276. 14

[107] Treichler, S. J. (2016). *Realm: Performance Portability through Composable Asynchrony*. Phd dissertation, Standford University. https://legion.stanford.edu/pdfs/treichler_thesis.pdf. 92

[Trilinos Project Team] Trilinos Project Team, T. *The Trilinos Project Website*. 18

[109] Trottenberg, U., Oosterlee, C. W., and Schüller, A. (2001). *Multigrid*. Academic Press, London NW1 7BY, UK. 70, 73

[110] Turner, D., Oline, A., Chen, X., and Benjegerdes, T. (2003). Integrating new capabilities into netpipe. In Dongarra, J., Laforenza, D., and Orlando, S., editors, *Recent*

Advances in Parallel Virtual Machine and Message Passing Interface, pages 37–44, Berlin, Heidelberg. Springer Berlin Heidelberg. 80

[111] Valiant, L. G. (1989). Bulk-synchronous parallel computers. In Reeve, M., editor, *Parallel Processing and Artificial Intelligence*, pages 15–22. John Wiley & Sons. 8

[112] Valiant, L. G. (1990). A bridging model for parallel computation. *Communications of the ACM*, 33(8). DOI 10.1145/79173.79181. 8

[Vazhkudai et al.] Vazhkudai, S. S., de Supinski, B. R., Bland, A. S., Geist, A., Sexton, J., Kahle, J., Zimmer, C. J., Atchley, S., Oral, S., Maxwell, D. E., and et al. The design, deployment, and evaluation of the coral pre-exascale systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, SC '18. IEEE Press. 88

[114] Williams, S., Kalamkar, D. D., Singh, A., Deshpande, A. M., Van Straalen, B., Smelyanskiy, M., Almgren, A., Dubey, P., Shalf, J., and Oliker, L. (2012). Optimization of geometric multigrid for emerging multi- and manycore processors. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–11. 70, 72

[115] Williams, S., Watterman, A., and Patterson, D. (2009). Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures. *Communications of the ACM*. 81

[116] Wu, W., Bouteiller, A., Bosilca, G., Faverge, M., and Dongarra, J. (2015). Hierarchical DAG Scheduling for Hybrid Distributed Systems. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 156–165. 35

[117] Yamazaki, I., Hoemmen, M., Luszczek, P., and Dongarra, J. (2017). Improving performance of gmres by reducing communication and pipelining global collectives. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1118–1127. 72

[118] Yamazaki, I., Ida, A., Yokota, R., and Dongarra, J. (2019). Distributed-memory lattice $\mathcal{H}$-matrix factorization. *The International Journal of High Performance Computing Applications*. 22

[119] Yamazaki, I., Thomas, S., Hoemmen, M., Boman, E. G., Świrydowicz, K., and Elliott, J. J. (2020). Low-synchronization orthogonalization schemes for s-step and pipelined krylov solvers in trilinos. In *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing*, pages 118–128. SIAM. 72

[120] Zhang, W., Almgren, A., Beckner, V., Bell, J., Blaschke, J., Chan, C., Day, M., Friesen, B., Gott, K., Graves, D., et al. (2019). Amrex: a framework for block-structured adaptive mesh refinement. *Journal of Open Source Software*, 4(37):1370–1370. 2

[121] Zhang, Y. and Mueller, F. (2012). Auto-generation and auto-tuning of 3D stencil codes on GPU clusters. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, page 155–164, New York, NY, USA. Association for Computing Machinery. 72

[122] Zhao, T., Williams, S., Hall, M., and Johansen, H. (2018). Delivering performance-portable stencil computations on cpus and gpus using bricks. In *2018 IEEE/ACM International Workshop on Performance, Portability and Productivity in HPC (P3HPC)*, pages 59–70. 72, 85

[123] Zheng, Y., Kamil, A., Driscoll, M. B., Shan, H., and Yelick, K. (2014). Upc++: a pgas extension for c++. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 1105–1114. IEEE. 8

[124] Zhu, H., Goodell, D., Gropp, W., and Thakur, R. (2009). *Hierarchical Collectives in MPICH2*, pages 325–326. Springer Berlin Heidelberg, Berlin, Heidelberg. 8

Vita

Yu Pei was born in Shaoguan, Guangdong province, China, in September 3rd, 1990. He attended Sun Yat-Sen University in Guangzhou, China from 2008 to 2013 where he obtained Bachelor's degree in both biotechnology and Statistics.

He obtained his Master degree in Biostatistics from the University of California, Davis in 2015, after that he worked briefly at UC Davis Energy Institute and Oak Ridge National Laboratory. During this period, he developed an interested in efficient statistical computations and the usage of supercomputers.

And he enrolled in the Ph.D. program in Computer Science at the University of Tennessee, Knoxville in Fall, 2016. During his studies, he worked as a graduate research assistant at the Innovative Computing Laboratory (ICL) under the supervision of Dr. Jack Dongarra and Dr. George Bosilca. His research interests focus on high-performance computing, including the evaluation of programming models, optimizations to task-based runtime systems and it's application to efficiently implement numerical linear algebra algorithms. Yu Pei is expected to receive his Doctor of Philosophy degree in Computer Science in December 2022.